Realizing Iterative-Relaxed Scheduler in Kernel Space

Master-Arbeit
Sreeram Sadasivam
2662284

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik

Fachgebiet Dependable Embedded
Systems and Software
Prof. Neeraj Suri Ph.D

Realizing Iterative-Relaxed Scheduler in Kernel Space
Master-Arbeit
2662284

Eingereicht von Sreeram Sadasivam
Tag der Einreichung: 16. Marz 2018

Gutachter: Prof. Neeraj Suri Ph.D
Betreuer: Patrick Metzler

Technische Universität Darmstadt
Fachbereich Informatik

Fachgebiet Dependable Embedded Systems and Software
Prof. Neeraj Suri Ph.D

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Master-Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die schriftliche Fassung stimmt mit der elektronischen Fassung überein.

Darmstadt, den 16. Marz 2018 Sreeram Sadasivam

1 Background

1.1 Software Verification

Software programs are becoming increasingly complex. With the rise in complexity and technological advancements, components within a software have become susceptible to various erroneous conditions. Software verification have been perceived as a solution for the problems arising in the software development cycle. Software verification is primarily verifying if the specifications are met by the software[8].

There are two fundamental approaches used in software verification - dynamic and static software verification[8]. Dynamic software verification is performed in conjunction with the execution of the software. In this approach, the behavior of the execution program is checked- commonly known as Test phase. Verification is succeeding phase also known as Review phase. In dynamic verification, the verification adheres the concept of test and experimentation. The verification process handles the test and behavior of the program under different execution conditions. Static software verification is the complete opposite of the previous approach. The verification process is handled by checking the source code of the program before its execution. Static code analysis is one such technique which uses a similar approach.

The verification of software can also be classified in perspective of automation - manual verification and automated verification. In manual verification, a reviewer manually verifies the software. Whereas in the latter approach, a script or a framework performs verification.

Software verification is a very broad area of research. This thesis work is focused on automated software verification for multithreaded programming.

1.2 Multithreaded Programming

Computing power has grown over the years. Advancements are made in the domain of computer architecture by moving the computing power from single-core to multi-core architecture. With such advancement, there were needs to adapt the programming designs from a serialized execution to more parallelizable execution. Various parallel programming models were perceived to accommodate the perceived progression. Multithreaded programming model was one of the designs considered for the performance boost in computing[3].

Threads are small tasks executed by a scheduler of an operating system, where the resources such as the processor, TLB (Translation Lookaside Buffer), cache, etc., are shared between them. Threads share the same address space and resources. Multithreading addresses the concept of using multiple threads for having concurrent execution of a program on a single or multi-core architectures. Inter-thread communication is achieved by shared memory. Mapping the threads to the processor core is done by the operating system scheduler. Multithreading is only supported in operating systems which has multitasking feature.

Advantages of using multithreading include:

- Fast Execution
- Better system utilization
- Simplified sharing and communication

- Improved responsiveness - Threads can overlap I/O and computation.
- Parallelization

Disadvantages:

- Race conditions
- Deadlocks with improper use of locks/synchronization
- Cache misses when sharing memory

1.3 Concurrency Bugs

Concurrency bugs are one of major concerns in the domain of multithreaded environment. These bugs are very hard to find and reproduce. Most of these bugs are propagated from mistakes made by the programmer[9]. Some of these concurrency bugs include:

- Data Race
- Order violation
- Deadlock
- Livelock

Non-deterministic execution behavior of threads is one of the reasons for having the among mentioned bugs. Data race and order violation are classified as race condition bugs. Whereas deadlock and livelock are classified as lack of progress bugs.

1.3.1 Race Condition

Race condition is one of the most common concurrency problems. The problem arises when there are concurrent reads and writes of a shared memory location. As stated above, the problem occurs with non-deterministic execution behavior of threads.

Consider the following example, where you have three threads and they share two variables x and y [3]. The value of x is initially 0.

Thread 1	Thread 2	Thread 3
(1) x = 1	(2) x = 2	(3) y = x

Table 1.1: Race condition example

If the statements (1), (2) and (3) were executed as a sequential program. The value of y would be 2. When the same program is split to three threads as shown in the above Table 1.1, the output of y becomes unpredictable. The possible values of y = {0,1,2}. The non-deterministic execution of the threads makes the output of y non-deterministic. Table 1.2 depicts possible executions for the above multithreaded execution.

The above showcased problem is classified as race condition bug. Ordered execution of reads and writes can fix the problem.

1.3.2 Lack Of Progress

Lack of progress is another bug class observed in multithreaded programs. Some of the bugs under this class include deadlocks and livelocks.

Execution Order	Value of y
(3),(1),(2)	0
(3),(2),(1)	0
(2),(1),(3)	1
(1),(3),(2)	1
(1),(2),(3)	2
(2),(3),(1)	2

Table 1.2: Possible executions

Deadlock

Deadlock is a state in which each thread in thread pool is waiting for some other thread to take action. In terms of multithreaded programming environment, deadlocks occur when one thread waits on a resource locked by another thread, which in turn is waiting for another resource locked by another thread. If a thread is unable to change its state indefinitely because the resource requested by it are being held by another thread, then the entire system is said to be in deadlock[4].

