A

Project Report On

GCC Extension for Detecting Critical Sections in a Multithreaded Environment

Submitted By

Vishal Dawange	**B80058525**
Mayur Jadhav	**B80058549**
Akash Kothawale	**B80058564**
Prasad Muley	**B80058576**

Under the guidance of

Prof. K.K.Nandedkar

In partial fulfilment of

Bachelor of Engineering

[B. E. Information Technology]

[May 2013]

AT

Department of Information Technology

Pune Institute of Computer Technology

Dhankawadi, Pune 411043

Affiliated to

University of Pune

Pune Institute of Computer Technology

Department of Information Technology
Dhankawadi, Pune 411043

CERTIFICATE

This is to certify that the Dissertation entitled

"GCC Extension for Detecting Critical Sections in a Multithreaded Environment",

Submitted by

Vishal Dawange	**B80058525**
Mayur Jadhav	**B80058549**
Akash Kothawale	**B80058564**
Prasad Muley	**B80058576**

is a record of bonafide work carried out by them, in the partial fulfillment of the requirement for the award of Degree of Bachelor of Engineering (Information Technology) at Pune Institute of Computer Technology, Pune under the University of Pune. This work is done during year 2012-2013, under our guidance.

Prof. K. K. Nandedkar	**Prof. Dr. Emmanuel M.**	**Dr. P. T. Kulkarni**
Project Guide	HOD, IT Department	Principal PICT

Examination:

Examiner ——————————

Date:

Acknowledgements

I am profoundly grateful to **Prof. K.K.Nandedkar** for her expert guidance and continuous encouragement throughout to see that this project rights its target since its commencement to its completion.

I would like to express deepest appreciation towards **Dr. P. T. Kulkarni**, Principal PICT, Pune, **Prof. Emmanual M.** HOD Information Technology Department and **Prof. Manish R. Khodaskar** (Project Coordinator) whose invaluable guidance supported me in completing this project.

I am particularly grateful to **Mr.Vedang Manerikar** (Helpshift Inc.) and **Mr.Gaurav Jain** (Marvell Semiconducor) who allows me to work in the company.

At last I must express my sincere heartfelt gratitude to all the staff members of Information Technology Department who helped me directly or indirectly during this course of work.

Vishal Dawange

Mayur Jadhav

Akash Kothawale

Prasad Muley

CERTIFICATE

This is to certify that the project report entitled

GCC Extension for Detecting Critical Sections in a Multithreaded Environment

Submitted by

Vishal Dawange	**B80058525**
Mayur Jadhav	**B80058549**
Akash Kothawale	**B80058564**
Prasad Muley	**B80058576**

is a bonafide work carried out by them with the Sponsorship from DREAMZ Group under the supervision of Mr. Vedang Manerikar, Mr. Gaurav Jain and has been completed successfully .

(Mr. Vedang Manerikar)

(Designation)
External Guide

(Mr. Gaurav Jain)

(Designation)
External Guide

Place : Pune
Date:

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 : Critical Section, Race Condition, GCC, Locks

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the recent years, from late 2005, clocks speeds havent advanced anymore. Instead, core counts have increased at the pace clock speed used to, and that has affected more and more programmers. As programmers started migrating from single core to multicore architectural programming, many problems arose.

One of the most difficult parts of multi-threaded 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 threads 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.

1.1 Background

In multithreaded programs, we often need to handle cases where critical sections exist, as the programs are at risk of producing unexpected outputs. This is done by introducing locking mechanisms so that only one thread may alter the data at once. This is necessary to avoid race conditions, which generally lead to unexpected results if the locking mechanism doesn't exist.

1.1.1 Multithreading

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler. A thread is a light-weight process. The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share the latter's instructions (its code) and its context (the values that its variables reference at any given moment). On a single processor, multithreading generally occurs by time-division multiplexing (as in multitasking): the processor switches between different threads. This context switching generally happens frequently enough that the user perceives the threads or tasks as running at the same time. On a multiprocessor or multi-core system, threads can be truly concurrent, with every processor or core executing a separate thread simultaneously[9].

Many modern operating systems directly support both time-sliced and multiprocessor threading with a process scheduler. The kernel of an operating system allows programmers to manipulate threads via the system call interface. Some implementations are called a kernel thread, whereas a lightweight process (LWP) is a specific type of kernel thread that shares the same state and information [9].

