Bachelor Thesis

Profiling Concurrency

Konrad Johannes Reiche

July 19, 2011

Supervised by
Prof. Dr. Marcel Kyas

FREIE UNIVERSITÄT BERLIN

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE
INSTITUTE OF COMPUTER SCIENCE

Declaration of Authorship

I hereby confirm that I have written this thesis on my own and that I have not used any other materials than the ones referred to. This thesis has not been submitted, either in part or whole, for a degree at this or any other University.

Konrad Johannes Reiche
July 19, 2011

Abstract

The purpose of this bachelor thesis is the development and usage of a profiler. A profiler is a tool which is used during software development for measuring the execution of an application. Since profiling can be understood in a broad sense, the measurable values differ in the same sense. A profiler could for instance, monitor and trace events, measure the cost of these events, attribute the cost of these events, etc.

The profiler developed within this scope will focus on concurrent events of applications written in Java. The first part of this thesis concentrates on the development of the profiler. The second part concentrates on the usage of the profiler. The profiler is implemented by using the Java Virtual Machine Tool Interface (JVMTI) in C++ and in addition a graphical user interface is implemented in Java by using the Swing library. Furthermore methods were investigated which help to extract knowledge from the gained data.

The developed profiler is able to collect data about threads and monitor locks. In particular the emerging contention between threads will be traced and visualized. Further, the profiler is able to give details about the actions which occurred during the use of thread coordination and synchronization mechanisms. Also the typical profiling feature of measuring the code execution time is provided.

The goal is to strengthen the conceptional understanding of concurrency and identify design flaws in the implementation of the profiled Java application. Thus, different use cases demonstrate the listed features on exemplary Java applications. Finally, the profiler itself and profiling as method will be evaluated.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

One big challenge in computing science is to write correct concurrent programs. Unlike the sequential programming model, which is intuitive and natural, the concurrent programming model is more prone to error. The concurrent programming model allows the execution of multiple computations at the same time. Different interleavings can eventuate into interference if no thoughts were put into the design of accessing multiple resources at the same time.

Quality assurance for concurrent programs is not only a concern of carefully considered program design, even experienced software engineers are facing bugs with the raise of complexity of the developed software [GBB$^+$06, p. xvii]. Testing tools help the developer with automated methods to find bugs, design flaws or performance bottlenecks. One common kind of testing tool at the low level of program analysis is *profiling*.

Program profiling is part of dynamic program analysis and used during software development for measuring the application. Typically profilers aim to measure the frequency and duration of program routines or memory consumption [ASU86, p. 733]. Since profiling can be understood in a broad sense, the kind of measurable values differ in the same sense. The choice of measured values is based on different requirements the profiler should satisfy. A profiler could for instance, monitor and trace events, measure the cost of these events, attribute the cost of these events, etc. [VL00]

In this bachelor thesis, one method of developing a profiler tool for Java programs is investigated with particular emphasis on concurrency-related events. For this the Java Virtual Machine Tool Interface (JVMTI) [Ora07] will be used to implement a native profiling agent[1] in C++. The agent will gather the data and provide them on a communication interface. A client written in Java will process the data and generate even more valuable information. The Java Swing library will be used to implement a visualization of the interpreted data.

This thesis is concerned with the type of profiler which provides information to the developer, as opposed to the type of profiler which is used by a compiler or run-time system

[1] An agent is a component of a software, which is capable of acting autonomously for the user or another program in order to accomplish a task [MM00].

to receive feedback [ASU86, p. 916]. Thus it is also subject to examine the methods in which the collected raw data should be interpreted, especially in how far it is feasible to strengthen the conceptional understanding of concurrency and identify design flaws in the implementation of the profiled Java application.

1.1 Motivation

Today, threads are rarely optional. Even if the Java application is programmed explicitly single-threaded, used frameworks might not be single-threaded. The Java Virtual Machine (JVM) creates threads at the start for maintenance, for instance garbage collection, event dispatching or finalization. When using frameworks for graphical user interfaces like AWT or Swing threads also have to be used in order to create an interactive user experience. The same applies to Java RMI or Java Servlets [GBB^{+}06, p. 9].

Using multiple threads sharing resources without proper synchronization will eventually lead to concurrency bugs due to non-deterministic scheduling. Concurrency bugs are, unlike memory and semantic bugs, a lot harder to detect. On the one hand they are usually under-reported and on the other hand it is hard to understand their patterns and manifestations due to their emergent nature involving interactions among multiple components [LPSZ08].

Thus it is no coincidence development teams are facing more and more bugs which emerged due to thread-related code. On the NetBeans developer it was reported, how they tried to patch a single class over 14 times in order to fix a threading problem [GBB^{+}06, p. xvii].

It is obvious, that a profiler cannot make up for the lack of understanding of how code is implemented in a thread-safe way. The profiler, however, can deliver results which could be used to check, whether the applied synchronization operate in the way the developer intended to. For instance the profiler could check the developers intentions in the following way:

- whether code parts were ever accessed by more than one thread at the same time
- whether code parts were ever accessed by more than one thread at all
- how many locks were used
- how many threads have contended for the same lock
- with which threads did another thread contend for a lock
- which actions happened during thread coordination

Problems with concurrency in practice are not limited to newcomers. Even experienced programmers face emergent errors which are rooted in threading and their interaction [GBB^{+}06, p. xvii]. This is due to applications growing very large. A large application will consist of many components interacting with each other. For the developer this is an additional burden, because designing thread-safe classes requires a more sophisticated design compared to non-thread-safe classes.

In order to expose those flaws testing is applied during software development. Existing testing techniques, however, address mainly sequential aspects of applications (statements, branches, etc.) instead of concurrent aspects (interleavings) [MQ07]. Concurrent aspects cannot be effectively countered, because the number of interleavings grows exponentially to the number of threads. This is in particular true for the mentioned large growing applications.

Profiling offers an alternative approach: instead of testing the application for single statements and their results, profiling deals with testing whole parts of the program. The ability to monitor or to measure code and to diagnose errors is called *code instrumentation* [SE04]. A common practice is to measure the execution by adding additional code, for instance code which prints the current time or the current stack trace. It's a manual process, which requires to determine the target code, instrument the code, compile the code, run the application, analyze the results and remove the code instrumentation afterwards. When the next problem occurs all the steps have to be repeated. This does not give a comprehensive overview of how the different parts of the application are performing together [Wil07].

Writing a standalone profiler avoids the problems of ad-hoc measurement because the application does not have to be modified. The tool is injected into the application and starts the code instrumentation automatically. Several snapshots of the applications form a profile of the application. The task of the profiler is to summarize the data in a meaningful way. The goal is to generate an overview of the whole application at run-time. The question is not, what statement can be made about some single lines of code, but what statement can be made about these single lines of code in the picture of the whole application.

This is especially useful when concurrency is used to exploit multiprocessor architecture. When applied inadequately this can lead to inefficient code. Profiling can help to unveil hot spots which cause this flaw. In order to improve the code it has to be measured.

1.2 Problem

Susan Graham, Peter Kessler and Marshall McKusick distinguish in their article *gprof: a call graph execution profiler* between two kinds of profiles: those which present counts of statements or routine invocations and those which present timing information about statement or routines [GKM04]. What both kind of profiles have in common is the concept of time. The profiled data is always related to a period of time. A profiler could measure a program from start to termination, but also for a short period during the program execution.

The same applies when profiling data is related to concurrency. Therefore the events are traced throughout the program execution. The question is: which kind of information should be traced in order to reveal information which help to understand the concurrent program behavior? Suitable data would include information about threads and their interaction with each other. Since thread interaction is performed by using synchronization, the profiler should make access to synchronization structures visible. For instance considering the following program in listing 1.1 which was adapted from the book *Java Concurrency in Practice* [GBB+06, p. 72]:

```java
public class EnhancedSynchronizedList<T> {
  public List<T> list =
    Collections.synchronizedList(new ArrayList<T>());

  public synchronized boolean putIfAbsent(T x) {
    boolean absent = !list.contains(x);
    if (absent) {
      list.add(x);
    }
    return absent;
  }
}
```

Listing 1.1: Extending a thread-safe class

A thread-safe list is extended by another class adding the method *putIfAbsent*, which adds only an element to the list if it is not present. Extending a thread-safe class does not imply the new class is thread-safe, too [GBB+06, p. 72]. The profiler should be able to reveal information with respect to the problem of thread-safety. This includes single events related to the synchronization, access to locks and thread state changes, but also acquiring and releasing a Java monitor, waiting and signaling on a Java monitor and the use of higher-level synchronization constructs, for instance *java.util.concurrent.Semaphore.* Further the profiler should be able to visualize the data in order to offer alternative presentations. With respect to synchronization the profiler should display the shared resources and the threads' access to it by using a resource-allocation graph[2].

1.3 Solution

In order to realize profiling an API provided by Oracle will be used: Java Virtual Machine Tool Interface (JVMTI) [Ora07]. JVMTI is a native programming interface for implementing tools and provides a rich set of functions for both: querying and controlling the JVM and reacting to occurring events. For a more detailed insight, methods have to be explored which interpret the data received through the JVMTI. This deals especially with understanding how concurrency is working in the Java programming language. The purpose of this work is

[2]A system resource-allocation graph is a directed graph and used to describe deadlocks [SG98, p. 210].

to research these methods. The agent runs in the same process as the JVM being examined. Therefore the architecture will consist of two parts: a front end and a back end. This reduces the already present interference with the target application's execution.

Back End A native profiling agent which gathers the relevant data.

The gathering of relevant data is not limited to functions provided by the JVMTI. First, any useful data receivable through the JVMTI has to be stored. Then the stored data has to be refined, for instance by linking different data together.

Front End A graphical user interface which interprets and prepares the data in order to display the information in a meaningful way.

The second component of the profiler is the visualization in the graphical user interface. Often it is enough to present the information in a formatted way, for instance in a tabular form. Additional illustration, however, for instance by using diagrams, can improve the understanding even better.

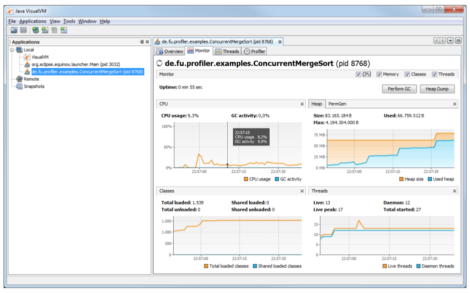
Thus the profiler uses a client-server architecture. The native profiling agent acts as the client sending the gathered data to the server. The server is the front end which processes the results.

Before presenting the solution in detail a brief overview over the backgrounds is given in chapter 2. In chapter 3, the profiler is discussed in full length in means of software engineering, including requirements, architecture, design and implementation. In chapter 4 the profiler usage is evaluated on different use cases. In the end a brief overview about the results is summarized and concluded in chapter 5.

1.4 State of the Art

There are many tools which have proven to be effective over time, but are still not complete and therefore in the process of development. Since this work focuses mainly on concurrent aspects of profiling, it would be interesting to see in how far these current profilers scope with this task. Two well known profilers are VisualVM and YourKit, both offering a graphical user interface.

VisualVM, which can be seen in figure 1.1, is part of the JDK and thus free and also open source [Ora11b]. As most profilers, VisualVM offers the ability to monitor the performance, memory consumption and thread activity. VisualVM allows it to produce a thread dump which includes a stack trace of every thread and whether a thread is in contend with other threads and if a deadlock was discovered. The thread activity is made understandable with different graphs and statistical information, for instance how long a thread was blocked.


(a) Monitor application performance and memory consumption

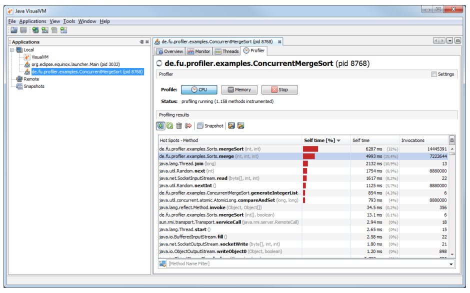
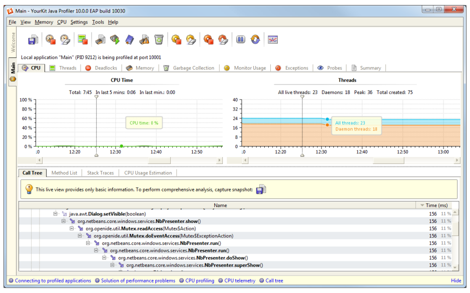

(b) Profile application performance and analyze memory allocation

Figure 1.1: VisualVM – a debugging and performance analysis tool


(a) Call graph shows which methods were called from certain methods

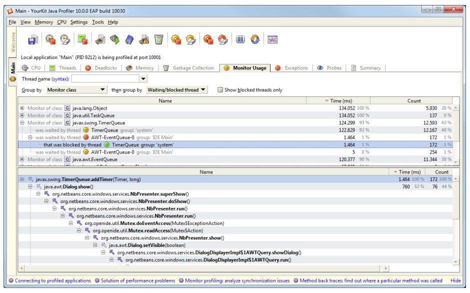

(b) Monitor profiling allows analysis of synchronization issue

Figure 1.2: YourKit – a Java and .NET profiler

For YourKit, which can be seen in figure 1.2 the purchase of license is required and it is closed source [You11]. It is more interesting than VisualVM as the developers have focused more on the concurrency aspects of Java. YourKit offers the same kind of features listed for VisualVM. In addition it deals with the profiling of Java monitors. Monitor profiling in YourKit helps with analyzing inconsistent synchronization. It shows which thread has called *Object.wait*, for how long and which threads were blocked on attempt to acquire an owned monitor lock. Especially for the last feature an attempt will be made to adapt this for the profiler of this work. YourKit is also able to construct call graphs[3]. YourKit distinguishes the call graphs by threads. Thus statements can be made about threads, for example which code was executed by a certain thread. Overall the offered features in the present profilers are quite limited with respect to concurrency aspects, when taking into account, that the *java.util.concurrent* package is not considered, although it offers a standard set of concurrency utilities which ease the task of developing multithreaded applications on a higher level of abstraction.

[3]A call graph is a directed graph. A node represents a subroutine and an edge between two nodes indicates, that one subroutine has called the other subroutine [GKM04].

Chapter 2

Background

In order to understand the problem domain, selected topics will be presented. Firstly, the basics will be given to understand how threads work in the Java programming language. Then the general synchronization concept in the Java programming language is introduced: Java monitor. Since the profiler focuses on waiting and signaling events of monitors, the concept of condition queues is explained as well. In the end an introduction to the Java Virtual Machine Tool Interface is given to reach an understanding of the used architecture. This will serve as a base to understand the implementation of the profiler which is presented in Chapter 3.

2.1 Java Threads

Threads are created and manged by the classes *java.lang.Thread* and *java.lang.ThreadGroup*. A thread can only be created by instantiating an object of the class *Thread* and is not yet active before the method *Thread.start* was invoked. What code the thread executes is defined in the method *Thread.run*. The original implementation of *Thread.run* is empty. Therefore this method has to be overridden by subclassing *Thread* or by providing a *Runnable* object. *ThreadGroup* represents a set of threads. A *ThreadGroup* can have other *ThreadGroup* objects. Thus it forms a tree data structure. A thread is allowed to access information of its own thread group, but not the information of any other thread group [AGH00, sec. 14.11].

Every thread has priority which gives a preference to the scheduler indicating whether a thread should be preferred over another thread. This priority can, however, make no guarantees. Thus priority cannot be used to write reliable code, for instance for mutual exclusion [GBB$^+$06, p. 218]. Every thread has a state. The state describes the thread in scope of the JVM, not in scope of the underlying operating system. A thread is always in one, and only in one, of the following states:

New A newly created thread, which has not started yet.

Runnable A thread which has been started, but has not terminated yet.

Blocked A thread which is waiting to acquire a monitor lock.

Waiting A thread which is waiting for another thread to perform a particular action.

Timed Waiting A thread which is waiting for another thread to perform a particular action or until the specified time has elapsed.

Terminated A thread which has been terminated.

A common confusion is to think, that a *Runnable* thread is actually executing code. Looking at the state definition again it should be clear that there is no state specified for a thread executing code. Another confusion is to think, that a thread which waits on a condition to become true will always be in the state *Waiting* or *Timed Waiting*. There is no specification which requires a thread ever to enter *Waiting* or *Timed Waiting*. This is due to two reasons. On the one hand the JVM could implement blocking by spin-waiting. On the other hand spurious wake-ups from *Object.wait* and *Condition.await* are permitted which makes it possible for a thread being more than one time in transition between *Runnable* and *Waiting* or *Timed Waiting*, although the condition has not become true yet [GBB+06]. Nevertheless a graph in figure 2.1 sketches the life cycle of a Java thread to give a conceptional idea of the possible state transitions.

