Execution Time Limitation of Interrupt Handlers in a Java Operating System

Meik Felser, Michael Golm, Christian Wawersich, Jürgen Kleinöder
University of Erlangen-Nürnberg
Dept. of Computer Science 4 (Distributed Systems and Operating Systems)
Martensstr. 1, 91058 Erlangen, Germany
{felser, golm, wawersich, kleinoeder}@informatik.uni-erlangen.de

Abstract

Device drivers are a very critical part of every operating system. They often contain code that is executed in interrupt handlers. During the execution of interrupt handlers, the processing of some other interrupts is usually disabled. Thus errors in that code can compromise the whole system.

This paper describes an approach to ensure that an interrupt handler is not allowed to use more than a specified amount of time. Our approach is based on a Java operating system and consists of a combination of verification at compilation time and run-time checks.

1 Introduction

Modern operating systems have to support a wide range of different hardware. To be flexible the code to access the hardware is modularized in device drivers. Each driver is responsible for the communication between the operating system and the corresponding device.

The communication between hardware and software is often realized with interrupts. Hence drivers have an exceptional position because they must be able to react to interrupts. During the execution of an interrupt handler the processing of new interrupts is usually disabled. Thus an error in an interrupt handler, e.g. an endless loop, does not only affect the functionality of the driver but can have severe effects on the whole system.

Beside the worst case of an interrupt handler containing an endless loop every interrupt handler, being executed exclusively, has an influence on the response time of the system. The scheduling, in particular real-time scheduling, is reliant on the possibility to interrupt a running thread. In almost every system this is realized by the timer interrupt. But this interrupt is delayed if the processor is executing another interrupt handler.

A common approach to minimize the execution time of an interrupt handler is to split it into two parts. A first-level handler is activated directly by the interrupt. This handler performs only time-critical actions and unblocks a second-level handler which performs all other tasks. However a faulty first-level handler can still compromise the whole system.

In this paper we introduce an approach to ensure an upper bound execution time for first-level interrupt handlers. We describe a Java bytecode verifier that is part of our Java operating system JX [GFW+02]. All device drivers for this system, including their interrupt handlers, are completely written in Java. The verifier can either predict an upper bound execution time for a method when the bytecode is translated into machine code or it extends the method to execute run-time checks.

In the next section we classify the possible errors of device drivers. After that we introduce our verifier in sections 3. In section 4 we describe the execution time analysis. Section 5 concludes the paper.

2 Classification of Errors

To estimate the probability of faulty device drivers, we take a look at the Linux kernel. More than 77% of the source code lines are device drivers (Linux 2.4.18). The large amount of supported devices implies that not every possible combination of drivers was tested. Beside implementation errors, side effects can cause unpredictable errors. Other studies e.g. [CY01] circumstantiate that most errors are found in device drivers.

For the further analysis we divide the errors in device drivers into three classes, according to the effect of their misconduct:

- Errors causing an instant system crash.
- Errors causing an instant crash of the device driver.
- Errors that do not crash the driver but lead to unexpectedly long execution.

The first class of behavior can be caused by faulty DMA transfers, for example, due to a wrong initialization of the device's DMA engine. It is very hard to counteract these effects in general, without any knowledge of the individual device. In the scope of this paper we trust the DMA initialization sequence of the driver ([GKB01]) and focus on the other classes of errors.

Implementation errors, such as buffer overflows or invalid pointers often lead to segmentation violations (or similar faults) and characterize the second class of errors. Using Java makes many of these bugs impossible because of the typesafety and the boundary checks at array accesses. But even without a typesafe language, an appropriate protection mechanism can delimit the effect of these errors, such that only the driver is compromised. After removing the driver, a new instance can be started, assuming that the error does not occur again, at least not immediately.

Errors of the third class are mainly based on wrong preconditions or caused by unexpected side effects from other device drivers. But also simple implementation faults can provoke these effects. If we do not fully trust our device drivers we must be able to handle such errors.

3 The Bytecode Verifier

When a Java class is loaded by our system, it's bytecode is analyzed by a verifier and translated into machine code. The verifier performs several tasks:

- It checks whether the byte code complies with the JVM specification.
- It tries to analyze the worst-case execution time.
- It checks the applicability of some optimizations. The null pointer analysis, for example, checks whether a reference can never become null. In this case the compiler does not need to insert a run-time check.