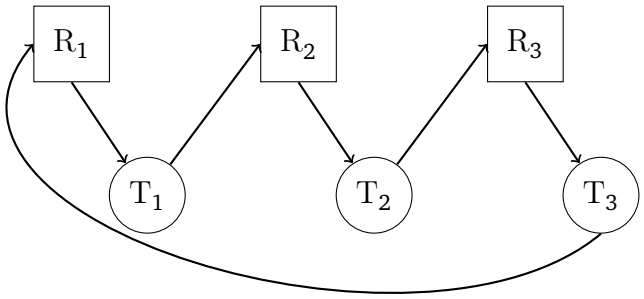

Figure 1.1: Dead Lock Example

In the example depicted in Fig 1.1, we have three threads T_1, T_2, T_3 and three resource instances R_1, R_2, R_3. The figure depicts hold and wait by each threads. Thread T_1 holds resource R_1 and waits for the acquisition of resource R_2 from thread T_2. T_2 cannot relinquish resource R_2, unless it acquires resource R_3 for its progress. But, resource R_3 is acquired by T_3 and is waiting for R_1 from T_1. Thus, making a circular wait of resources. This example clearly explains the dependency of resources for the respective thread progress.

Deadlock can occur if all the following conditions are met simultaneously.

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

These conditions are known as Coffman conditions[6].

Deadlock conditions can be avoided by having scheduling of threads in a way to avoid the resource contention issue.

Livelock

Livelock is similar to deadlock, except the state of threads change constantly but, with none progressing. Livelock is special case of resource starvation of threads/processes. Some deadlock detection algorithms are susceptible to livelock conditions when, more than one process/thread tries to take action[9][4]. The above mentioned situation can be avoided by having one priority process/thread taking up the action.

1.4 Model Checking

From section 1.3, it is very clear that there needs to be verification for multithreaded programs. The verification solutions range from detecting causality violations to correctness of execution[7]. Model checking is an example of such a technique. It is used for automatically verifying correctness properties of finite-state concurrent systems[5][2]. This technique has a number of advantages over traditional approaches that are based on simulation, testing and deductive reasoning. When solving a problem algorithmically, both the model of the system and the specification are forumulated in a precise mathematical language. Finally, the problem is formulated as a task in logic, namely to verify whether a given structure adheres to a given logical formula. The technique has been successfully used in practice to verify complex sequential designs and communication protocols[5]. Model checker tries to verify all possible states of a system in a brute force manner[1]. Thus, making state explosion as one of the major challenges which is discussed in detail in section 1.4.1. Model checking tools usually verify partial specification for liveness and safety properties[7]. Some of these techniques are discussed in section 1.4.4. Model checking algorithms generate set of states from the instructions program which are later analyzed. There is a need to store these states for asserting the number of visits made them are atmost once. There are two methods commonly used to represent states:

- Explicit-state model checking
- Symbolic model checking

1.4.1 State explosion problem

The state space of a program is exponential in nature when it comes to number of variables, inputs, width of the data types ,.etc. Presence of function calls and dynamic memory allocation makes it infinite[7]. Concurrency makes the situation worse by having interleaving of threads during execution. Interleaving generates exponential number of ways to execute a set of statements/instructions. There are various techniques used to avoid the state explosion problem.

1.4.2 Explicit-state Model Checking

1.4.3 Symbolic Model Checking

1.4.4 Techniques

1.5 Partial Order Reduction

Partial Order Reduction(POR) is a technique used for reducing the size of state space to be searched by a model checking algorithm[10].

1.6 Dynamic POR

1.7 Deterministic Multi-Threading

Bibliography

[1] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. Principles of model checking. MIT press, 2008.

[2] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. Systems and software verification: model-checking techniques and tools. Springer Science & Business Media, 2013.

[3] Richard H Carver and Kuo-Chung Tai. Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs. John Wiley & Sons, 2005.

[4] Sagar Chaki, Edmund Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. Concurrent software verification with states, events, and deadlocks. Formal Aspects of Computing, 17(4):461–483, 2005.

[5] Edmund M Clarke, Orna Grumberg, and Doron Peled. Model checking. MIT press, 1999.

[6] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. ACM Computing Surveys (CSUR), 3(2):67–78, 1971.

[7] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 27(7):1165–1178, 2008.

[8] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. Fundamentals of software engineering. Prentice Hall PTR, 2002.

[9] Carmen Torres Lopez, Stefan Marr, Hanspeter Mössenböck, and Elisa Gonzalez Boix. A study of concurrency bugs and advanced development support for actor-based programs. arXiv preprint arXiv:1706.07372, 2017.

[10] Doron Peled. Ten years of partial order reduction. In Computer Aided Verification, pages 17–28. Springer, 1998.