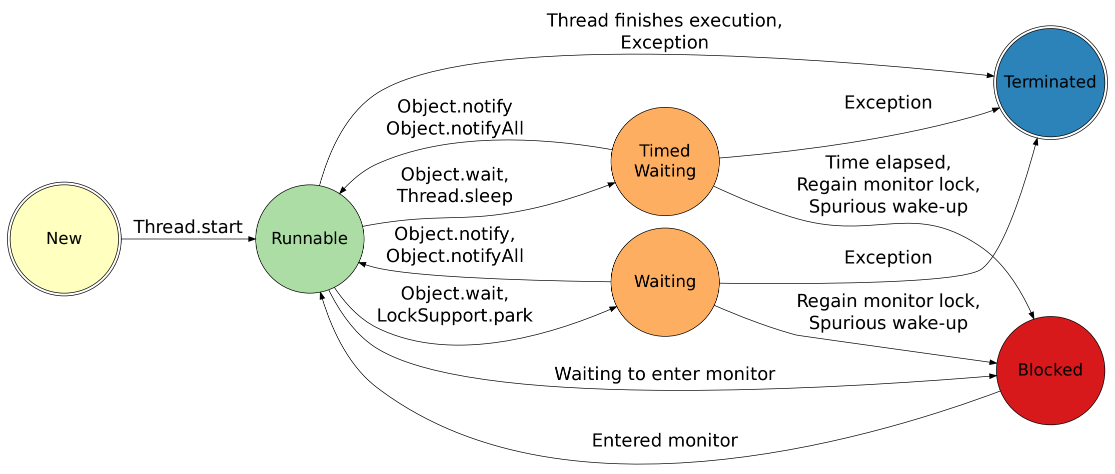

Thread finishes execution, Exception
Terminated
Object.notify Object.notifyAll
Exception
Timed Waiting
Object.wait, Thread.sleep
Time elapsed, Regain monitor lock, Spurious wake-up
New
Thread.start
Runnable
Object.notify, Object.notifyAll
Waiting
Exception
Object.wait, LockSupport.park
Regain monitor lock, Spurious wake-up
Blocked
Waiting to enter monitor
Entered monitor

Figure 2.1: Life cycle of *java.lang.Thread*

When a thread has become *Waiting* or *Timed Waiting* and was notified or the timer of *Timed Waiting* has elapsed the thread tries to reacquire the monitor lock, because it was released after invoking *Object.wait*. Without this behavior, no other thread could succeed as one thread has to own the objects monitor lock in order to make invocations to *Object.notify*, *Object.notifyAll* or *Object.wait* [GBB+06, p. 297].

Threads are a fundamental part in profiling. Every thread consumes a considerable amount of memory when being created. Also taking thread scheduling into account context switches are expensive, too. When a thread blocks, for instance due to waiting for a monitor which is held by another thread, the scheduler usually suspends the thread. If threads block frequent, this incurs a lot context switches, thus increasing the scheduling overhead and reducing the throughput. The actual cost of a context switch is depended on the used platform. It is said a context switch costs about an equivalent of 5.000 to 10.000 clock cycles or several microseconds [GBB$^+$06, p. 230].

Every thread is associated with a call stack. A Java thread computes a stack trace based on this call stack. For debugging purposes stack traces are crucial and should not be omitted. Knowing in what context a thread is executing adds valuable semantic to the tracked events. A stack trace consists of one or more stack trace elements. A stack trace element contains information about the class name, the file name, the line number, the method name and whether it is a native method [AGH00, sec. 12.6].

2.2 Java Monitors

A *critical section* is code which may access a shared resource, but must not be accessed by more than one thread at a time. To prevent interference the access to these critical sections is synchronized. In Java the equivalent action to enter a synchronized critical section is to acquire the lock of an object. The equivalent action to leave a synchronized block is to release the lock of an object. Every *java.lang.Object* has a lock associated with it. The lock can be acquired and released by using the *synchronized* statement in methods [AGH00, sec. 14.3]. These built-in locks are called *intrinsic locks* or *monitor locks* [GBB$^+$06, p. 25].

Locks are owned per thread. Thus a thread which invokes more than one time synchronized on the same object within the same method can proceed without blocking. The acquired lock is not released until the outermost *synchronized* is exited. Throwing an exception releases a lock as well. Since the *synchronized* statement includes both, acquiring and releasing a lock, it is impossible to forget releasing a lock in the end [GBB$^+$06, p. 25].

When publishing and sharing objects the use of *synchronized* is not only used for atomicity or demarcating critical sections. A significant effect of using *synchronized* is *memory visibility.* Thus in addition to prevent threads from modifying objects when they are in use by other threads, it is also required that other threads can see the new state of the object [GBB$^+$06, p. 33].

With respect to profiling, developers are concerned with two issues: the performance impact due to monitor contention and the reasons for occurring deadlocks [VL00]. One reason to use profiling in section 1.1 was motivated through the lack of scalability in multiprocessor systems. This is mainly due to too much monitor contention.

Monitor Contention Evolves when a thread holds a monitor lock and one or more threads become blocking for trying to acquire this monitor lock. Monitor contention evolves typically when a thread holds a monitor too frequently or too long.

Synchronization to achieve mutual exclusion on critical sections is often not sufficient to implement concurrent applications. Sometimes threads have to wait for a certain condition to become true. In avoidance of using busy waiting this requires the threads to communicate with each other. In order to implement thread communication one could use the waiting and signaling events of the Java programming language provided by a monitor. Even though it is recommended to use higher-level locking primitives instead [GBB+06, p.300], using *Object.wait*, *Object.notify* and *Object.notifyAll* is still a common way to implement thread communication. In addition many Java library classes make use of these methods. Usually this is not visible to the developer. Hence, it is another motivation for the profiler, as the profiler should make this use visible. Since some of the examples in section 4.2 make use to this paradigm as well it will be explained briefly.

2.3 Condition Queues

Using the methods *Object.wait*, *Object.notify* and *Object.notifyAll* is bound to a specific pattern: *condition queues*. Condition queues allow threads to wait for a specific condition to become true. Thus the elements of a condition queues are not the objects, but the threads themselves. As every object is able to act as a lock, every object is able to act as a condition queue, too. Both concepts are tightly bound to each other. In order to wait on an object for a condition, the thread has to own the lock of the object. Waiting for a state-based condition and preserving the state consistency depends on each other. [GBB+06, p. 297]. In figure 2.2 it is illustrated how the monitor lock mechanism and the waiting and signaling are functioning together.

Object.wait The current thread starts to wait until another thread invokes the *Object.notify* method or the *Object.notifyAll* method. It atomically releases the lock and requests the underlying operating system to suspend the thread.

Object.notify Wakes up a single thread that is waiting on the objects' monitor. Upon waking, the notified thread will try to reacquire the lock before returning.

Object.notifyAll Wakes up all threads that are waiting on the objects' monitor. Upon waking, the notified thread will try to reacquire the lock before returning.

The reason why it is recommended to use higher-level locking primitives is due to the easy incorrect use of condition queues. One of the reasons is due to the fact, that *Object.wait* can

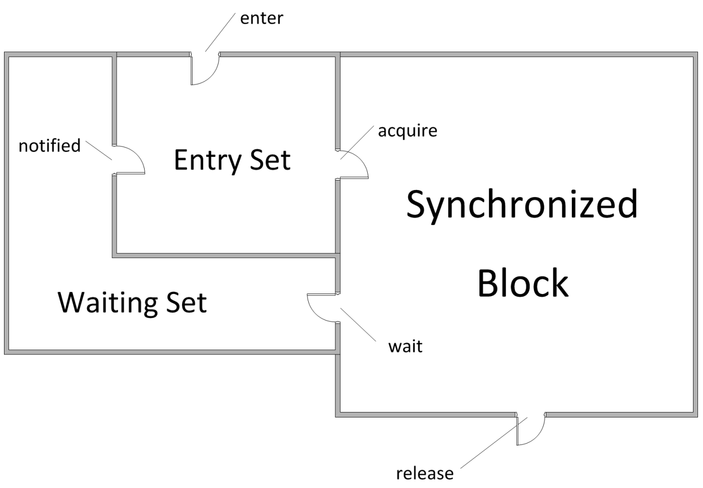


Figure 2.2: A Java style monitor

return although the condition has not become true yet, as the method is allowed to return spuriously. Another reason is that the object could be associated to more than one condition predicate. If a thread gets notified it is also not clear, whether the condition has become true. *Bounded Buffer*[1] is an example for one condition queue associated with two predicates: *not full* and *not empty*. These are many reasons why the following canonical form shown in listing 2.1 should be applied. The listing is adapted from the book *Java Concurrency in Practice* [GBB+06, p. 301], but the general structure is also described in the Java Platform Standard Edition 6 API Specification [Ora11a].

```
void stateDependentMethod() throws InterruptedException {
  // condition predicate must be guarded by lock
  synchronized (lock) {
    while (!conditionPredicate()) {
      lock.wait();
    }
    // object is now in desired state
  }
}
```

Listing 2.1: Canonical form for a state-dependent method

This alone, however, is not sufficient to avoid incorrect use. Great care has also to be taken when performing the notifications. First of all, whenever the condition predicate becomes true someone has to perform a notification. Second, as a thread cannot return from *Object.wait* without reacquiring the lock, the thread doing the notification should immediately release the lock. In general *Object.notifyAll* should be preferred over *Object.notify*

[1] A bounded buffer is a multislot communication buffer which is shared between threads. Producer threads deposit objects in the buffer and consumer threads fetch them. The buffer has a queue containing those objects which have not been fetched yet [And99, p. 161].

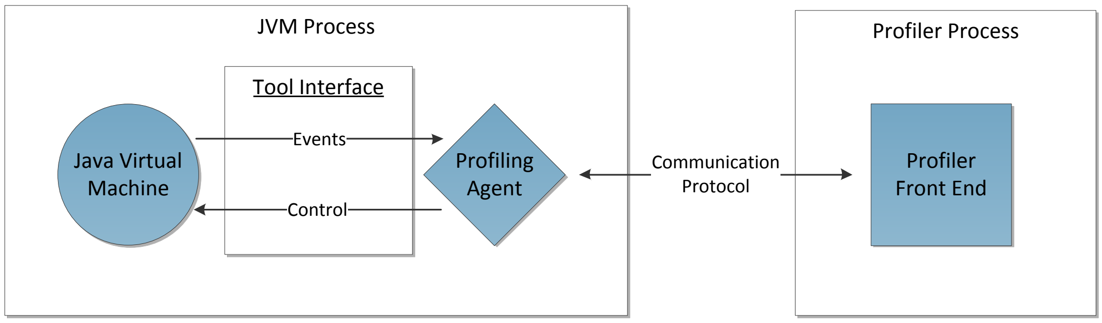


Figure 2.3: Overall architecture proposed by the design of the JVMTI

as a liveness failure named *missed signals* could occur [GBB+06, p. 301]. Single notifications can be used over *Object.notifyAll* when only one condition predicate is associated with the condition queue, each thread executes the same logic when returning from *Object.wait* and a notification enables at most one thread to proceed. Most of the time this is not case [GBB+06, p. 303].

With respect to profiling applications it is obvious, that *Object.notifyAll* is very inefficient and causes more performance impact compared to single notifications. It should be task of the profiler to make this performance impact visible and measure it to some extend.

2.4 Java Virtual Machine Tool Interface

With Java 5 the Java Virtual Machine Tool Interface (JVMTI) was introduced which replaced the Java Virtual Machine Profiling Interface (JVMPI) and the Java Virtual Machine Debug Interface (JVMDI). The JVMTI is a native programming interface to create tools for inspecting the state and to control the execution of an application running in a JVM. Therefore it can be used to create tools which can be used for: profiling, debugging, monitoring, thread analysis, and coverage analysis tools [Ora07]. The following section summarizes the article *Java Virtual Machine Profiler Interface* [VL00] describing the general design and architecture of the interface. Only slight adaptions were made, since the basic ideas did not change in the transition from JVMPI to JVMTI [O'H11].

The JVMTI offers an interface to support the development of a profiler. This is an alternative approach to direct profiling support. This way different profilers can be used with the same Java Virtual Machine and the same profiler can be used with different virtual machines – it decouples the JVM from the presentation of the profiling information. Figure 2.3 illustrates the typical overall architecture.

The profiler agent and the Java Virtual Machine are executing in the same process. The binary function-call interface enables both to interact with each other. The profiling data is sent, by using a communication protocol, to the profiler front end. The profiler front end presents the data. In general the profiler agent is implemented as a shared library. This agent is loaded together with the Java Virtual Machine. The JVM looks for a specified entry hook and both, agent and JVM, initialize a pointer table for the provided JVMTI functions [VL00].

The JVMTI belongs to the event-based profilers. Thus event hooks are provided by the interface which can be used to obtain more specific data related to the triggered event. Every event type has to be activated separately in the profiling agent in order to receive notifications. This allows it to minimize the overhead from the beginning. Every event callback includes a JNI[2] environment pointer in order to facilitate JNI usage. To give an idea of the event mechanism some examples are given [Ora07]:

Thread Start Thread start events are generated by a new thread before its initial method executes.

Thread End Thread end events are generated by a terminating thread after its initial method has finished execution.

Method Entry Method entry events are generated upon entry of Java programming language methods (including native methods).

Method Exit Method exit events are generated upon exit from Java programming language methods (including native methods). This is true whether termination is caused by executing its return instruction or by throwing an exception to its caller.

Monitor Contended Enter Sent when a thread is attempting to enter a Java programming language monitor already acquired by another thread.

Monitor Contended Entered Sent when a thread enters a Java programming language monitor after waiting for it to be released by another thread.

How the callback functions are implemented is shown exemplary in listing 2.2. The callback function provides a pointer to the JNI and JVMTI environment. Further function parameters offer information about the present event. In this case it is possible to gain information about the currently executing thread and the method which is entered.

The design for the JVMTI is sophisticated with respect to cost attribution to specific execution contexts, especially in a multithreaded environment. The developers have decided to present information at a method call level, since they believe there is little reason to attribute cost to a finer granularity than a method, the source code line number and where the method was executed.

[2]Java Native Interface is a framework which enables the developer to make calls from a native application to a Java application and vice versa.

```
static void JNICALL callbackMethodEntry(jvmtiEnv *jvmti_env,
    JNIEnv *jni_env, jthread thread, jmethodID method) {

  enter_critical_section(jvmti);
  {
    jvmtiThreadInfo threadInfo;
    char *methodName;

    jvmti->GetThreadInfo(thread, &jvmtiThreadInfo);
    jvmti_env->GetMethodName(method, &name, NULL, NULL);

    std::cout << threadInfo.name << " " << methodName << std::endl;
  }
  exit_critical_section(jvmti);
}
```

Listing 2.2: JVMTI callback function which prints the name of the executing thread and the current method name every time a method is entered

It is argued, that a flat profile, which would only consist of the methods' execution time, does not contribute enough information. For instance, the fact that a time consuming method was invoked has little use. To counter this problem, it is suggested to make use of the *GetStackTrace* function. The function generates the dynamic stack trace of a thread. This way it is possible to learn which part of the program contributed to the invocation of the time consuming method. The concept of adding a stack trace in order to learn about the context is a general strategy to improve the information gain [VL00].

2.4.1 Thread-Safety

The profiling agent interacts with a multithreaded environment, on this account the agents design has to be thread-safe as well. This applies in particular when accessing static or extern data within the event callbacks, but also when calling native system functions. Ideally all event callback functions are designed in a re-entrant style. The simplest way to implement this is to create a single *raw monitor*[3] at the beginning and use this monitor in every callback function. This assures: only one callback function is active at any given point in the time. The use of a raw monitor can already be seen in figure 2.2. The actual function code is wrapped inside two function calls which acquire and release the monitor lock. Their definitions can be seen in listing 2.3.

2.4.2 Usage

Different functions provided by the JVMTI facilitate the implementation of different kind of profilers. The task of profiling the methods' execution time can be realized by using code instrumentation: measure the time by calculating the system time difference between

[3]A raw monitor is basically a Java monitor in a native environment.

```
static void enter_critical_section(jvmtiEnv *jvmti)
{
  jvmtiError error;
  error = jvmti->RawMonitorEnter(gdata->lock);
  Agent::Helper::checkError(jvmti, error, "Cannot enter with raw monitor");
}

static void exit_critical_section(jvmtiEnv *jvmti)
{
  jvmtiError error;
  error = jvmti->RawMonitorExit(gdata->lock);
  Agent::Helper::checkError(jvmti, error, "Cannot exit with raw monitor");
}
```

Listing 2.3: Defining functions for acquiring and releasing a raw monitor in order to establish mutual exclusion

entering the callback function for *MethodEnter* and *MethodExit*. Another typical task is heap profiling. Heap profiling can be implemented by using bytecode instrumentation[4] to tag objects on every allocation, react to the event *ObjectFree* and obtain the stack trace to identify busy heap allocation sites.