To prevent erroneous drivers especially the first two steps are important. In the first step the verifier checks whether the bytecode fulfills the Java virtual machine specification [LY99]. This type of verification is performed by most Java run-time systems. The major task is to ensure pointer safety. It must be guaranteed that no numeric value is interpreted as reference. This is mainly realized by testing whether the number and types of operands on the stack is correct for each operation. Besides this, the access restrictions of variables and methods are checked. The compliance to the JVM specification guarantees typesafety and thus eliminates all errors related to invalid pointers. The second step is described in the next section.

4 Execution Time Limitation

In this section we describe the worst-case execution time analysis of our bytecode verifier. In contrast to other papers (e.g. [OS97], [EE00]) we are not interested in the tightest possible worst-case execution time or the exact program flow. We only want to ensure that a specified amount of time is not exceeded.

4.1 Worst-case execution time analysis

To analyze the worst-case execution time for a method the verifier builds up a flow graph of the respective method.

In [Sha89] an additive rule is proposed to evaluate the worst-case execution time of a program based on the worst-case execution time of parts e.g. single instructions. With the prerequisite, that the worst-case execution time of every Java bytecode is known, we only have to find the longest path through the flow graph.

One problem is, that the run-time estimation of some bytecodes is very hard, for example the allocation of a new object can lead to an GC run of unpredictable duration. Therefore we prohibit the GC to run in an interrupt handler and have reserved a little amount of the heap for object allocation in interrupt handlers.

The second problem is, that the program flow is rarely linear. A conditional jump (e.g. caused by an *if* statement) splits the program flow into two branches. In the case of an *if* statement we have to estimate the execution time for each branch separately and use the maximum. It is more complex if the flow graph contains backward jumps leading to cycles (e.g. caused by loops). The execution time of simple loops with known length is calculated by multiplying the execution time of the body by the number of iterations. Therefore the problem is to determine the number of iterations for each loop.

We distinguish three types of loops:

- simple loops with fixed length,
- simple loops with simple condition function, and
- complex loops.

The first type is the easiest, but very few loops are of that type. The second type can be managed in an analytical approach [Bli94]. If the loop condition depends on a monotonic function, a fixed upper bound can be evaluated. But the number of iterations mostly depend on relatively complex loop conditions. A general approach to analyze the execution time of every kind of loop is impossible without further information from the user [Par93] or basic restrictions concerning the kind of loop conditions [PK89].

Since our goal is not to evaluate the worst-case execution time, but to limit it, we do not use an analytical approach to handle loops. Instead we use an approach of partial execution. *If* statements are handled as described above. But if we recognize a loop we execute its condition function to determine whether another iteration is executed.

4.2 Partial execution

On Java bytecode level loops are realized with conditional jumps as in most machine languages. Therefore the first task is to determine which conditional jump leads to a loop and which is only caused by an *if* statement. The correlation between the parameters of the loop condition and the number of iterations needs not to be analyzed. We only have to determine which parameters, more precisely which bytecodes, influence the number of iterations. These bytecodes are recognized in a recursive application of simplification rules [Alt01] which try to identify typical constructs, such as *if* conditions or simple loops. In the context of this paper we are not interested in the details of this process. We concentrate on the results. The analysis supplies us with a list of bytecodes which have to be executed to determine the length of loops. This splits the bytecodes into two categories. Those which have to be executed and those we only need to simulate.

To simulate a bytecode means that we only have to add the execution time of the operation to our sum. Whereas we need to evaluate a result for each bytecode which was marked for execution because it is needed by another operation, for example, a conditional jump. Special attention must be paid to the `invoke` bytecodes. These instructions are used to call another method. If these bytecodes are used, the verifier analyzes the called method and begins the partial execution of that method unless there already exists valid execution time information for that method (e.g. due to a prior analysis of the method).

If the execution of a bytecode depends on parameters which were defined outside the scope of the analysis, a start value for each parameter must be provided at the beginning of the partial execution. If we do not trust the source of these parameters we have to insert checks that verify the given values at run time and ensure that they match the expected values or are within an expected range of values.

The approach of partial execution is not able to estimate the worst-case execution time for all methods. For example if the method contains an endless loop, the partial execution loops infinitely. But this is not a problem since we are only interested, whether a method is executed within a specified amount of time. By checking the elapsed time during the partial execution this can be evaluated for almost every method. Thus this procedure is sufficient for our intention.

4.3 Checks at run time

Some situations prohibit the use of the partial execution approach. First it is not always possible to supply a suitable set of start values. Second the Java bytecode allows loop constructs which can not be handled by our simplification rules (e.g. if a basic block is shared by two loops), therefore the flow graph can not be reduced. Although bytecode of such complexity is never created by a Java to bytecode compiler, handwritten bytecode can contain such complex loop constructions and, anyhow, comply to the JVM specification.

