GCC Extension for Detecting Critical Sections in a Multithreaded Environment

Vishal Dawange[1], Mayur Jadhav[2], Akash Kothawale[3], Prasad Muley[4]
Department of Information Technology
Pune Institute of Computer Technology, University of Pune, Maharashtra, India
[1] vishalsd27@gmail.com , [2] rajemayur.413@gmail.com , [3] akash.kothawale@gmail.com , [4] pmmuley@gmail.com

***Abstract*—Concurrent programming is gaining popularity, since Moore's law is nearing its end of life. The significant truth is that processors aren't getting faster anymore, but you get more cores.**

In concurrent / parallel programming, one of the toughest tasks for programmers is synchronizing the access of shared memory to avoid basic problems of concurrency. While working on a large code base, programmers may miss out synchronizing certain critical section leaving the code vulnerable for race conditions to occur.

We propose an approach that extends the capability of GCC to identify all the critical sections in multithreaded programs which has synchronization bugs, i.e. race conditions may occur due to incorrect/no use of locking and unlocking mechanisms, which will warn the programmers by pointing out the exact location where the problems would occur.

Furthermore, it also provides synchronization to that section of code, by introducing proper locking methods. This stage is optional, and programmers may synchronize the section themselves too.

***Keywords*—Compilers, Multithreading, Critical Section, GCC, Locks.**

I. Introduction

In the recent years, from late 2005, clocks' speeds haven't advanced anymore. Instead, core counts have increased at the pace clock speed used to, and that has affected more and more programmers [1]. As programmers started migrating from single core to multicore architectural programming, many problems arose.

One of the most difficult parts of multithreaded programming is when programmers need to handle critical sections (A critical section of a multithreaded program is a section of code where shared data are accessed by the multiple threads) which are responsible for causing data races and are also one of the reasons for causing deadlocks. Data races occur due to incorrect or no synchronization of the threads in the programs.

In order to avoid data races, synchronization is achieved by using lock/unlock mechanism (semaphores, monitors, pthread_mutex, etc.) to limit a thread's concurrent access to shared resources, if it is being used by other thread.

To resolve the issue of manually handling each and every critical section or to debug large programs with synchronization bugs we are proposing an idea with which we identify critical sections which may cause data races, and avoid them by introducing proper synchronizations.

II. Background

A. Critical Sections

In concurrent programming, a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task, or process will have to wait for a fixed time to enter it (aka bounded waiting).

By carefully controlling which variables are modified inside and outside the critical section, concurrent access to that state is prevented. A critical section is typically used when a multithreaded program must update multiple related variables without a separate thread making conflicting changes to that data. In a related situation, a critical section may be used to ensure a shared resource, for example a printer, can only be accessed by one process at a time.

How critical sections are implemented varies among operating systems. [2]

B. Race Conditions

Race conditions arise in software when separate processes or threads of execution depend on some shared state. Operations upon shared states are critical sections that must be mutually exclusive. Failure to do so opens up the possibility of corrupting the shared state.

Race conditions are notoriously difficult to reproduce and debug, since the end result is nondeterministic, and highly dependent on the relative timing between interfering threads. Problems occurring in production systems can therefore disappear when running in debug mode, when additional logging is added, or when attaching a debugger, often referred to as a Heisenbug. It is therefore highly preferable to avoid race conditions in the first place by careful software design than to fix problems afterwards.

Race conditions occur especially in multithreaded, concurrent, parallel or distributed programs [2].

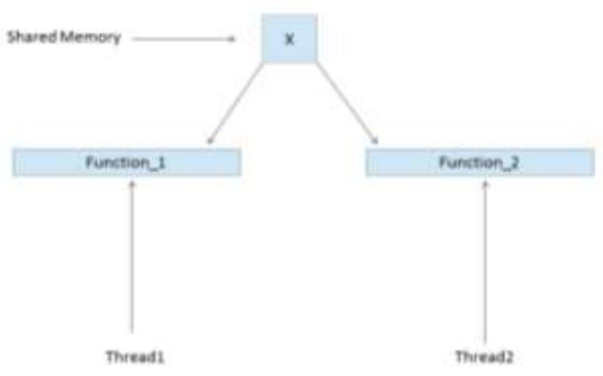

Fig. 1. Race Condition occurring in a critical section

C. Synchronization

Thread synchronization or serialization, strictly defined, is the application of particular mechanisms to ensure that two concurrently-executing threads or processes do not execute specific portions of a program at the same time. If one thread has begun to execute a serialized portion of the program, any other thread trying to execute this portion must wait until the first thread finishes.

Synchronization is used to control access to state in small-scale multiprocessing systems - in multithreaded environments and multiprocessor computers - and in distributed computers consisting of thousands of units [2].

III. Proposed Work & Implementation

The important aspect of the approach is that we are analyzing the programs statically. This helps us check each and every possible case that may occur during execution of the programs.

To implement the current idea of detecting critical sections, we have designed architecture as follows.

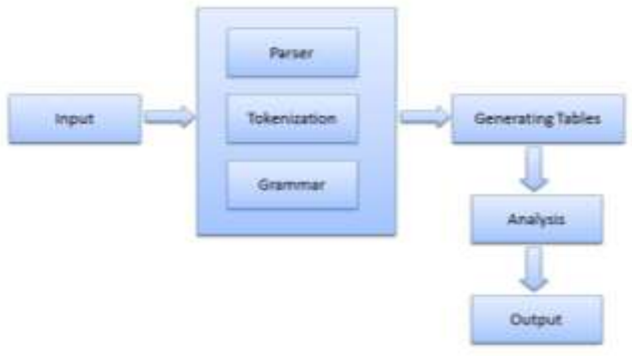

Fig. 2. Architecture.