Taking concurrency into account thread and monitor profiling is especially interesting. Frequently contended monitors can be tracked by hooking to the *MonitorContendedEnter* event and monitors being held for a relative long time can be tracked by hooking to the *MonitorContendedEntered* event [VL00]. Thread profiling can be implemented by tracing the threads states. The thread states primary utility is as a source of debugging information [GBB$^+$06, p. 251]. In JVMTI a threads state can be received by the function *GetThreadState*. The returned value can be classified within the decisions in figure 2.4.

Within the Java programming language a threads state can also be received by invoking *Thread.getState*. The state defined in *java.lang.Thread.State*, however, is only a subset of the states returned by the JVMTI function. The state received by the JVMTI function can be mapped to *java.lang.Thread.State* by using appropriate conversion masks [Ora07]. This raises the question, whether the profiling agent should deliver only the mapped states or the JVMTI states. Since the set of JVMTI states is too versatile an intermediate solution is chosen. The profiling agent will basically deliver the virtual machine state. When appropriate another state might be returned, for instance when the thread is sleeping, the delivered state will be *Sleeping* instead of *Timed Waiting*.

2.4.3 Overhead

Injecting a native profiling agent into a Java Virtual Machine influences the program execution no matter what. Thus using a profiler adds overhead. Native code is executed between

[4]Bytecode instrumentation is a process where additional bytecode is added to the bytecode of a set of classes before they are loaded by the virtual machine.

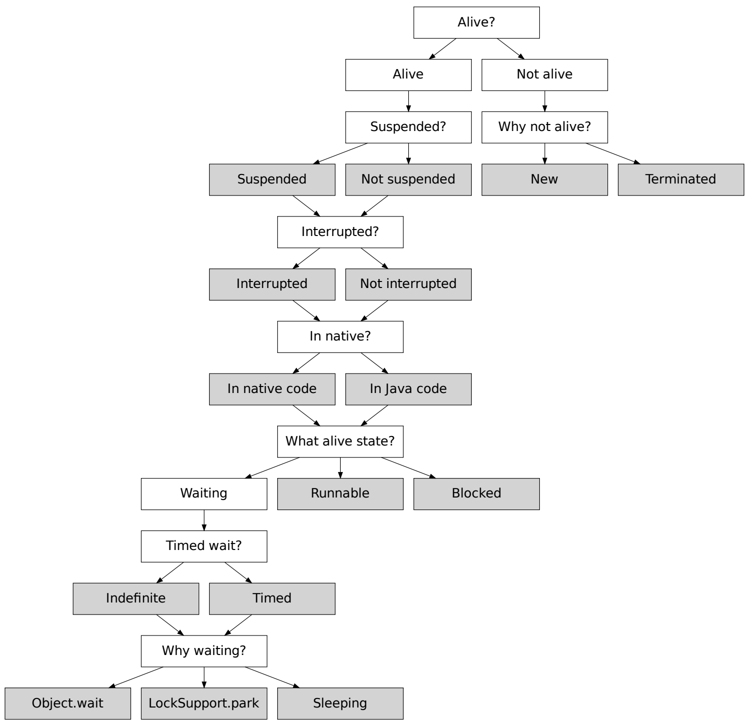


Figure 2.4: Java thread state according to the JVMTI

Java code, bytecode is used to instrument Java classes or further agent threads are launched which execute their tasks concurrently in addition to the rest of the Java threads. Most of the time the profiling agent will slow down the Java program significantly. For example, instrumenting every method called in Java can easily slow down the execution by multiple times. The consequence is, that the reported execution time or measured clock cycles become meaningless. Sophisticated overhead calculations could help to achieve real results. The fact, however, that the L1 cache is changed due to instrumentation and that the Java code is less efficient, will not change.

Unfortunately, this code can also introduce timing or synchronization artifacts. The possibility remains that these artifacts mask bugs which would occur otherwise. Bugs that disappear in this manner are called *Heisenbug* [Ray96]. The problem will be generalized: every alteration of the program execution due to the profiler agent will be considered as *Heisenbug*, too. This includes, but is not limited to: alteration of the thread scheduling, differences in the occurrence of certain interleavings and changed execution time. In section 4.4 an attempt is made to measure the described overhead which is produced by executing the Java program while the profiler agent is loaded.

Chapter 3

Profiler

The development of the profiler can be understood in means of a software engineering process. It is a piece of software with certain requirements, a chosen architecture and design and a sophisticated implementation. This chapter focuses on the problems which were faced during the development process and how they were approached.

3.1 Requirements

The ideal scenario is a program that automatically displays all the important data to the user in a convenient and informative way. The important data are those which let the user understand his profiled program with no questions left. Unfortunately it is not clear, what are the important information. On the one hand this is due to the variety of programs and on the other hand this is due to the number of collected data. Firstly, the task will be to collect enough information with respect to concurrent Java and secondly, offer possibilities to filter the information when there are too many which would otherwise result in an information overload. The functional requirements are addressing the problem statement in section 1.2. They are described in the following:

Thread Overview A general overview of the threads should be displayed, including their names, priorities and especially its state.

Monitor Overview A general overview of the monitors should be displayed, including the threads relation to the monitor.

Monitor Log Events which involve a Java monitor should be presented, for instance lock acquisition, waiting and signaling, or thread contention.

Visualization The profiled information should be visualized in order to ease the understanding of the data. For instance a resource-allocation graph can be used to display the threads and how they are related to the present locks.

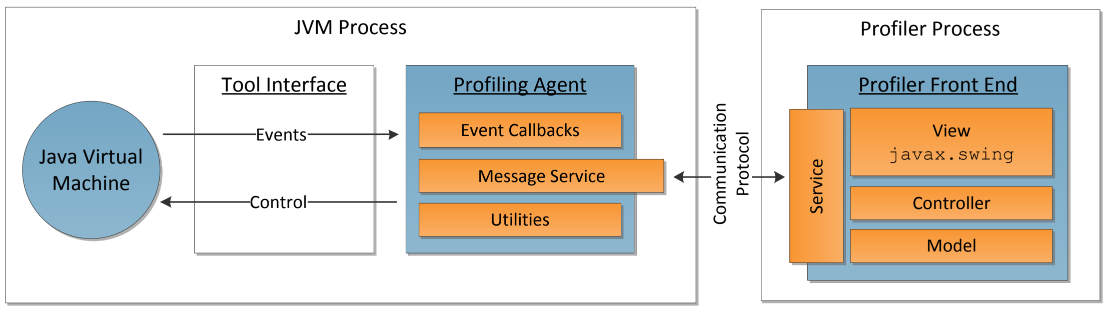


Figure 3.1: Architecture of the profiler

Stack Traces Extensive use of stack traces should be made. This way the profiled data can be put into a context.

Method Profiling Measuring how much time was spent in which part of the program defined on a method call level.

Logging Logging of every recorded event in order to cope with the number of data. This way it is possible to go back in time and inspect a past state of the profiled application.

Non-functional requirements should be:

Efficiency As profiling has a great impact on the execution, the overhead should be minimized as good as possible.

Portability The profiler should be usable on different platforms which support the Java platform as well.

3.2 Architecture

As proposed by the article *Java Virtual Machine Profiler Interface* [VL00] the profiling agents front end typically executes in its own process. This is due to two reasons. Firstly, profiling has already strong impacts on the execution of programs. Using a graphical user interfaces would increases the impact even more. Secondly, the target virtual machine might not run on the local machine. A profiling agent with a graphical user interface would be useless when started on a distant server machine. Thus the architecture illustrated in figure 2.3 will be maintained.

A more detailed illustration of the architecture can be seen in figure 3.1. The architecture consists of three components: the profiling agent, a communication protocol and the Java front end. The agents inner structure is based on three components as well. The core of the implementation manages all the event callbacks. Auxiliary functions can be found in the namespace *Utilities*. They encapsulate functions which help to extract more information out of the data provided by the JVMTI. The agent uses a client socket which is located in the *Message Service* class. The data is prepared and serialized into a message which implements the communication protocol. The message is sent via the socket to the target host.

The Java front end listens on a server socket for incoming messages. The task for communication is encapsulated in the *Service* package. Every received message is deserialized, processed and displayed on the graphical user interface. Since the Java front end uses Swing to implement the visualization the overall structure is based on the model-view-controller architecture. The model is basically the representation of the profiled JVM.

In the decision of the transport protocol TCP was chosen over UDP as packet loss and packet duplicates cannot be tolerated. When measuring the relative execution of a method this is omittable. When a message is missed out which contains information about a monitor action problems can evolve. On the one hand it would be confusing for the user as it is expected to see every possible step of the synchronization. On the other hand this could lead to errors due to message processing in the front end, for instance, when constructing the resource-allocation graph.

The task of the Java front end is not limited to displaying the received data. The data is vague and leaves room for interpretation. Therefore also the Java front end implements logic for processing the data. For instance, message can contain information about threads and their current state. The Java front end watches the threads for state changes and when they occur, the total time a thread has spent in the previous state is calculated.

3.3 Design

The design is a refinement of the requirements and the architecture so far. It gives a more concrete description of how the functionality should be implemented. Since the profiler deals with different kind of data, it is obvious to present these data in different views. These views are separated by the problems the profiler ought to solve. With respect to the graphical user interface the views are separated by different tabs. There are two kinds of data: data which is counted, for example how often a certain event has occurred and data which relates to a greater context, for instance an event which occurred and influenced a present object. Information within a greater context is modeled as log entry. Counts and log entries are typically presented in a tabular form. When it is feasible, a more graphical representation is chosen instead.

3.3.1 Thread Overview

The thread overview view should give a brief overview of the threads which were active at a time during the profiling. This includes a list of all threads and their available information describing their state. Further these information are visualized: on a pie chart diagram the threads' state distribution can be tracked and on another chart the thread state over time is displayed. This view serves as an entry point for the user to see which threads are involved. The thread state visualization might indicate the current behavior. A pie-chart with the majority of threads being blocked indicates heavy monitor contention.

3.3.2 Monitor

The JVMTI allows to gain additional information about the usage of a Java monitor. The information is limited to the following: the number of times the owning thread has entered the monitor, the number of threads waiting on a monitors condition and the number of threads waiting to be notified by other threads. For the waiting threads and threads waiting to be notified a list of pointers to the actual JNI thread object *jthread* is also receivable. In the requirements it was stated that stack traces will be used as an overarching concept. Since the actual threads waiting can be gained, their current stack traces will be presented, too. This will help to learn where and maybe why the threads are waiting.

3.3.3 Monitor Log

There are two kinds of actions which are performed in context of a certain monitor: locking and waiting and signaling. The locking will be displayed in means of acquisition and release of locks and the waiting and signaling will be displayed in means of invocations to *Object.wait*, *Object.notify* and *Object.notifyAll*. For ordering purposes timestamps will be used to uniquely identify each event. Since profiling is about measuring, the time spent during two events will be calculated and displayed. It was discussed how the number of events can easily lead to an information overload. A typical method to counter this issue is filtering. Therefore it is possible to filter the monitor log by event types and by threads. This way, for instance two threads and their actions can be easily compared.

3.3.4 Event Log

When profiling, logging is an indispensable requirement. It is not sufficient to display just the current state as the graphical user interface is updated too frequently. Therefore all recorded events should be logged. In order to do so, the following design approach was chosen. The JVMTI is an event-based profiler gathering the information for each occurred event. Therefore the data is logged per event, too. The following event types are distinguished:

Thread Event A thread event contains information about a threads' life cycle, for instance whether a thread has started, has terminated or its state has changed.

Monitor Event A monitor event contains information about a Java monitor, for instance locking or signaling and waiting. An example of locking is when a thread cannot enter a monitor since it is owned by another thread. In this case the owning thread is contained in this event.

Method Event A method event contains information about the execution of a method, for example which thread executed the method, how long it took and how many CPU clock cycles were consumed since then.

3.3.5 Method Profiling

Since the subject is about measuring program execution, it is obvious to deal with the measurement of method execution in Java. Method measurement can be used to discover and locate performance bottlenecks. Thus the question arises in which part of the program is the most time spent? Other values which could be measured are the total number of method invocations, which thread invoked which method or from a debugging perspective: which method was called from which method?

It was already discussed that profiling adds unpredictable overhead to the program being profiled. The JVMTI counters this with the event-based design. Using a method start and a method end as an event hook, however, eliminates these advantages. Based on the type of the profiled Java program, reacting to every method invocation can lead from low to very high performance impact. Typically there are two types of method profiling [VL00]:

Tracing On every method enter and on every method exit native agent code is executed in order to measure the time spent in the method. The greater the number of method invocations in the Java program, the more the speed of the Java program will slow down.

Sampling The profiler dumps periodically the stack traces of every thread and analyzes them in order to assign a cost unit to the stack trace. The cost unit could be used, for instance, to find the slowest part of the code. Sampling adds a lot less overhead to the profiled application compared to *Tracing*. It is, however, more likely that small methods are missed completely.

Tracing has the advantage of observing the total number of method invocations. Sometimes the overhead can be ignored, when the performance impact does not make the profiling impossible. Since tracing works with high granularity the relative method execution time can be calculated and displayed in percent. When nearly the entire time is used to execute one method, there is no need of knowing how long it took, when the goal is to find the slowest part of the code. To reduce the overhead filters can be introduced. Defined through the user only certain methods are included or some methods are excluded while profiling. From these two types only the tracing is implemented in this profiler.

3.4 Implementation

The implementation section covers the most details in solving problems with respect to the programming part of developing the profiler. These problems arose during the implementation of the software. It will be presented how some of the design choices are implemented in the end.

3.4.1 Communication Protocol

One of the most crucial parts is the implementation of the communication protocol. Since it forms one of the three major components in the architecture it is necessary to have a flexible and extensible implementation. This way it is assured that newly introduced types of data can be easily integrated.

At first the textual format XML was chosen. A single XML document was used to serialize the information of one event. Even though the basic implementation worked out very well, problems emerged due to the continuously growing set of information which had to be preserved in a single document. With this continuously growing set of information the effort, to write the document construction on the one side and to write the document parsing on the other side, grew equally.

For this reason the communication protocol *protobuf* is used now instead. *Protocol Buffers* [Goo11] is a way of encoding data into a serialized form. The advantage is that the code to access the data is generated. This way the integration into different language can be done without further effort. This way Protocol Buffers is used as inter-process communication between the two languages C++ and Java. The structure of a protobuf message is specified in *proto* text files. Once the file is defined, the protobuf compiler generates the corresponding data access classes for the appropriate programming language.

Protobuf describes its records with a structure identified by the keyword *message*. It contains a series of key-value pairs, similar to the record type *struct* in C. For the communication protocol all possible data is described in one message type. Confining all data within one message has the advantage, that all the data can be received from one object. Even if the structure of this message changes, the class type will not change. For that purpose there is only one message type: *AgentMessage*. A message, however, can contain further messages.

An excerpt of the *protobuf* file, used to model the messages, is shown in listing 3.1. Every

```
option java_package = "de.fu.profiler.service";
option java_outer_classname = "AgentMessageProtos";

message AgentMessage {

        required int64 timestamp = 1;
        required int64 systemTime = 2;
        required int32 jvmPid = 3;

        // ...

        optional ThreadEvent threadEvent = 4;
        optional MonitorEvent monitorEvent = 5;
        optional MethodEvent methodEvent = 6;
}
```

Listing 3.1: Describing the communication protocol with the protobuf format

```
message ThreadEvent {

  repeated Thread thread = 1;

  enum EventType {
    STARTED = 0;
    ENDED = 1;
  }

  required EventType eventType = 2;
}
```

Listing 3.2: Defining single messages for communicating the occurrence of an event

message has a timestamp, information about the current system time and the process identifier of the observed JVM. As the JVMTI architecture is event-based the message protocol was designed in a similar way. Further messages, for instance *ThreadEvent* and *MonitorEvent* are encapsulated inside the *AgentMessage* and describe an occurred event. They are optional, which means there has to be no message when constructing the *AgentMessage*. This allows a flexible composition of different message types. Each field in the message has a unique tag number. These tags are used for identification purposes in the message binary format. Besides, tags with values from 1 to 15 take one byte to encode and tags with values from 16 to 2047 take two bytes to encode.

One type of event message, the *ThreadEvent*, is shown in listing 3.2. Since protobuf allows an easy modeling of entities, further messages are used to describe JVMTI related objects, for instance a thread or a monitor. This can also be seen in listing 3.2. The *ThreadEvent* message can have an arbitrary number of *Thread* messages. The definition of a thread message can be seen in listing 3.3.