In relation to interrupt handlers this is almost no problem. Most interrupt handlers are only dispatchers. Thus they have a linear control structure and the verifier can determine the worst-case execution time at compile time. Alternatively complex non-linear interrupt handlers can be split into a linear first-level handler and a more complex second-level handler. This would justify the alternative policy to only accept drivers which pass the execution time analysis at compilation time. But we do not want to be that restrictive.

Thus our verifier can handle these cases as well. It can analyze the worst-case execution time of one loop iteration and insert run-time checks into the method's bytecode. The checks monitor the number of iterations and can terminate the method with an exception if the specified time limit is exceeded. Alternatively the run-time checks can be created by our bytecode to native code compiler. Then the verifier supplies the native code compiler with hints, where to add those checks. The advantage is, that hardware clocks can be used to measure the exact time used by the method, whereas the bytecode variant is based on estimated data.

4.4 Terminating drivers

When an interrupt handler is terminated, the corresponding driver is uninstalled. This is reasonable, because next time this interrupt occurs, the handler may exceed it's time limit again. The termination of a driver leads to the following other problems:

- The data structures of a driver could be inconsistent.
- The device still generates interrupts, stressing the system.

To handle these problems a device driver must implement a `terminated` method. This method is called when the interrupt handler of the device is aborted. The task of this method is to clean up internal data structures and to reset the device.

The design of this method has to fulfill special requirements, because it is executed while interrupts are disabled. The verifier must be able to estimate the worst-case execution time of the method at compile time. If this is not possible, the driver is not allowed to be installed at all.

To fulfill these requirements the method can only contain loops where the number of iterations can be checked at compile time. In general, it should be possible to build a linear `terminate` method, so that this is not a serious restriction. If it is not possible, the `terminate` method can wake up another thread, which runs later with enabled interrupts and thus does not need to accomplish the strict requirements.

After cleanup, the system should be in an appropriate condition for installing another (or may be the same) driver for the device, so that applications which rely on the service of that specific device can still run.

5 Conclusion and future work

We can assure an upper bound execution time for every interrupt handler. This is done either at compile time or with checks at run time.

At compile time the verifier uses an approach of partial execution to determine the execution time. As a precondition a start value must be provided for each parameter. Up to now the result of our verifier is a table which contains the number of executions for each bytecode. To get a time information from this abstract data we need a mapping from the bytecodes to the worst-case execution time of each bytecode (cp. [PG01]). The simplest way to do this is a table of execution time information for each bytecode. This data depends on the target architecture and must be obtained in a calibration step.

If it is not possible to evaluate the worst-case execution time at compile time, we extend the critical methods with additional code. This code checks the elapsed time at run time and terminates the method if a specified amount of time is exceeded.

Thus our system is less vulnerable to erroneous interrupt handlers. In combination with the protection provided by the typesafe language Java, the occurrence of system crashes due to faulty device drivers is drastically reduced.

A side effect of the execution time limitation is that the system delay, due to the execution of interrupt handlers, is within certain limits. This may be profitable for real-time applications.

6 References

Alt01 M. Alt. Ein Bytecode-Verifier zur Verifikation von Betriebssystemkomponenten. Diplomarbeit (master thesis), University of Erlangen, Dept. of Comp. Science 4, July 2001.

Bli94 J. Blieberger. Discrete Loops And Worst Case Performance. *Computer Languages*, 20(3):193-212, 1994.

CY01 A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In: *Symposium on Operating System Principles 01'*, 2001.

EE00 J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In: *Proc. of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)*, Orlando, December 2000.

GFW+02 M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX Operating System. In: *Proc. of the Usenix Annual Technical Conference*, Monterey, June 2002.

GKB01 M. Golm, J. Kleinoeder, F. Bellosa. Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System. In: *Proc. of the 8th Workshop on Hot Topics in Operating Systems*, May 2001.

LY99 T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Second Edition, Addison-Wesley, Reading/Massachusetts, 1999

OS97 G. Ottosson and M. Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In: *Proc. of SIGPLAN 1997 Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, June 1997.

Par93 C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31-62, March 1993.

PG01 P. Puschner, G. Bernat. WCET Analysis of Reusable Portable Code. In: *Proc. of the 13th International Euromicro Conference on Real-Time Systems*, June 2001.

PK89 P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Journal of Real-Time Systems*, 1(2):159-176, September 1989.

Sha89 A. Shaw. Reasoning about Time in Higher-Level Language Software. I*EEE Transactions on Software Engineering*, 15(7):875-889, July 1989.