# 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

# Contents

# List of Figures

## List of Tables

## Abstract

Abstract comes here...

# 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[2].

There are two fundamental approaches used in software verification - dynamic and static software verification[2]. 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 focussed 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[1].

Threads are a small tasks executed by a scheduler on 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[3]. 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 [1]. 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 example 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. Ordered execution of reads and writes can fix the problem.

### 1.3.2 Deadlock

### 1.3.3 Livelock

| 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

Bibliography

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

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

[3] 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.