The object-oriented aspects of C++ and Java allow to access the data with only few lines of code. This way the whole parsing process which would have been required with XML is

```
message Thread {

  required int32 id = 1;
  required string name = 2;
  required int32 priority = 3;

  enum State {
    NEW = 0;
    RUNNABLE = 1;
    BLOCKED = 2;
    WAITING = 3;
    TIMED_WAITING = 4;
    TERMINATED = 5;
  }

  required State state = 4;
  required bool isContextClassLoaderSet = 5;
  required bool isDaemon = 6;
  optional int64 cpuTime = 7;
}
```

Listing 3.3: Modeling JVMTI related entities by additional messages

```
message MemoryEvent {

  required Thread thread = 1;

  enum EventType {
    ALLOCATION = 1;
    FREE = 2;
    }

  required EventType eventType = 2;
  required string className;
  required int32 size;
}
```

Listing 3.4: Extending the current message protocol by a hypothetical new event

avoided. This advantage was significant during the software development. The requirements were not clear from the beginning. This way it was possible extend the present message for new information ad-hoc. For example, if the profiler agent should start to collect information about the heap it could be easily integrated into the current *AgentMessage* by adding the code shown in figure 3.4.

3.4.2 Ordering the Messages

Another problem arose when collecting and displaying information which rely on a causal ordering of events. Since the profiling agent is executed concurrently, the sent messages are not necessarily in the same order as the Java code is executed. This way a series of information could break the expected order. For instance, the graphical user interface displayed, that a certain thread had left *Object.wait* before even entering the method. In

```
extern "C" {
  __inline__ uint64_t rdtsc() {
    uint32_t lo, hi;
    __asm__ __volatile__ (
        "xorl %%eax,%%eax \n        cpuid"
        ::: "%rax", "%rbx", "%rcx", "%rdx");
    __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
    return (uint64_t) hi << 32 | lo;
  }
}
```

Listing 3.5: Generating timestamps by reading the 64-bit register *Time Stamp Counter* on x86 processors

order to establish a mechanism which ensures, that the happened-before relationship is maintained, timestamps were introduced. The first approach taken, was to use the UNIX system function *clock_gettime*.

With respect to the non-functional requirement of portability, however, it is questionable whether there are no superior ways to generate timestamps, since using *clock_gettime* is system-dependent. On x86 architectures the RDTSC instruction is available. It returns a 64-bit time stamp counter. With every clock cycle this counter is increased. The value is read by using Assembly which can be seen in listing 3.5.

Taking multiprocessor systems into account, another problem arises. There are different counters across the available processor cores and they do not synchronize with each other. Since the agent thread of the profiler runs in its own thread it is unlikely different timestamp counters will interfere. The possibilities remains that the agent thread is rescheduled on another core. Therefore no safety guarantees can be made that the timestamps are totally ordered and correct. Practice with the developed profiler has shown, that message reordering did not occur with the applied use cases from section 4.2.

3.4.3 Waiting And Signaling Events

Unfortunately, the JVMTI does not provide for every preferable event a callback function with sufficient information. This problem was faced the first time when implementing the monitor logging mechanism for waiting and signaling events. This includes invocations to *Object.wait*, *Object.notify* and *Object.notifyAll*. The JVMTI offers the event *Monitor Wait* for entering *Object.wait* and *Monitor Waited* for leaving *Object.wait*. For those events a local JNI reference to the Java monitor object is given. Thus it is easy to identify the monitor and gain information, for instance how many threads are waiting on it.

Whereas the signaling part done by the notification does not provide a separate event hook. For this reason the event *MethodExit* was used checking every time, whether the method name is *notify* or *notifyAll*, the class name *Object* and the method itself executing in native code. The downside is that *MethodExit* does not provide a local JNI reference to the Java monitor object. Thus the total state cannot be tracked with every event. But since profiling is about profiling the application in large sense this should satisfy the requirements at first glance. It also shows very clearly the limitations of the JVMTI without further modification of the profiled JVM.

3.4.4 Contextualization by Stack Traces

Another issue arose in the same usage: profiling waiting and signaling events. Since the Java programming language uses monitors also in their Java class library the profiled events might occur more often than expected. Therefore it has to be distinguished when a method is called due to library implementation and when a method is called from the profiled Java application. Even if there is no library code accessing the waiting and signaling functionality, a large application might have different components using this mechanism. Hence the context of a method invocation is important. During profiling for testing purposes the number of calls to *Object.wait* and *Object.notifyAll* differed.

In order to solve this problem, a complete dump of the available stack traces was added to the appropriate *AgentMessage*. When a new thread comes into existence a Java stack is allocated for it by the Java Virtual Machine. The Java stack is composed of multiple stack frames [AGH00, sec. 12.6]. These stack frames help to identify what the thread is currently executing. Analyzing the stack trace is an approach to put the events into context. Answering questions like: where is *Thread-2* trying to acquire the monitor lock? The JVMTI offers the function *GetStackTrace* to be applied on a specified thread. Further functions, for example *GetMethodName* can be applied to receive more details.

The stack trace indicates in which scope the method was invoked. This way it was revealed that calls to *Object.notifyAll* were made within *java.util.ResourceBundle* in the method *endLoading*. This is simply due to the reason *System.format* uses *ResourceBundle*.

3.4.5 Contention and Deadlock Detection

The simplest case of deadlock is *deadly embrace*. A *deadly embrace* evolves when a thread holds a lock forever and other threads try to acquire this lock. The consequence of this is that the other threads will block forever waiting [GBB$^+$06, p. 205].

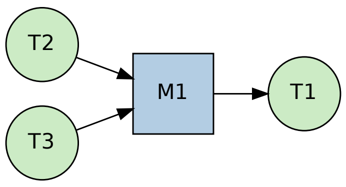
In order to detect a deadlock several approaches can be taken. A Java thread dump provides already information whether a Java deadlock has occurred. A Java thread dump can be received by several ways for instance by tools like *jps* and *jstack* or by sending an operating system dependent signal to the JVM. The JVMTI does not provide a function to obtain the current thread dump. Detecting deadlocks, however, can be realized by using the given information when a thread contends with another thread for entering a monitor. The logic for detecting a deadlock is implemented in the Java front end.

A resource-allocation graph can be used to detect the deadlocks. A resource-allocation graph consists of processes P and resources R [SG98, p. 210]. The resources can have different numbers of instances. From the Java perspective P is a thread and the resource R is a monitor lock. A monitor lock can only be acquired or released, having only one instance makes it a binary resource. A directed edge from a monitor to a thread means the thread owns the lock of this monitor. A directed edge from a thread to a monitor means the thread is waiting to acquire the lock of this monitor. The following lemmata state when a deadlock has occurred.

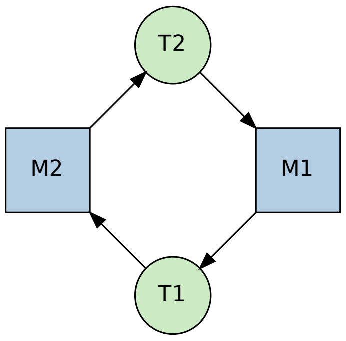
Lemma 1. *If the graph contains no cycle, then the state does not contain a deadlock.*

Lemma 2. *If the graph contains a cycle and every resource is a binary resource, then the state contains a deadlock.*

Two examples of these graphs are shown in figure 3.2.



(a) Thread t_2 and t_3 are in contention with Thread t_1, since thread t_1 owns the monitor lock



(b) Thread t_1 and t_2 are waiting for each other to release the monitor lock – a deadlock occurred

Figure 3.2: Illustrating different resource-allocation graphs

The *JUNG* library [OFN11] is used to implement a visualization of the monitor contention by constructing and drawing the appropriate resource-allocation graph. This is not necessarily a gain in information, but should give an interesting view of the monitor lock distribution in general. The JVMTI offers two event hooks which help to track the locking.

Monitor Contended Enter Sent when a thread is attempting to enter a Java programming language monitor already acquired by another thread.

Monitor Contended Entered Sent when a thread enters a Java programming language monitor after waiting for it to be released by another thread.

This means, there will be no tracking of locks which are directly acquired without contention and there will be no tracking of locks which are released while no other threads waits on them to be released. For the deadlock detection this is no problem, since only contention can evolve to a deadlock. When the event *Monitor Contended Enter* happens it means a graph with 3 nodes and two edges can be constructed at once. When the event *Monitor Contended Entered* happens it means one edge will disappear and one edge will change the direction. Finally the resource-allocation graph is limited to deadlock detection. Hence, not every single allocation step can be tracked.

3.4.6 Method Profiling

In order to implement one of the discussed methods of method profiling the *AgentMessage* was simply extended by a *MethodEvent*. Within this work only the type tracing was implemented. The procedure can be described in four steps:

1. For every available thread a stack is allocated.
2. When a method is entered, the current system time is retrieved and the result is pushed on the appropriate stack.
3. When a method is left, the timestamp is popped and the difference is calculated. A *MethodEvent* message is generated and sent to the Java front end.
4. Finally, the results are cumulated in the Java front end.

This way no method is left out and the total call order is maintained. The emerging overhead, due to the fact of sending a message each time a method is left, is huge. It was already proposed [VL00] to make heavy use of filters in this case. At a minimum it is reasonable to filter every method of a class from the packages *com.sun*, *java*, *javax*, *sun* or *sunw*. They include the whole Java library code. This way the profiler focuses on the code which can be actually influenced by the developer.

It is, however, important to leave the choice whether these classes should be filtered to the user. With an active filter on the *java* package, the *java.util.concurrent* package will not be profiled neither. In this case, the profiler cannot keep track of the actions invoked by other synchronization structures, for instance the number of *Semaphore.acquire* invocations.

Chapter 4

Usage

In order to test the profiler this chapter will provide a demonstrative overview of the profilers capabilities. First, in section 4.1 the developed profiler will be presented and how it is used. This is followed by a set of use cases. The use cases will demonstrate the implemented functionality of the profiler and what cognition can be gained. Seven use cases examine different aspects of the profiler. This is followed by an evaluation in section 4.3 which will reflect on the results, where potential weak points are and what solutions could be applied to counter these weak points. In section 4.4 the caused alterations in program execution by using the profiling agent will be measured and evaluated as well.

4.1 Manual

The system on which the target's application will be profiled has to be prepared. The profiling agent is compiled as a shared library: *libjcp.so.* This file has to be inserted into the library path. Whether Java can load the profiler agent, can be tested afterwards by invoking the command shown in listing 4.1.

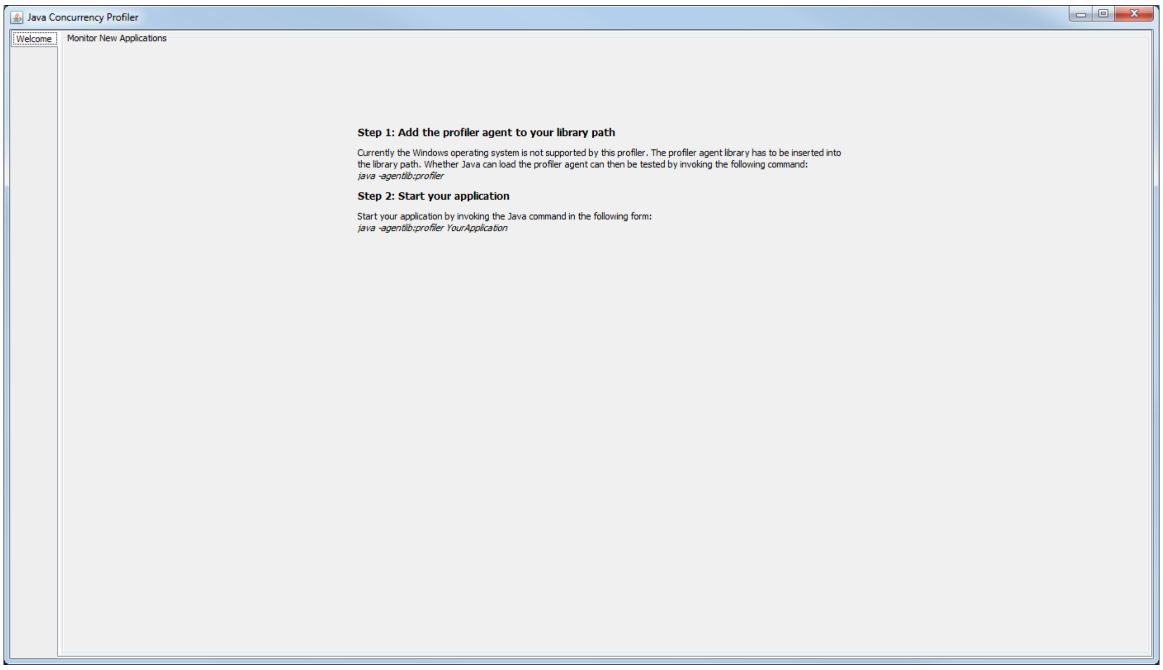
```
$ java -libagent:jcp
```

Listing 4.1: Testing whether the system can find the profiler agent

If no error is displayed, telling that the library agent could not be found, everything should work accordingly. Next, the Java front end has to be started by executing the *java-concurrency-profiler.jar* on the machine which should be used for receiving the results. If no port number is passed to the front end it will listen per default on port 49125. If the front end should listen on another port it has to be executed with passing the port number as command-line argument. This can be seen in listing 4.2.

```
$ java -jar java-concurrency-profiler.jar [port]
```

Listing 4.2: Starting the Java front end with passing an optional port parameter



Step 1: Add the profiler agent to your library path

Currently the Windows operating system is not supported by this profiler. The profiler agent library has to be inserted into the library path. Whether Java can load the profiler agent can then be tested by invoking the following command:
java -agentlib:profiler

Step 2: Start your application

Start your application by invoking the Java command in the following form:
java -agentlib:profiler YourApplication

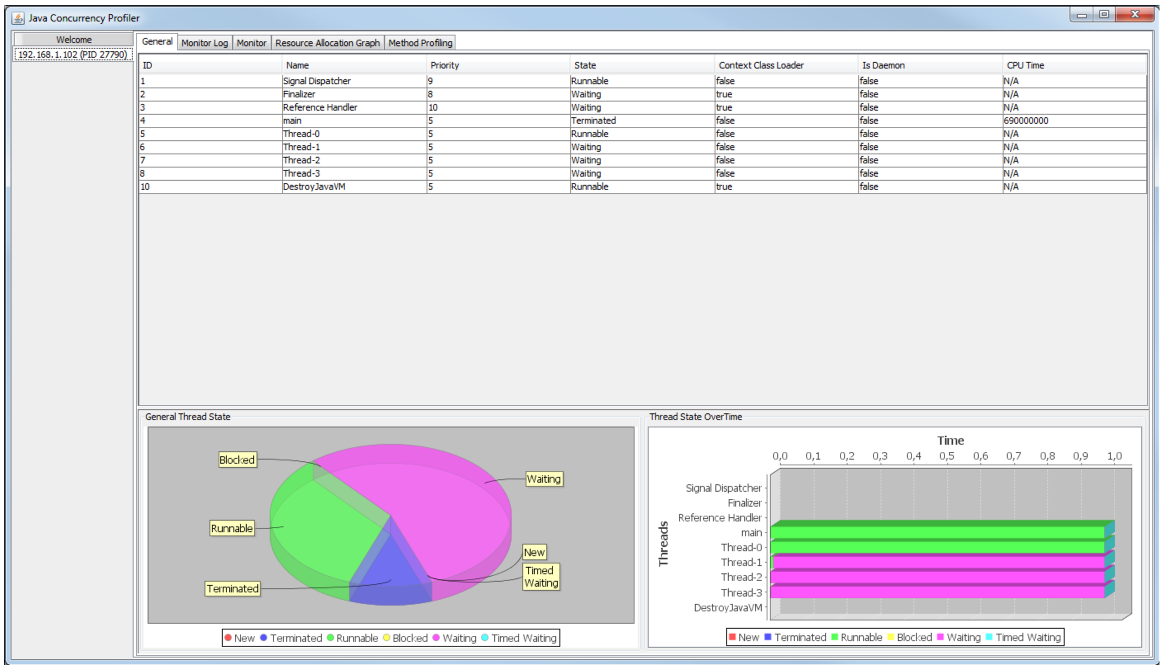
Figure 4.1: Displaying the front ends welcome screen and waiting for incoming results

Finally, in order to start the profiling itself the Java application has to be started in a similar way described above. If, however, another machine is used for receiving the results, additional parameters are required: the host and the port of the target machine. An example is shown in listing 4.3. Otherwise it will try to connect to the localhost on port 49125.

```
$ java -agentlib:jcp=[host],[port] HelloWorld
```

Listing 4.3: Starting the profiling process with two optional parameters specifying the Java front ends network address

When the Java front end is started a window frame should appear. Only the the tab *Welcome* should be seen, as in figure 4.1, with similar instructions to the described above. A new tab will appear, when the first data is received from the profiling agent. The panel attached to the tab has different views which can be selected by additional tabs. The initial view will be the general overview, which is shown in figure 4.2. The general overview consists of a table and two diagrams. The table contains information about the threads which were active at a time during profiling. The information about a thread consist of its identifier, its name, its priority, its current state, whether it has an associated context class loader, whether it is a daemon thread and the estimated CPU time utilized by the thread in nanoseconds. The pie chart diagram shows the current threads' state distribution. The bar chart shows for each thread how much time was spent in which state in percent.