A. Input File

A normal multithreaded program is given as input with critical sections. This file is first compiled to check for errors and only when it is compiled error free; it heads to the next stage.

B. Parser, Tokenizer, Grammar

Flex (The Fast Lexical Analyser) is a tool for generating scanners. A scanner, sometimes called a tokenizer, is a program which recognizes lexical patterns in text. The flex program reads user-specified input files, or its standard input if no file names are given, for a description of a scanner to generate [4].

With it we separate all the unwanted text (ex. Comments, pre-processors) and tokenize the program to find out shared memory, functions, threads, and locks.

In order to find the above from the token, it analyzes its input for occurrences of text matching the regular expressions for each rule. Whenever it finds a match, it executes the corresponding code [3].

The different in-built functions were monitoring are:

- pthread_create
- pthread_join
- pthread_cond_wait
- pthread_cond_signal
- sem_wait
- sem_post

C. Generating Tables

Yacc (Yet Another Compiler Compiler) provides a general tool for describing the input to a computer program. We specify the structures of the input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification [5].

Once all the unwanted text is ignored from the code, with the help of Yacc we generate the following data structures:

- Global variables' table (Id, access, variable_name, line_number)
  Stores the entries of all the shared variables that exist in the program

- User-defined Functions (Id, function_name, return_type, number_of_parameters, line_number)
  Stores the entries of all the functions defined by the user in the program.

- Functions' Local symbol table (Id, access_type, name, data_type, function_id, line_number)
  Stores all the local variables associated with a function.

- Semaphore table (id, name, function, location)
  Stores all the locations of sem_wait() and sem_post functions used in the code

- Thread table (Id, thread_name, function_name, function_id, thread_attribute, parameters, parent_thread)
  Stores all the threads associated with which functions and their attributes

- Variables' log table (Id, function_name, shared_object,scope,line_number, thread_function_id, thread_id)
  Stores a log of all variables which may cause data races and helps analysis the proof.

- Critical Section table (Id, shared_object, thread_function_index,minimum_location_of_critical_section, maximum_location_critical_section, function_name)

D. Analysis

Once all tables are generated we analyze them. If any shared variable exists in multiple functions and is unhandled i.e. it is not protected with the function calls *sem_wait(), sem_post()*, then we categorize that shared variable as unhandled critical section and add its entry to the critical section table.

After getting a table having all the unhandled critical sections, we analyze it for the location where we need to provides locks and unlocks.

From the Critical Section Table, we get the line numbers of first and last occurrence of shared variables' usage. Depending on their closeness, we decide the number of times we need to lock, and unlock.

E. Output

Once all tables are analyzed and we display all the unhandled critical sections that exist. The various attributes of the critical sections that are found are:

- The shared object which is being accessed my multiple threads
- The threads which are accessing the same shared object
- The functions in which the critical section exist
- The location pertaining to the line number of the critical section

We then decide if the programmer handles the critical sections himself, or we have to handle it. Once the permissions are given, we add *sem_wait() sem_post()*, recompile the code to produce bug free programs.

IV. Placement of Our Tool in GCC Hierarchy

Our tool exists adjacent to the current architecture of GCC but runs once the compilation is over and no errors are generated. It keeps a copy of the original source code, works on it, and pushes its warnings along with the warnings of the compiler

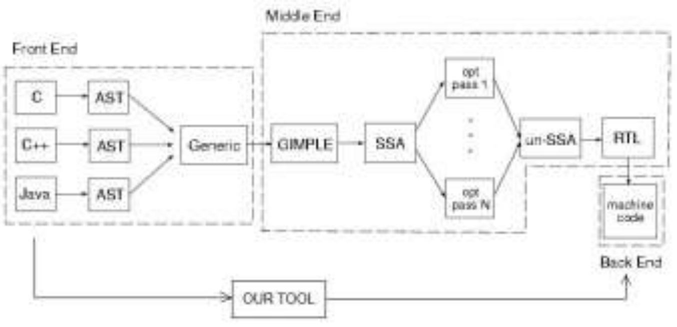

Fig. 3. Placement of our tool [6]

V. Conclusion

The overall approach helps solve many practical problems. Bugs that are notoriously difficult to find in concurrent programming are handled by the compiler itself. In large code bases data races possibilities be will perfectly identified and this framework will help automatically detect critical section and provide Lock/Unlock system without involvement of the programmer.

VI. Future Enhancements

The current implementation supports POSIX libraries. It can be further extended to support other thread and synchronization libraries. The concept has been implemented for GCC and can be ported to other compilers and programming language.

Go (http://golang.org) an expressive, concise, clean, and efficient language with its concurrency mechanisms makes it easy to write programs that get the most out of multicore and networked machines. Porting this concept to such languages will enhance its power even further.

Acknowledgment

Authors acknowledge the support extended by Dept. of Information Technology, Pune Institute of Computer Technology, for sanctioning our project. We also thank our guides Prof. Kirti. Nandedkar, Mr. Vedang Manerikar and Mr. Gaurav Jain for their inputs during the development.

References

[1] Dr. Niranjan, N. Chiplunkar, B. Neelima, Mr. Deepak, “Multithreaded Programming Framework Development for gcc infrastructure”, Computer Research and Development (ICCRD), 2011

[2] Critical Sections, Race Conditions, Thread Synchronization http://en.wikipedia.org/wiki/

[3] GCC - http://gcc.gnu.org/

[4] Fast Lexical Analyzer - http://flex.sourceforge.net

[5] Yacc - http://dinosaur.compilertools.net/yacc/index.html

[6] GCC Internals - http://gcc.gnu.org/onlinedocs/gccint