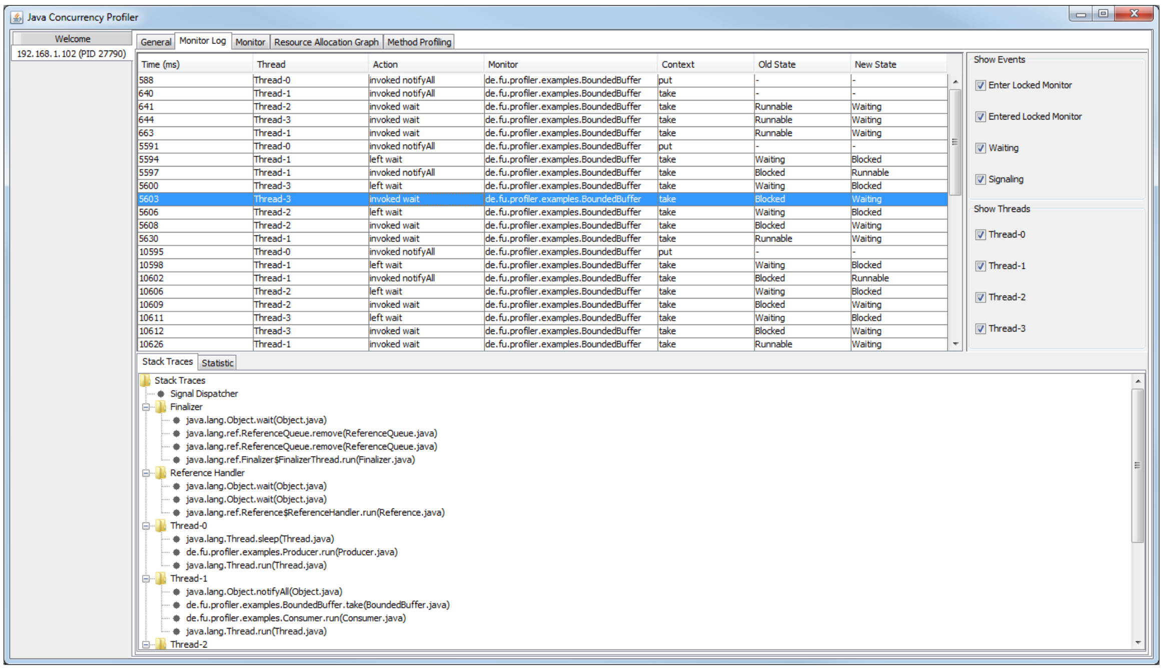
ID	Name	Priority	State	Context Class Loader	Is Daemon	CPU Time
1	Signal Dispatcher	9	Runnable	false	false	N/A
2	Finalizer	8	Waiting	true	false	N/A
3	Reference Handler	10	Waiting	true	false	N/A
4	main	5	Terminated	false	false	690000000
5	Thread-0	5	Runnable	false	false	N/A
6	Thread-1	5	Waiting	false	false	N/A
7	Thread-2	5	Waiting	false	false	N/A
8	Thread-3	5	Waiting	false	false	N/A
10	DestroyJavaVM	5	Runnable	true	false	N/A

Figure 4.2: Serving as an entry point the general view shows the involved threads and their states

The monitor log view, which can be seen in figure 4.3, shows the existing log entries which contain information about actions related to the present Java monitors. The log entries are displayed in a table. An entry is represented by a row in the table. Every entry has information about the time, since the application has started, the thread which performed the action, the type of action, the class type of the involved monitor, the method in which the action was performed and if available, the previous and the new state of the executing thread.

On the right side of this view is a list of checkboxes which allows to filter the content of the table. One the one hand it is possible to filter the table by certain event types. For instance, when deselecting the *Waiting* checkbox, all log entries with an action related to *Object.wait* are hidden. On the other hand the table can be filtered by threads, too. This list of checkboxes is created dynamically. Obviously only present threads in the monitor log can be used for filtering.

When selecting a log entry the graphical user interface will show the stack trace of every thread which was present when the event occurred at the bottom of this view. The stack trace is displayed in a tree form. The bottom of this view has another tab containing statistical information about how often a certain event has occurred per thread. For instance, how often has a particular thread contended when trying to acquire a monitor lock. The statistical panel is independent of the selected log entry.



Time (ms)	Thread	Action	Monitor	Context	Old State	New State
588	Thread-0	invoked notifyAll	de.fu.profiler.examples.BoundedBuffer	put	-	-
640	Thread-1	invoked notifyAll	de.fu.profiler.examples.BoundedBuffer	take	-	-
641	Thread-2	invoked wait	de.fu.profiler.examples.BoundedBuffer	take	Runnable	Waiting
644	Thread-3	invoked wait	de.fu.profiler.examples.BoundedBuffer	take	Runnable	Waiting
663	Thread-1	invoked wait	de.fu.profiler.examples.BoundedBuffer	take	Runnable	Waiting
5591	Thread-0	invoked notifyAll	de.fu.profiler.examples.BoundedBuffer	put	-	-
5594	Thread-1	left wait	de.fu.profiler.examples.BoundedBuffer	take	Waiting	Blocked
5597	Thread-1	invoked notifyAll	de.fu.profiler.examples.BoundedBuffer	take	Blocked	Runnable
5600	Thread-3	left wait	de.fu.profiler.examples.BoundedBuffer	take	Waiting	Blocked
5603	Thread-3	invoked wait	de.fu.profiler.examples.BoundedBuffer	take	Blocked	Waiting
5606	Thread-2	left wait	de.fu.profiler.examples.BoundedBuffer	take	Waiting	Blocked
5608	Thread-2	invoked wait	de.fu.profiler.examples.BoundedBuffer	take	Blocked	Waiting
5630	Thread-1	invoked wait	de.fu.profiler.examples.BoundedBuffer	take	Runnable	Waiting
10595	Thread-0	invoked notifyAll	de.fu.profiler.examples.BoundedBuffer	put	-	-
10598	Thread-1	left wait	de.fu.profiler.examples.BoundedBuffer	take	Waiting	Blocked
10602	Thread-1	invoked notifyAll	de.fu.profiler.examples.BoundedBuffer	take	Blocked	Runnable
10606	Thread-2	left wait	de.fu.profiler.examples.BoundedBuffer	take	Waiting	Blocked
10609	Thread-2	invoked wait	de.fu.profiler.examples.BoundedBuffer	take	Blocked	Waiting
10611	Thread-3	left wait	de.fu.profiler.examples.BoundedBuffer	take	Waiting	Blocked
10612	Thread-3	invoked wait	de.fu.profiler.examples.BoundedBuffer	take	Blocked	Waiting
10626	Thread-1	invoked wait	de.fu.profiler.examples.BoundedBuffer	take	Runnable	Waiting

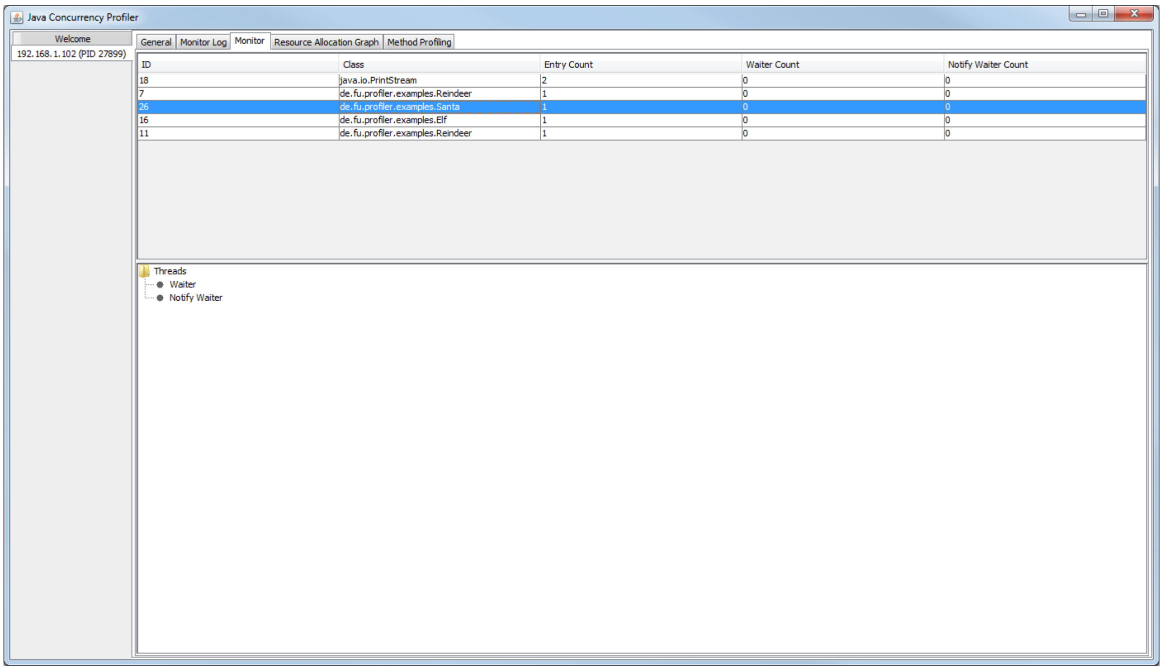
```
Stack Traces
  Signal Dispatcher
  Finalizer
    java.lang.Object.wait(Object.java)
    java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java)
    java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java)
    java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java)
  Reference Handler
    java.lang.Object.wait(Object.java)
    java.lang.Object.wait(Object.java)
    java.lang.ref.Reference$ReferenceHandler.run(Reference.java)
  Thread-0
    java.lang.Thread.sleep(Thread.java)
    de.fu.profiler.examples.Producer.run(Producer.java)
    java.lang.Thread.run(Thread.java)
  Thread-1
    java.lang.Object.notifyAll(Object.java)
    de.fu.profiler.examples.BoundedBuffer.take(BoundedBuffer.java)
    de.fu.profiler.examples.Consumer.run(Consumer.java)
    java.lang.Thread.run(Thread.java)
  Thread-2
```

Figure 4.3: Tracing the actions performed on the profiled monitors and visualizing a complete stack trace for each log entry

In figure 4.4 the monitor view is shown. The monitor view lists all the used Java monitors in a table. The table contains information about the class type of the monitor and its current state. The state is described by three information: the number of times the owning thread has entered the monitor, the number of threads waiting to enter the monitor and the number of threads waiting to be notified by other threads. When a row is selected in the table further information are displayed at the bottom of this view. In a tree form both types of waiting threads are displayed. The concept of contextualizing by using stack traces is given here, too. If there are any threads waiting, it is possible to get their current stack trace by expanding the corresponding thread node.

A screenshot of the resource-allocation graph view can be seen in figure 4.5. All present threads are represented on the panel as red circles, even though they are not related to the monitors at all. The monitors are represented as gray squares. For identification purposes the threads are labeled with their names and their identifier and the monitors are labeled with their class type and their identifier. When a circle in the graph is detected by the front end the tab title will turn its color to red and change the label from *Resource-Allocation Graph* to *Deadlocked*.

Finally, the method profiling view is shown in figure 4.6. The information is displayed in a table as well. Each row of the table represents a method. The information provided by the table consists of the relative time spent in the method expressed in percent, the average



Java Concurrency Profiler

Welcome

192.168.1.102 (PID 27899)

General | Monitor Log | Monitor | Resource Allocation Graph | Method Profiling

ID	Class	Entry Count	Waiter Count	Notify Waiter Count
18	java.io.PrintStream	2	0	0
7	de.fu.profiler.examples.Reindeer	1	0	0
26	de.fu.profiler.examples.Santa	1	0	0
16	de.fu.profiler.examples.Elf	1	0	0
11	de.fu.profiler.examples.Reindeer	1	0	0

Threads
- Waiter
- Notify Waiter

Figure 4.4: Presenting the profiled monitors and information about their usage

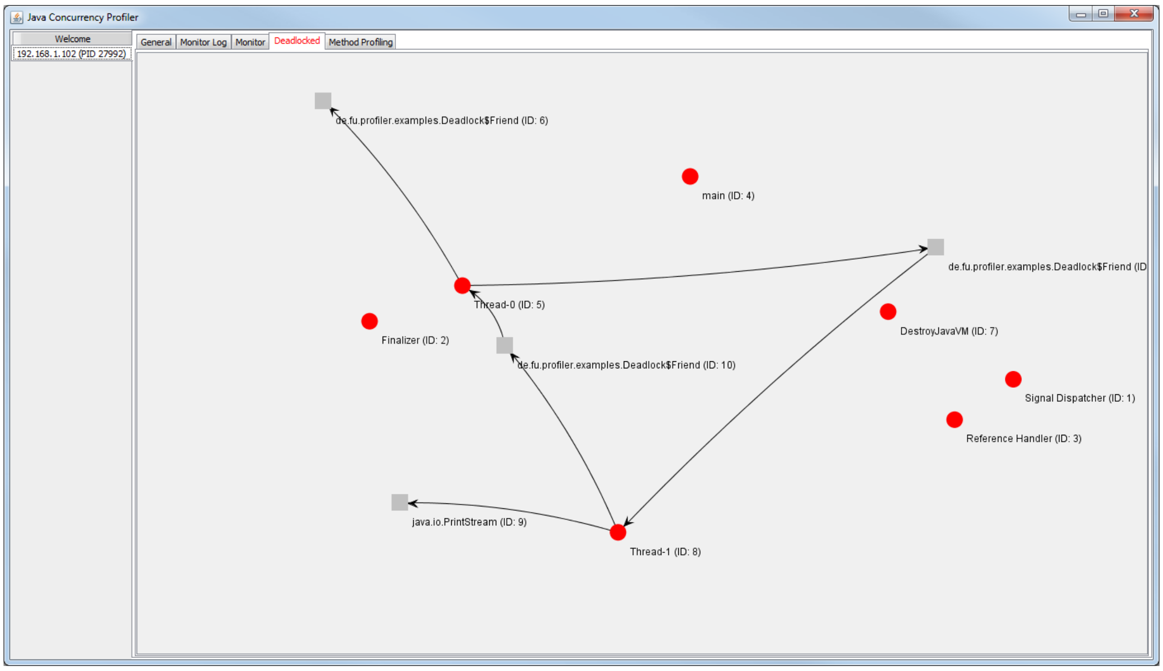
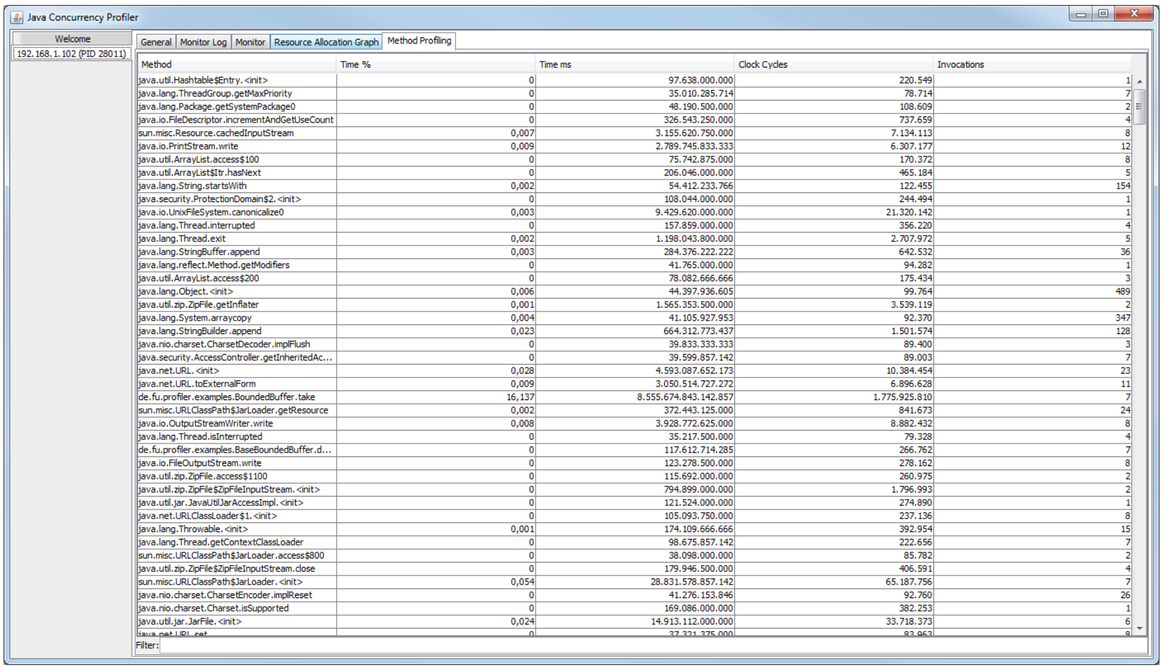

Java Concurrency Profiler
Welcome
192.168.1.102 (PID 27992)
General
Monitor Log
Monitor
Deadlocked
Method Profiling
de.fu.profiler.examples.Deadlock$Friend (ID: 6)
main (ID: 4)
de.fu.profiler.examples.Deadlock$Friend (ID
Thread-0 (ID: 5)
DestroyJavaVM (ID: 7)
Finalizer (ID: 2)
de.fu.profiler.examples.Deadlock$Friend (ID: 10)
Signal Dispatcher (ID: 1)
Reference Handler (ID: 3)
java.io.PrintStream (ID: 9)
Thread-1 (ID: 8)

Figure 4.5: Constructing dynamically a resource-allocation and checking for deadlocks


Java Concurrency Profiler
Welcome
192.168.1.102 (PID 28011)
General
Monitor Log
Monitor
Resource Allocation Graph
Method Profiling
Filter:

Method	Time %	Time ms	Clock Cycles	Invocations
java.util.Hashtable$Entry.<init>	0	97.638.000.000	220.549	1
java.lang.ThreadGroup.getMaxPriority	0	35.010.285.714	78.714	7
java.lang.Package.getSystemPackage0	0	48.190.500.000	108.609	2
java.io.FileDescriptor.incrementAndGetUseCount	0	326.543.250.000	737.659	4
sun.misc.Resource.cachedInputStream	0,007	3.155.620.750.000	7.134.113	8
java.io.PrintStream.write	0,009	2.789.745.833.333	6.307.177	12
java.util.ArrayList.access$100	0	75.742.875.000	170.372	8
java.util.ArrayList$Itr.hasNext	0	206.046.000.000	465.184	5
java.lang.String.startsWith	0,002	54.412.233.766	122.455	154
java.security.ProtectionDomain$2.<init>	0	108.044.000.000	244.494	1
java.io.UnixFileSystem.canonicalize0	0,003	9.429.620.000.000	21.320.142	1
java.lang.Thread.interrupted	0	157.859.000.000	356.220	4
java.lang.Thread.exit	0,002	1.198.043.800.000	2.707.972	5
java.lang.StringBuffer.append	0,003	284.376.222.222	642.532	36
java.lang.reflect.Method.getModifiers	0	41.765.000.000	94.282	1
java.util.ArrayList.access$200	0	78.082.666.666	175.434	3
java.lang.Object.<init>	0,006	44.397.936.605	99.764	489
java.util.zip.ZipFile.getInflater	0,001	1.565.353.500.000	3.539.119	2
java.lang.System.arraycopy	0,004	41.105.927.953	92.370	347
java.lang.StringBuilder.append	0,023	664.312.773.437	1.501.574	128
java.nio.charset.CharsetDecoder.implFlush	0	39.833.333.333	89.400	3
java.security.AccessController.getInheritedAc...	0	39.599.857.142	89.003	7
java.net.URL.<init>	0,028	4.593.087.652.173	10.384.454	23
java.net.URL.toExternalForm	0,009	3.050.514.727.272	6.896.628	11
de.fu.profiler.examples.BoundedBuffer.take	16,137	8.555.674.843.142.857	1.775.925.810	7
sun.misc.URLClassPath$JarLoader.getResource	0,002	372.443.125.000	841.673	24
java.io.OutputStreamWriter.write	0,008	3.928.772.625.000	8.882.432	8
java.lang.Thread.isInterrupted	0	35.217.500.000	79.328	4
de.fu.profiler.examples.BaseBoundedBuffer.d...	0	117.612.714.285	266.762	7
java.io.FileOutputStream.write	0	123.278.500.000	278.162	8
java.util.zip.ZipFile.access$1100	0	115.692.000.000	260.975	2
java.util.zip.ZipFile$ZipFileInputStream.<init>	0	794.899.000.000	1.796.993	2
java.util.jar.JavaUtilJarAccessImpl.<init>	0	121.524.000.000	274.890	1
java.net.URLClassLoader$1.<init>	0	105.093.750.000	237.136	8
java.lang.Throwable.<init>	0,001	174.109.666.666	392.954	15
java.lang.Thread.getContextClassLoader	0	98.675.857.142	222.656	7
sun.misc.URLClassPath$JarLoader.access$800	0	38.098.000.000	85.782	2
java.util.zip.ZipFile$ZipFileInputStream.close	0	179.946.500.000	406.591	4
sun.misc.URLClassPath$JarLoader.<init>	0,054	28.831.578.857.142	65.187.756	7
java.nio.charset.CharsetEncoder.implReset	0	41.276.153.846	92.760	26
java.nio.charset.Charset.isSupported	0	169.086.000.000	382.253	1
java.util.jar.JarFile.<init>	0,024	14.913.112.000.000	33.718.373	6
java.net.URL.set	0	37.321.375.000	83.963	8

Figure 4.6: Presenting the method profiling information in a tabular form which can be filtered by methods

time spent expressed in milliseconds, the average clock cycles consumed when executing the method and the total number of invocations. The table can be sorted by clicking on the column headers. The concept of filtering is applied here, too. At the bottom of the view a name can be entered into a text field. Only methods containing this name will be shown. It is suggested to start with the prefix of the method, hence the package names.

4.2 Use Cases

The following use cases will demonstrate the capabilities of the profiler. The results of the use cases are going evaluated afterwards. There are use cases to demonstrate how the profiler can strengthen the conceptual understanding of the underlying concurrency mechanisms. There are also use cases which are related more to practical, respectively real examples.

4.2.1 Extending a Thread-Safe Class

First of all the example which was addressed in the problem statement in section 1.2 will be revisited. In the listing 4.4 the code is shown again.

```
public class EnhancedSynchronizedList<T> {
  public List<T> list =
    Collections.synchronizedList(new ArrayList<T>());

  public synchronized boolean putIfAbsent(T x) {
    boolean absent = !list.contains(x);
    if (absent) {
      list.add(x);
    }
    return absent;
  }
}
```

Listing 4.4: Revisiting the problem of extending a thread-safe class

Problem

Extending a thread-safe class does not imply the thread-safety is retained. The method *Collections.synchronizedList* returns a thread-safe list. The list is extended by the method *pufIfAbsent*, which only adds an element, if the element is not present yet. The question was: is the profiler able to reveal information about the thread-safety?

Approach

Since thread-safety is about encapsulating the needed synchronization to a sufficient degree [GBB^{+}06, p. 18], the profiler will be used to trace how monitor of the list is accessed.

Profiling

Class	Context
de.fu.profiler.examples.EnhancedSynchronizedList	EnhancedSynchronizedList.putIfAbsent
de.fu.profiler.examples.EnhancedSynchronizedList$1	Runnable.run
java.util.Collections$SynchronizedCollection	Collections$SynchronizedCollection.add

Table 4.1: Listing the profiled monitors and in which context each monitor was accessed in the *Extending a Thread-Safe Class* example

Evaluation

Apparently there are more monitors than might have been expected in the beginning. The class *Collections$SynchronizedCollection* was expected to be used as a monitor and its method *add* is within scope of this monitor. *EnhancedSynchronizedList$1* is the anonymous inner-class implementation of *java.lang.Runnable* which is used to execute the threads accessing the list. The monitor of class type *EnhancedSynchronizedList*, however, should not be present at all. The method *putIfAbsent* is synchronized on the *EnhancedSynchronizedList* object instead on the *Collections$SynchronizedCollection* object.

```java
public class EnhancedSynchronizedList<T> {
  public List<T> list =
    Collections.synchronizedList(new ArrayList<T>());

  public boolean putIfAbsent(T x) {

    synchronized (list) {
      boolean absent = !list.contains(x);
      if (absent) {
        list.add(x);
      }
      return absent;
    }
  }
}
```

Listing 4.5: Extending a thread-safe class by using *client-side locking*

The problem is, that the *synchronized* modifier of the method *putIfAbsent* implies, that the class defining the method *putIfAbsent* will be used as monitor. The class defining the method is *EnhancedSynchronizedList*. A *synchronized* block using *list* as the lock has to be used in order to reestablish thread-safety. This approach is called *client-side locking* or *external locking* [GBB+06, p. 73]. The solution can be seen in listing 4.5.

4.2.2 Overhead of Santa Claus Problem

The Santa Claus Problem was original defined by John A. Trono [Tro94] and is a typical exercise for undergraduate students when introduced to concurrency.

Problem

There are three parties: Santa Claus, nine reindeer and some elves. Santa Claus is sleeping and can only be awaken by all nine reindeer or by three elves. When three elves are at Santa, any other elf has to wait. Each member of a party is represented by a *Java Thread* which executes its own task with dependencies to the described conditions.

In *How to Solve the Santa Claus Problem* [Ba98] Modechai Ben-Ari states the following problem to the Java solution. Java can declare an arbitrary number of condition variables, however, the signaling mechanism does not define which thread is notified, thus no FIFO can be guaranteed. The absent of FIFO can lead to race conditions.

In the opposite to Ada 95 which specifies *immediate resumption* in Java the condition has to be re-checked if the condition still holds. Further Modechai Ben-Ari states the basic problem in the Java implementation is the high-overhead which is due to the method *notifyAll*. Every thread is reactivated and every thread has to re-check the condition.

Since reactivating a thread leads to context switches and context switches are expensive [GBB+06, p. 230]. Since there are so many threads involved it would be interesting to profile the mentioned overhead and see if it can be seen by the profiler. The idea is to see how many threads are awakened and return immediately back to sleep. In the worst case *Object.notifyAll* results in $\mathcal{O}(n^2)$ wakeup where in the contrast $\mathcal{O}(n)$ would be sufficient to do the task [GBB+06, p. 303].

Approach

For this problem useful information can be gained through the monitor log view. Since this view presents the invocations of the methods *Object.wait* and *Object.notify* and records their total occurrence, it should reveal the overhead. Further also statistical numbers collected about method invocations could give proof for the produced overhead.

Profiling

The results are shown in the table 4.2 and 4.3.

Thread	#wait	#notify	#notifyAll	#contended	#entered
Santa Clause	13	134	0	23	23
Elf-0	10	0	0	14	14
Elf-1	8	2	0	16	15
Elf-2	8	0	0	8	8
Elf-3	8	0	0	17	17
Elf-4	10	3	0	17	17
Elf-5	8	1	0	9	8
Elf-6	8	0	0	10	10
Elf-7	10	5	0	15	15
Elf-8	8	1	0	14	14
Elf-9	7	0	0	9	9
Reindeer-0	6	0	0	6	6
Reindeer-1	6	0	0	11	11
Reindeer-2	6	1	0	12	11
Reindeer-3	6	1	0	10	11
Reindeer-4	6	0	0	14	13
Reindeer-5	6	0	0	12	12
Reindeer-6	6	0	0	14	15
Reindeer-7	6	0	0	14	14
Reindeer-8	6	1	0	11	11

Table 4.2: Number of times a certain event has occurred in the *Santa Claus Problem*

Time (ms)	Thread	Action	Context	Old State	New State
3539	Elf-0	invoked wait	Elf.run	Runnable	Waiting
3544	Elf-7	invoked wait	Elf.run	Runnable	Waiting
3551	Elf-6	invoked wait	Elf.run	Runnable	Waiting
3553	Santa Clause	invoked notify	Elf.showOut	-	-
3555	Elf-6	left wait	Elf.run	Waiting	Blocked
3560	Santa Clause	invoked notify	Elf.showOut	-	-
3566	Elf-0	left wait	Elf.run	Waiting	Blocked
3633	Santa Clause	invoked notify	Elf.showOut	-	-
3636	Elf-7	left wait	Elf.run	Waiting	Blocked
3639	Santa Clause	invoked wait	Santa.run	Runnable	Waiting
3684	Elf-2	invoked notify	Santa.ask	-	-
3687	Elf-3	invoked wait	Elf.run	Runnable	Waiting
3689	Elf-2	invoked wait	Elf.run	Runnable	Waiting
3698	Elf-4	invoked wait	Elf.run	Runnable	Waiting
3700	Elf-5	invoked wait	Elf.run	Runnable	Waiting
3702	Elf-1	invoked wait	Elf.run	Runnable	Waiting
3707	Elf-8	invoked wait	Elf.run	Runnable	Waiting
3708	Santa Clause	left wait	Santa.run	Waiting	Blocked
3721	Santa Clause	invoked notify	Elf.showIn	Blocked	Runnable

Table 4.3: Logged access to the used monitors in the *Santa Claus Problem*

Evaluation

In table 4.3 only a fraction of the profiled results is shown, since these are only 19 entries from a total number of 442 entries. The profiler uncovered the thread state changes during the thread coordination. Every elf goes to sleep and is woken-up afterwards by the Santa Claus thread. Taking the number of elves and reindeers into account there are a lot of context switches due to thread coordination. The most invocations are done by the Santa Claus thread, since this thread is the thread coordinator. This is also shown through the results of figure 4.2. Santa Claus has with 13 calls made to *Object.wait* and a total number of 134 calls made to *Object.notify* the most invocations. Overall about 29, 4% of the time was spent in *Object.wait*. Without contention due to monitor acquisition this might be negligible, but the results show every entity taking part, contends from 6 times to 23 times, waiting for the monitor lock to be released.

4.2.3 Fairness of Bounded Buffer

Another typical task in multithreading is the bounded buffer problem. A bounded buffer is a multislot communication buffer which is shared between threads. Producer threads deposit objects in the buffer and consumer threads fetch them. The buffer has a queue containing those objects which have not been fetched yet [And99, p. 161]. A simple example of using this pattern could be defined as following:

1. data is shared through a bounded buffer, capable of holding only a limited number of items
2. the producer puts a series of text messages into the bounded buffer
3. the consumer thread retrieves the messages and prints them

This pattern is very typically for multi-processor programs as both processes share the same data. The *producer* and *consumer* threads have to communicate. There are several ways to implement this program. For this use case condition queues will be used the same way as they were introduced in section 2.3.

Problem

Based on the capacity of the bounded buffer and the number of producer and consumer threads the threads will have to contend for accessing the bounded buffer. When only consumer threads have to contend with each other for accessing the bounded buffer, it would be interesting to see, whether the profiler can reveal information about the fairness of condition queues in Java. Will every consumer have the possibility to take its turn?

Approach

In order to track the events stated above, the monitor log will be appropriate as invocations to *Object.wait* and *Object.notifyAll* and their related events have to be traced.

Profiling

The bounded buffer is profiled with the following configuration:

- bounded buffer capacity: 1
- number of producer: 1
- number of consumer: 4
- number of objects produced: 4

The results are shown in table 4.4, 4.5 and 4.6.

Time (ms)	Thread	Action	Context	Old State	New State
570	Thread-0	invoked notifyAll	BoundedBuffer.put	-	-
645	Thread-2	contended with Thread-1	BoundedBuffer.take	-	-
647	Thread-3	contended with Thread-1	BoundedBuffer.take	Runnable	Blocked
648	Thread-4	contended with Thread-1	BoundedBuffer.take	Runnable	Blocked
649	Thread-1	invoked notifyAll	BoundedBuffer.take	-	-
650	Thread-4	entered after contention	BoundedBuffer.take	Blocked	Runnable
653	Thread-4	invoked wait	BoundedBuffer.take	Runnable	Waiting
655	Thread-2	entered after contention	BoundedBuffer.take	Blocked	Runnable
657	Thread-2	invoked wait	BoundedBuffer.take	Runnable	Waiting
668	Thread-3	entered after contention	BoundedBuffer.take	Blocked	Runnable
669	Thread-3	invoked wait	BoundedBuffer.take	Runnable	Waiting
677	Thread-1	invoked wait	BoundedBuffer.take	Runnable	Waiting
5571	Thread-0	invoked notifyAll	BoundedBuffer.put	-	-
5572	Thread-1	left wait	BoundedBuffer.take	Waiting	Blocked
5574	Thread-1	invoked notifyAll	BoundedBuffer.take	Blocked	Runnable
5575	Thread-3	left wait	BoundedBuffer.take	Waiting	Blocked
5576	Thread-3	invoked wait	BoundedBuffer.take	Blocked	Waiting
5577	Thread-2	left wait	BoundedBuffer.take	Waiting	Blocked
5578	Thread-2	invoked wait	BoundedBuffer.take	Blocked	Waiting
5580	Thread-4	left wait	BoundedBuffer.take	Waiting	Blocked

Table 4.4: Logged access to the used monitors in the *Bounded Buffer* example

Thread	Role	#wait	#notify	#notifyAll	#contended	#entered
Thread-0	Producer	6	0	8	0	0
Thread-1	Consumer	12	0	5	0	0
Thread-2	Consumer	15	0	1	1	1
Thread-3	Consumer	15	0	1	1	1
Thread-4	Consumer	18	0	1	1	1

Table 4.5: Number of times a certain event has occurred in the *Bounded Buffer* example

Time (ms)	Thread	Action	Context	Old State	New State
653	Thread-4	invoked wait	BoundedBuffer.take	Runnable	Waiting
5580	Thread-4	left wait	BoundedBuffer.take	Waiting	Blocked
5583	Thread-4	invoked wait	BoundedBuffer.take	Blocked	Waiting
10585	Thread-4	left wait	BoundedBuffer.take	Waiting	Blocked
10588	Thread-4	invoked wait	BoundedBuffer.take	Blocked	Waiting
15601	Thread-4	left wait	BoundedBuffer.take	Waiting	Blocked
15602	Thread-4	invoked wait	BoundedBuffer.take	Blocked	Waiting
20598	Thread-4	left wait	BoundedBuffer.take	Waiting	Blocked
20602	Thread-4	invoked wait	BoundedBuffer.take	Blocked	Waiting
20631	Thread-4	left wait	BoundedBuffer.take	Waiting	Blocked
20632	Thread-4	invoked wait	BoundedBuffer.take	Blocked	Waiting
20635	Thread-4	left wait	BoundedBuffer.take	Waiting	Blocked

Table 4.6: Filtered results of the monitor log for displaying the actions of a single thread

Evaluation

It can be easily figured out which thread fulfilled which kind of role by looking at the results shown in table 4.4. *Thread-0* is the producer, since the thread accesses the *BoundedBuffer.put* method. The rest of the threads are consumer. At the beginning all threads, except *Thread-0* and *Thread-1* become blocked, because *Thread-1* is the first consumer to acquire the *BoundedBuffer* monitor lock. It is evident that only 1 consumer thread is able to succeed. Even though the other threads are awakened, only *Thread-1* is able always to pass the condition. The other threads remain between *Waiting* and *Blocking* state.

For the collected statistics it is obvious that *Thread-0* has the least contention, since there are more consumer than producer. Further *Thread-1* has the least calls to *Object.wait*, since it succeeds all the time and has most of the calls to *Object.notifyAll*. In this case having most of the calls to *Object.notifyAll* correlates to being the thread which is able to acquire the monitor lock and execute the state dependent code. The incurring overhead can be exemplary seen in figure 4.6 for one thread. Within a time period of 19982 ms *Thread-4* does nothing except trying to acquire the monitor lock and go back into the *Waiting* state again. Dependent of the implementation of thread blocking this will result in equal many context switches.

4.2.4 Bottleneck in Concurrent Merge Sort

Merge sort is a divide and conquer algorithm. Since independent parts of a program are not critical they can be executed concurrently without further synchronization mechanisms [And99, p. 45]. In this implementation of concurrent merge sort a fixed number of threads is used to distribute the task.

This can be realized by maintaining a list of threads. Whenever there are two free spots in the list the recursion is executed by two newly created threads. If there is only one free spot in the thread pool the recursion is executed by one newly created thread and the second recursion is executed by the thread controller. If there is no spot in the thread pool at all both recursions are executed by the thread controller. In order to guarantee independent recursive calls, every time a new thread is created the thread controller waits for its termination.

Problem

A concurrent program which relies on little synchronization is expected to execute very fast. Therefore it would be interesting to find those parts inside the implementation where most of the time is spent. Further it would desirable to observe on which threshold value the produced overhead by using more threads exceeds the increase in efficiency.

Approach

The method profiling tab in the profiler offers an appropriate view to approach this problem. The methods and their total time spent is displayed, this offers a good comparison to find the slowest parts of the program. For the second problem this view is also useful. In addition the bar chart displaying the thread states over time could be useful as well, whether the threads' states indicates the produced overhead by the thread creation.

Profiling

A list of 40.000 Integers is sorted concurrently by 2 threads and 1 control thread on a machine with two processor cores.

Method	Time (%)	Time (ms)	Clock Cycles	#Invocations
Sorts.mergeSort	56,613	3,0	6.772.974	79998
ConcurrentMergeSort.main	5,738	24.290,0	54.921.034.003	1
Thread.join	4,185	4.429,2	10.014.723.311	4
Thread.run	4,155	8.794,5	19.885.212.945	2
Sorts.run	4,155	8.794,5	19.884.863.870	2
ConcurrentMergeSort.generateIntegerList	3,132	13.259,0	29.979.229.802	1
Random.nextInt	2,863	0,3	684.763	39999
Random.next	2,218	0,2	530.324	40000
Object.wait	2,092	4.428,5	10.013.413.368	2
Sorts.merge	1,139	0,1	272.660	39999
AtomicLong.compareAndSet	1,017	0,1	242.933	40000
AccessController.doPrivileged	0,775	65,7	148.542.818	50
PrintStream.printf	0,489	2.071,0	4.684.392.704	1

Figure 4.7: Methods in which most of the time was spent during the execution of Concurrent Merge Sort

Evaluation

The most significant observation was the huge impact on the overall execution time. A list of 40.000 Integers is typically sorted in about 188 ms on the machine used for profiling. On the same machine the sorting took about 45.204 ms with the profiler agent being executed. This is the first example on which the performance impact is actually noticeable.

Taking the results of table 4.7 into account, most of the time is spent in the method which actual performs the sorting: *Sorts.mergeSort*. Apart from *ConcurrentMergeSort.main* which executes the whole program, a large percentage of time is spent in the only synchronization mechanism left: *Thread.join*. The next interesting methods are *Random.nextInt* which took about $2,8\%$ of the total time. According to the results of the profiler it seems there is little left for optimization. It could be interesting to research if it is possible to improve the performance of *Sorts.mergeSort* by using other operations and other data structures.

A threshold value for indicating when the produced overhead by the thread creation exceeds the increase in efficiency could not be found. It was visible to the profiler that more and more threads had an increasing contention, because the threads' states over time was heavy influenced by the states *Blocked* and *Waiting*. Also the time spent in *Thread.join* was increased from about 4% to 8%. But in the end these results were to vague to give an exact threshold value.

4.2.5 Instrumenting Explicit Locks

ReentrantLock is a mutual exclusion lock with the same semantics as the monitor lock, but with additional features, for instance timed lock waits, interruptible lock waits, fairness and the ability to implement non-block-structured locking [GBB+06, p. 285]. *ReentrantLock* implements the interface *Lock* which can be seen in listing 4.6. The equivalent for entering and leaving a *synchronized* block are invocations to *Lock.lock* and *Lock.unlock*.

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException();
  void unlock();
  Condition newCondition();
}
```

Listing 4.6: Interface definition of *java.util.concurrent.locks.Lock*

Problem

Mutual exclusion is a strong locking discipline. Often data-structures are mostly read and only sometimes written. A read write lock strategy would allow multiple readers, but only one writer at a time. This capability is provided by the *ReadWriteLock* interface, which exposes two *Lock* objects, one for reading access and one for writing access. The interface definition is shown in listing 4.7. Internally there is only one locking structure on which the actions are performed. In some cases the *ReadWriteLock* performances better than mutual exclusion and sometimes it performs slightly worse [GBB+06, p. 286].

```
public interface ReadWriteLock {
  Lock readLock();
  Lock writeLock();
}
```

Listing 4.7: Interface definition of *java.util.concurrent.locks.ReadWriteLock*

In the proceeding *Relocker* by Max Schäfer, Manu Sridharan, Julian Dolby and Frank Tip a program is presented which transforms monitor locks into *ReentrantLocks* and *ReentrantLocks* into *ReadWriteLocks*. The performance trade-off can differ heavily. For their paper a synthetic benchmark was implemented to test all three synchronization mechanisms on a *Map*. A defined number of writer and reader threads is spawned which execute random read and write operations to the *Map*. The throughput is measured in number of total operations per second [SSDT11].

The mix of reader and writer threads and the chosen architecture influences the outcome of this comparison. It would be interesting to see in how far the performance difference can be detected when profiling the same synthetic benchmark.

Approach

The three synchronization mechanism are working internally differently. On that account not the same strategy can be used to approach this problem.

1. Access to the monitor lock can be tracked in the usual way of observing the monitor log view and evaluating the statistical results on the occurred events.
2. Access to *ReentrantLock* and the *ReadWriteLock* is harder to track, since the JVMTI does not provide event hooks like it does for monitor related events.

ReentrantLock and *ReadWriteLock*, however, are implemented by using *AbstractQueueSynchronizer*. From the Java source code, as seen in listing 4.8, it can be derived that the lock is gained by invoking *AbstractQueueSynchronizer.acquire*.

```
public static class ReadLock implements Lock  {

  public void lock() {
    sync.acquireShared(1);
  }
  // ...

public static class WriteLock implements Lock  {

  public void lock() {
    sync.acquire(1);
  }
  // ...
```

Listing 4.8: Extract of source code for the *ReadLock* and *WriteLock*

Further the documentation states, when the lock is already held, the thread becomes disabled until the lock is released. Frequent disabling should become visible in the threads state information provided by the profiler. The second approach is to evaluate the results of the method profiling view. How much time was spent in the *lock* method could give indications whether contention has emerged.

Profiling

The benchmark was initialized with 2 reader and 1 writer threads. A reader thread reads 1000 times from the shared *Map* and a writer thread writes 1000 times to the shared *Map*. The results are shown in table 4.7 and 4.8 and 4.9.

Thread	Role	#contention
Thread-0	Reader	279
Thread-1	Reader	274
Thread-2	Writer	270

Table 4.7: Number of times a thread was in contention with another thread

Method	Time (%)	Time (ms)	#Invocations
ReentrantSyncMap.containsKey	8,574	1.398.782.574	9691
ReentrantLock.unlock	6,961	511793545	21504
ReentrantLock.lock	6,884	506080847	21505
ReentrantLock$NonfairSync.lock	6,669	49026705	21505
AbstractQueuedSynchronizer.release	6,651	488992358	21504
AbstractQueuedSynchronizer.acquire	5,742	1865171978	4867
AbstractQueuedSynchronizer.acquireQueued	5,412	1763032115	4853
AbstractQueuedSynchronizer.unparkSuccessor	5,168	1197296572	6824
LockSupport.unpark	5,029	116525269	6824

Table 4.8: Methods in which most of the time was spent during the execution using ReentrantLock

Method	Time (%)	Time (ms)	#Invocations
ReentrantReadWriteLock$ReadLock.lock	8,445	6,7	18572
ReadWriteMap.containsKey	8,395	13,8	9242
AbstractQueuedSynchronizer.acquireShared	6,6	6,9	18573
ReadWriteMap.size	5,402	10,5	7256
AbstractQueuedSynchronizer.doAcquireShared	16,9	2,5	2661
AbstractQueuedSynchronizer.parkAndCheckInterrupt	4,0	8,2	4181
LockSupport.park	3,963	2,8	4181
ReentrantReadWriteLock$Sync.tryAcquireShared	3,7	2,7	21984
ReadWriteMap.put	3,783	3,0	2001
Unsafe.park	3,699	1,4	4181

Table 4.9: Methods in which most of the time was spent during the execution using ReadWriteLock

Evaluation

At first glance the contention seems to be quite high for the case of synchronizing the *Map* with monitor locks. Until taking a look at the monitor log it was not clear what caused so much contention, even though every reader executes 1000 reads. There are two methods on which the reader thread might content: *size* and *get*. Since *Map.size* is invoked in every iteration step, this explains what multiplier lead to the number of contention.

Unfortunately the attempt failed to compare the three different locking mechanism with each other. Especially the profiling of the explicit locks does not contain enough information in order to attribute it in a meaningful way. The first approach to counter this problem could be the implementation of measuring how much time the contention took using monitor locks. For the explicit locks once again the time between executing *Lock.lock* and *Lock.unlock* could be measured. In addition it would be necessary to check the threads' state when entering *Lock.lock* for actually being able to detect contention is explicit locks. This use case demonstrated the limitations of the profiler, especially with respect to profiling higher-level locking primitives from the *java.util.concurrent* package.

4.2.6 Dynamic Lock-Ordering Deadlock

Indiscriminate use of locking can cause *lock-ordering deadlocks*. Since the JVM does not recover from deadlocks it makes sense to check whether the application ensures the preclusion of conditions which can cause this liveness hazard [GBB$^+$06, p. 205].

Problem

Considering the following code in listing 4.9 which was adapted from Java Concurrency in Practice [GBB$^+$06, p. 208]. The method transfers money from one account to another. Both account locks are acquired in order to update both balances atomically. The example instantiates two accounts and executes the *transferMoney* method concurrently for every account. It would be interesting to see, how the resource-allocation graph is able to visualize the occurring contention.

Approach

The resource-allocation graph view will be sufficient for this example.

Profiling

The program did not terminate and the profiler found a circle in the graph which means that a deadlock has occurred.

```java
public void transferMoney(Account fromAccount, Account toAccount, int amount)
    throws InsufficientFundsException {

  synchronized (fromAccount) {
    synchronized (toAccount) {
      if (fromAccount.getBalance().compareTo(amount) < 0) {
        throw new InsufficientFundsException();
      } else {
        fromAccount.debit(amount);
        toAccount.credit(amount);
      }
    }
  }
}
```

Listing 4.9: Dynamic lock-ordering deadlock

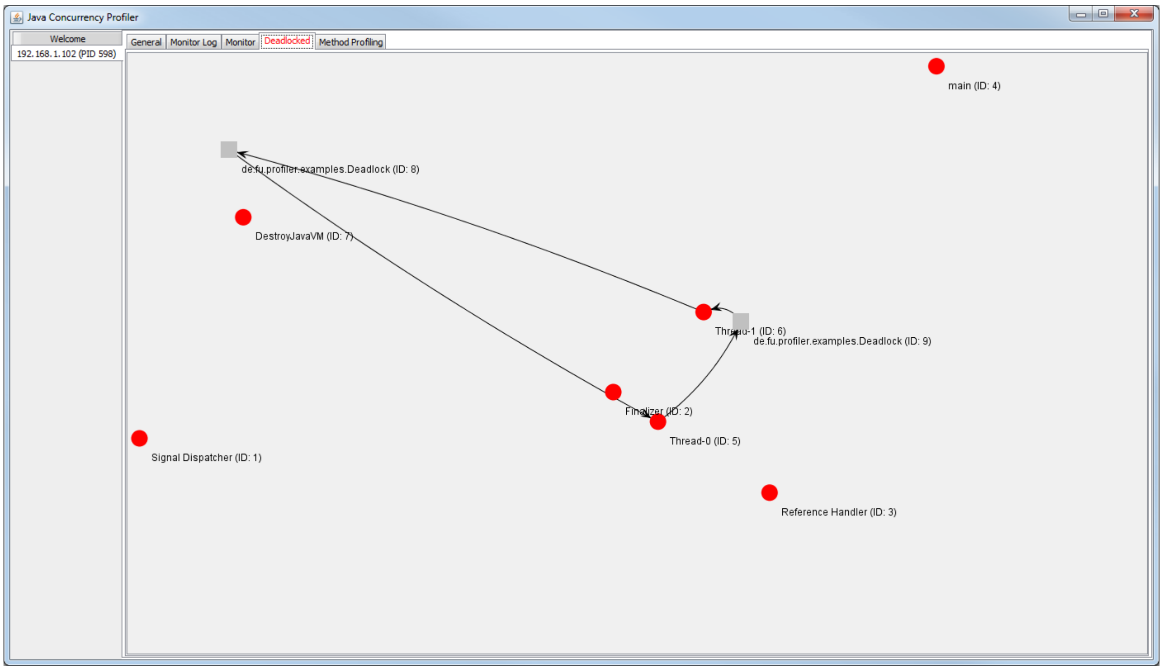


Figure 4.8: Illustrating the occurrence of a deadlock with the dynamically constructed resource-allocation graph

Evaluation

The first monitor lock acquisition of both threads succeeded. But when trying to acquire the other monitor lock both threads contended with each other. The graphical user interface alerts with a change to the tab that a deadlock has been found. The circle can clearly be seen in the resource-allocation graph which is shown in figure 4.8.

Revisiting the code shown in listing 4.9 it seemed like all threads acquire the locks in the same order. But the actual lock-order depends on the arguments passed into the method. A deadlock can arise, when two threads are calling *transferMoney*: thread A wants to transfer money from *Account X* to *Account Y* and Thread B wants to do the opposite. Lock-ordering deadlocks can happen if threads acquire the same locks in a different order. A program will be free of lock-ordering deadlocks if all threads acquire the locks they need in a fixed global order [GBB+06, p. 208].

It is obvious that the resource-allocation graph cannot point out the actual design flaw, but it still might help to find it. Observing the current state of a resource-allocation graph and trying to understand the steps which have occurred can help to reproduce to some degree how the program was executed.

4.2.7 Synchronizing on Strings

The monitor lock on which the synchronization should be performed can be selected freely. This means that the monitor lock does not have to be *this*, like it is the case when adding the *synchronized* modifier to a method. An example of this possibility can be seen in listing 4.10. A *FileServer* has a file storage and offers two types of requests: *defaultRequest* and *fileRequest*. Both request types have to be executed atomically. Since the *defaultRequest* and the *fileRequest* can be executed independently it is desired to reduce the contention, respectively increase the throughput. For this reason both classes, *FileServer* and *Storage* have their own private lock object. For self explanatory reasons a string literal is used.

Problem

The problem arises when deploying the *FileServer*. It seems like the desired improvement in throughput could not be achieved. In order to check, whether the synchronization is performed as intended the profiler is applied.

Approach

The monitor view, monitor log view and the resource-allocation graph view should be appropriate since monitor related events have to be traced.

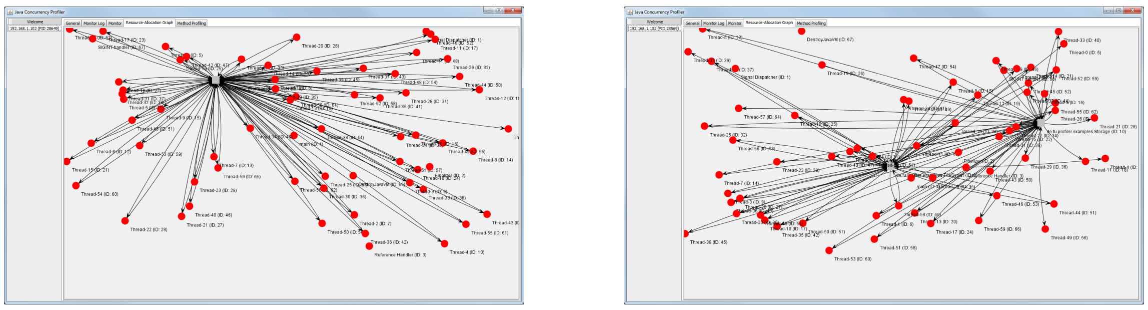
Profiling

Class	Context
de.fu.profiler.examples.FileServer	FileServer.defaultRequest Storage.getFile

Table 4.10: Listing the profiled monitors and in which context each monitor was accessed in the *Synchronizing on Strings* example

```java
public class FileServer {

  int hits = 0;
  Storage storage = new Storage();
  private String lock = "LOCK";

  public void defaultRequest() {
    synchronized (lock) {
      ++hits;
      // some default processing
    }
  }

  public File fileRequest(int id) {
    return storage.getFile(id);
  }
}

class Storage {

  Map<Integer, File> files = new TreeMap<Integer, File>();
  private String lock = "LOCK";

  public File getFile(int id) {
    synchronized (lock) {
      return files.get(id);
    }
  }
}
```

Listing 4.10: Synchronizing on string literals



(a) One monitor in use (b) Two monitors in use

Figure 4.9: Displaying the different resource-allocation graph patterns in the *Synchronizing on Strings* example

Evaluation

In figure 4.10 it can be seen how one monitor lock was used during the whole profiling, unlike two monitors which was the intention of the program presented above. The difference between both cases are also illustrated in the resource-allocation graph view, which can be seen in figure 4.9.

The reason for this problem is rooted in how the Java programming language handles strings. In the Java Language Specification it is stated, that string literals are interned in order to share unique instances. This means, using a string literal as monitor lock can lead to deadlock or performance loss. This concurrency bug was already present several times in the past. One example is the *BoundedThreadPool* class of the open source HTTP server Jetty. The code contained a *synchronization* block using a string literal as lock. This bug was reported in their issue tracker [Fou11].

4.3 Evaluation

When using the profiler the first time, it is noticed, that the amount of data being revealed to the user goes beyond the written code. This already starts in the general view, where threads were shown which are used for JVM maintenance: Reference Handler, Signal Dispatcher, Finalizer and DestroyJavaVM. A similar situation was experienced when dealing with the monitor log view. Used monitors and related actions to it, like lock acquisition and waiting and signaling, are not only shown because the developer made invocations to them in the code, but also because Java is using them for implementations across the library code. Both can lead to more information than expected and therefore care has to be taken, when which event is within the scope of the application being profiled.

Often it is hard to compare the profiled results due to their different implementation. This was best seen in the use case *Instrumenting Explicit Locks.* The lack of particular capabilities demonstrated the profilers incompleteness. But then again, this is also an approach to explore methods which improve the extraction of information. Based on these results new strategies to extend the profiler can be developed.

A clear advantage of the profiler was shown when dealing with a large number of interacting threads. The monitor log view was quickly overloaded with information and it is hard to figure out how single threads interacted with each other. The concept of filtering has proven to be useful, since it allowed to compare a choice of threads with each other. The same is true for filtering the information by using other criteria, for example event types.

Delivering stack traces for every event, wherever it made sense, helped to understand the rest of the information even better. For instance, when different roles were assigned to threads it was unclear how to recognize these roles from the profiler view. The stack trace revealed which code the threads were executing, thus helping to classify the threads.

It is interesting to see how few information can lead to meaningful conclusions. This was shown in the example *Extending A Thread-Safe Class* and *Synchronizing on Strings*, where the number and class type of used monitors made it possible to figure out erroneous behavior. In contrast, the visualization of threads' states did not help to clearly identify correlations. Since there outcome were to vague or too obvious, attempts to interpret the presented data failed. It is conceivable, that alternative presentations, for instance by presenting the state changes on a timeline, could improve the utility.

4.4 Overhead

In section 2.4.3 it was described that using a profiler will influence the program execution in any case. If it also the case, that method events are used to implement method profiling, the overhead will increase dramatically. This assumption was confirmed when profiling the use cases. Non-intensive computation tasks, as it was shown in the use cases *Overhead of Santa Claus Problem* and *Fairness of Bounded Buffer*, had little impact. This can be explained by the fact threads spent a large part of their execution time in the states *Waiting* and *Blocked*. When only few methods are executed, the overhead is negligible. The overhead increases at the point, where the timespan between events is decreased to a minimum. Otherwise, the agent has sufficient time for transmitting the messages.

In contrast, when dealing with computation intensive tasks the CPU is busy executing instructions. Every method exit leads to native profiler code being executed. For instance, as seen in the use case *Bottleneck of Concurrent Merge Sort* there are many method invocations in a short period of time. This minimizes the timespan between two occurring events. In order to counter this problem it is proposed to apply bytecode instrumentation instead. An alternative approach was already discussed in section 3.3.5: sampling instead of tracing, the profiler dumps periodically a stack trace of every thread and analyzes it. In order to measure the incurring overhead selected use cases were executed with and without the profiler agent.

	Profiler?	
Use Case	No	Yes
Overhead of Santa Claus	15.107 ms	18.390 ms
Fairness of Bounded Buffer	20.102 ms	21.543 ms
Dynamic Lock-Ordering Deadlock	95 ms	1.276 ms
Bottleneck of Concurrent Merge Sort	305 ms	51.707 ms

Table 4.11: Comparing consumed execution time between normal execution and profiled execution

The results can be seen in table 4.11. The differences between both types of runs behaved as expected. Little difference in execution time could be observed on the use case *Fairness of Bounded Buffer*. This does not mean, that the overhead is minimized when used in a producer consumer environment. It is more due to the fact, that each producer and consumer invokes *Thread.sleep* in order to simulate processing of their operations. Depending whether the processing of their operations is a computation intensive task or a non-computation intensive task is decisive for more realistic results.

Profiler?	Deadlocked	Terminated
Yes	286	714
No	102	898

Table 4.12: Comparing the number of times a deadlock has occurred between normal execution and profiled execution

Overhead is not only about increase of the total execution time. Every alteration of the program execution can be understood as overhead, respectively *Heisenbug* as defined in section 2.4.3. While testing the profiler such a behavior could only observed in one case: *Dynamic Lock-Ordering Deadlock*. It seemed like the program is more likely to deadlock when executed with the profiler agent. In order to confirm the assumptions a script was written which automatically executed the use case with and without profiling agent. The results are shown in table 4.12.

To sum up, it can be stated that the experienced overhead is far too large. The use cases are executing in a bearable time with respect to analyzing programs. This might change when applying the profiler on real applications. Before this is possible, the overhead has to be minimized a lot.

Chapter 5

Conclusion

The purpose of this bachelor thesis was the development and usage of a profiler. A profiler is a tool which is used during software development for measuring the execution of an application. It was shown that profiling can be understood in a broad sense, the measurable values differ in the same sense. It should also be clear now, that performance measuring is not the most important task, when your application does not perform its action in the same way the developer intended to.

The profiler works out very well with small applications. Already these small programs reveal the overhead produced by the profiler, an overhead which renders the profiling results sometimes into uselessness. It makes clear, how important it is to take the overhead into account, performance impact on the one hand and different executions' results on the other hand, as it was seen in the deadlock example.

Profiling allows a flexible and decoupled method to analyze the programs execution at a low level. The tool, however, is not as good as long as meaningful correlations can be observed. Most profilers concentrate on pure CPU and memory profiling. The profiler developed on the context of this bachelor thesis was an approach to profile applications in oder to reveal the underlying semantic of a program with respect to concurrency.

5.1 Outcome

Taking the described functional requirements in section 3.1 into account it was possible to fulfill all functional requirements. The JVMTI provides a sophisticated way to receive most of the required data. Even though there is a noticeable information gain, there are still limitations for certain areas. Often it would be better to receive a lot more information than provided by the callback facilities of the JVMTI. This, however, is also a question of which kind of information should be profiled.

The profiler developed in this work focused on counting statements. The exact number of times a certain method or action has occurred can be used to determine if the program is performing as expected. Precise examination of these results can uncover subtle errors.

At this level it is possible to compare the profilers functionality with the functionality of a debugger, which should rather show the number of times a piece of code is executed [GKM04]. This is in contrast to the profiler description given in chapter 1 which aims to give an overview of the whole execution. With concurrency this is not always possible and it is required to see each step done in the synchronization process. When, however, considering the fact, that the counting information is extended, such as adding a timestamp or a stack trace, the actual characteristic of a profiler is revealed: putting together all the statements on a timeline, with respect to their context, results in a profile which describes the programs execution. Then it confirms with the definition given in chapter 1 again.

Considering the two non-functional requirements efficiency and portability, there are more curtailments. Even though the overhead was an overarching topic which was always considered, the method profiling lead to a heavy impact on the program execution. In order to counter this problem, further filters were introduced. The portability is valid to some degree. The Java front end can be executed on every machine which can execute a Java Virtual Machine. The agent, however, relies on UNIX system functions and cannot be compiled for a windows machine as a dynamic linked library.

The overall solution is suitable when reflecting it to the problem statement described in section 1.2. The profiler is able to trace events with respect to concurrent Java, always attributing the events with respect to time. Information about threads and their interactions with each other could be included and are presented in different forms in the graphical user interface. Especially synchronization mechanisms, including access to locks, thread state changes and waiting and signaling could be made visible to the user. To some degree the graphical visualization is also a success, since the resource-allocation graph could be implemented with few means. The diagrams are helpful, but it is imaginable that there are a lot more possibilities in order to improve the process of communicating information to the user.

In the problem statement it was also described to trace events related to higher-level synchronization constructs like *java.util.concurrent.Semaphore*. Unfortunately this was not implemented at all. This is due to different reasons. Among other topics this will be discussed in detail in the following section which covers future work.

5.2 Future Work

Even though the profiler covers the basic mechanism for synchronization and thread coordination this is not sufficient. The *java.util.concurrent* package is a rich library and offers many higher-level abstraction of concurrency tasks which can ease the implementation of concurrent software. Since most of the data structures provided by this package implement their own synchronization and do not rely on monitor locking or monitor waiting and signaling, it is hard for the profiler to detect this during run time. An example would be to list all the used *Semaphore* objects and track their current state and list when a thread invokes *Semaphore.acquire* and *Semaphore.release*. This, however, cannot be implemented in a meaningful way without bytecode instrumentation.

5.2.1 Bytecode Instrumentation

Bytecode instrumentation is the modification of Java classes during run time. This has the advantage, that the code instrumentation is limited to specified Java classes. The callback functions in a profiling agent are invoked every time the appropriate event occurs. Bytecode instrumentation to the opposite executes only the extra code which was inserted for the selected class. The JVMTI supports bytecode instrumentation natively. Libraries like ASM allow class transformation by inserting Java code. As a result, this technique is a lot more efficient compared to pure JVMTI event hook profiling. In addition the code is optimized by the Just-In-Time compiler. [Ora07].

5.2.2 Record Scheduling Decisions

Another useful improvement of the current profiler would be the calculation of scheduling decision. It would offer a higher abstraction of the results currently provided by the profiler. Scheduling decision could include which thread of a group of threads waiting for a condition to become true is able to proceed after notification. These results could be tracked over time and be summarized. This way it would be possible to check fairness properties which is supported by some data structures provided in the *java.util.concurrent* package.

5.2.3 Overhead Reduction

With the reasons stated in the section 4.4 it is suggested to investigate how the current profiling can be improved by using bytecode instrumentation. This way the overhead, especially produced due to method profiling, would be reduced. It would be interesting to see in how far the performance impact changes. Further methods could be explored which can be used to do sophisticated overhead calculations in order to modify the profiled results to more realistic results.

5.2.4 Interactive Profiling

Another improvement would be an extension to the profiler agent side, which could not only send messages, but receive messages in order to change the behavior. This way it would be possible to start method profiling not prior to the user. Then only those capabilities are used that are actually needed which is in analogous to the design decisions for the JVMTI [VL00].

A related topic would be the introduction of ad-hoc profiling. VisualVM and YourKit can attach to a JVM which is already running. This way, it would be possible to avoid shutting down already running processes. This has an advantage to those processes where it is not preferred to terminate them during operation.

5.2.5 Alternative Profiling Methods

JVMTI is not the only way of profiling Java applications. Another approach is to use aspect-oriented languages like AspectJ, which modifies the application at run-time, thus another method of code instrumentation. A profiling agent does not have to be written in a native language. Java agents allow similar facilities. Even though the functionality of a Java agent is a subset of the functionality of a native agent, it still would be interesting to see in how far both differ or whether profiling tools are written more effectively and efficiently in Java.

Bibliography

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Third Edition*. Addison-Wesley, 2000.

[And99] G.R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 1999.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Ba98] Mordechai Ben-ari. How to Solve The Santa Claus Problem. *Concurrency: Practice & Experience*, 10:485–496, 1998.

[Fou11] Eclipse Foundation. Jetty Issue # 352 – Don't use strings as locks. WWW, 2011. accessed July 18, 2011, 05:36pm `http://jira.codehaus.org/browse/JETTY-352`.

[GBB$^+$06] Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, and Tim Peierls. *Java Concurrency in Practice*. Addison-Wesley Longman, Amsterdam, 2006.

[GKM04] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 39:49–57, April 2004.

[Goo11] Google. Protocol Buffers - Google's Data Interchange Format. WWW, 2011. accessed July 17, 2011, 09:10pm `http://code.google.com/p/protobuf/`.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning From Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. *SIGOPS Oper. Syst. Rev.*, 42:329–339, March 2008.

[MM00] Zakaria Maamar and Bernard Moulin. An Overview of Software Agent-Oriented Frameworks. *ACM Comput. Surv.*, 32, March 2000.

[MQ07] Madanlal Musuvathi and Shaz Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[OFN11] Joshua O'Madadhain, Danyel Fisher, and Tom Nelson. JUNG - Java Universal Network/Graph Framework. WWW, 2011. accessed July 17, 2011, 12:02am `http://jung.sourceforge.net/`.

[O'H11] Kelly O'Hair. The JVMPI Transition to JVMTI. WWW, 2011. accessed July 18, 2011, 08:16pm `http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition/`.

[Ora07] Oracle. Java Virtual Machine Tool Interface (JVM TI) 1.2 Reference. WWW, 2007. accessed July 18, 2011, 07:55pm `http://download.oracle.com/javase/6/docs/platform/jvmti/jvmti.html`.

[Ora11a] Oracle. Java Platform, Standard Edition 6 API Specification. WWW, 2011. accessed July 18, 2011, 07:23pm `http://download.oracle.com/javase/6/docs/api/`.

[Ora11b] Oracle. VisualVM – a debugging and performance analysis tool. WWW, 2011. accessed July 18, 2011, 01:35am `http://visualvm.java.net`.

[Ray96] Eric S. Raymond. *The New Hacker's Dictionary (3rd Edition)*. MIT Press, Cambridge, MA, USA, 1996.

[SE04] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. *SIGPLAN Not.*, 39:528–539, April 2004.

[SG98] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts, 5th Edition*. Addison-Wesley-Longman, 1998.

[SSDT11] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Refactoring Java Programs for Flexible Locking. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 71–80, New York, NY, USA, 2011. ACM.

[Tro94] John A. Trono. A New Exercise in Concurrency. *SIGCSE Bull.*, 26:8–10, September 1994.

[VL00] D. Viswanathan and S. Liang. Java Virtual Machine Profiler Interface. *IBM Syst. J.*, 39:82–95, January 2000.

[Wil07] Andrew Wilcox. Build Your Own Profiling Tool. WWW, 2007. accessed July 16, 2011, 12:50 http://www.ibm.com/developerworks/java/library/j-jip/.

[You11] YourKit. YourKit – a Java and .NET profiler. WWW, 2011. accessed July 18, 2011, 01:30am http://www.yourkit.com/.