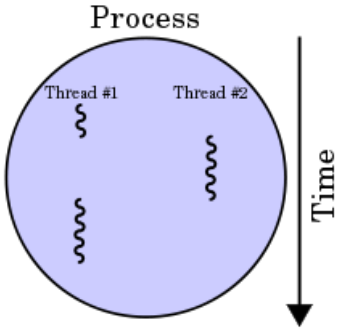

Figure 1.1: Threads

Thread-safeness:

Thread-safeness: in a nutshell, refers an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions. For example, suppose that your application creates several threads, each of which makes a call to the same library routine: This library routine accesses/modifies a global structure or location in memory. As each thread calls this routine it is possible that they may try to modify this global structure/memory location at the same time. If the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe.[6]

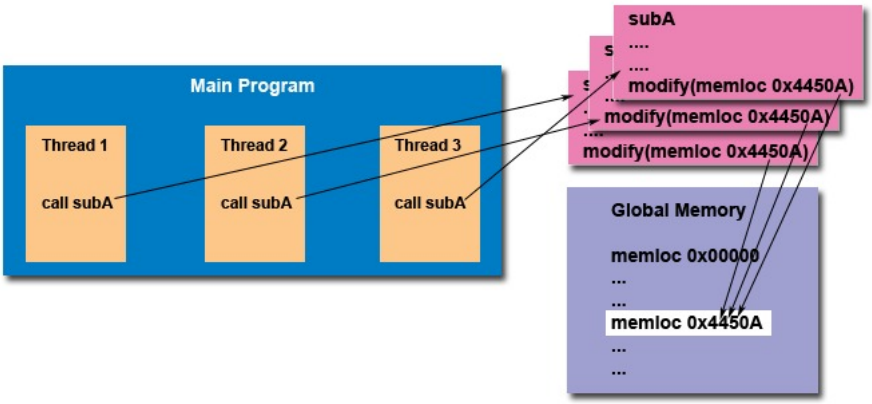

Figure 1.2: Multithreading

The implication to users of external library routines is that if you aren't 100 percent certain the routine is thread-safe, then you take your chances with problems that could arise. Recommendation: Be careful if your application uses libraries or other objects that don't explicitly guarantee thread-safeness. When in doubt, assume that they are not thread-safe until proven otherwise. This can be done by "serializing" the calls to the uncertain routine, etc [6].

Shared Memory Model:

All threads have access to the same global, shared memory Threads also have their own private data Programmers are responsible for synchronizing access (protecting) globally shared data [6].

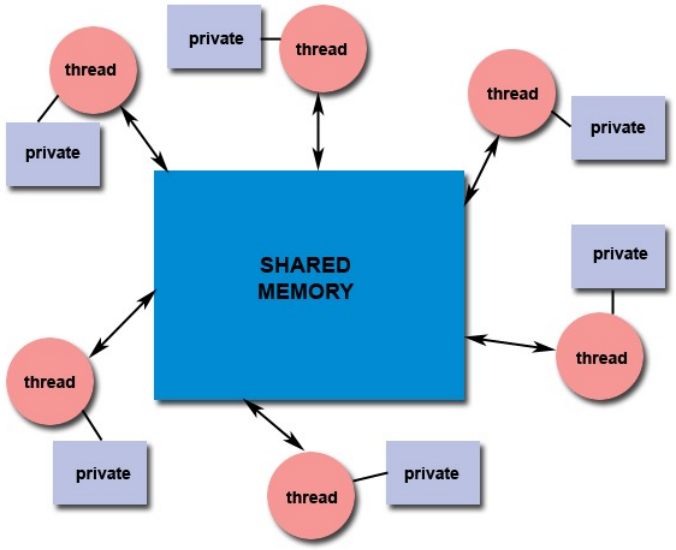

Figure 1.3: Shared Memory

1.1.2 POSIX thread (pthread) libraries

The POSIX thread libraries are a standards based thread API for C/C++. It allows one to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing. Threads require less overhead than ”forking” or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process. While most effective on a multiprocessor system, gains are also found on uniprocessor systems which exploit latency in I/O and other system functions which may halt process execution. (One thread may execute while another is waiting for I/O or some other system latency.)

Parallel programming technologies such as MPI and PVM are used in a distributed computing environment while threads are limited to a single computer system. All threads within a process share the same address space. A thread is spawned by defining a function and its arguments which will be processed in the thread. The purpose of using the POSIX thread library in your software is to execute software faster.

The subroutines which comprise the Pthreads API can be informally grouped into four major groups:

- Thread management: Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling etc.)

Creating and Terminating Threads:

  - pthread_create (thread, attr, start_routine, arg)
  - pthread_exit (status)
  - pthread_cancel (thread)
  - pthread_attr_init (attr)
  - pthread_attr_destroy (attr)

- Mutexes: Routines that deal with synchronization, called a ”mutex”, which is an abbreviation for ”mutual exclusion”. Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
  Mutex Variables:

  - pthread_mutex_init (mutex, attr)
  - pthread_mutex_destroy (mutex)
  - pthread_mutexattr_init (attr)
  - pthread_mutexattr_destroy (attr)

- Condition variables: Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values.
  Condition variable:

  - pthread_cond_init (condition, attr)
  - pthread_cond_destroy (condition)
  - pthread_condattr_init (attr)
  - pthread_condattr_destroy (attr)

- Synchronization: Routines that manage read / write locks and barriers.

  - pthread_mutex_lock (mutex)
  - pthread_mutex_trylock (mutex)
  - pthread_mutex_unlock (mutex)
  - pthread_cond_wait (condition, mutex)
  - pthread_cond_signal (condition)

1.1.3 Critical Section

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 multi-threaded 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.

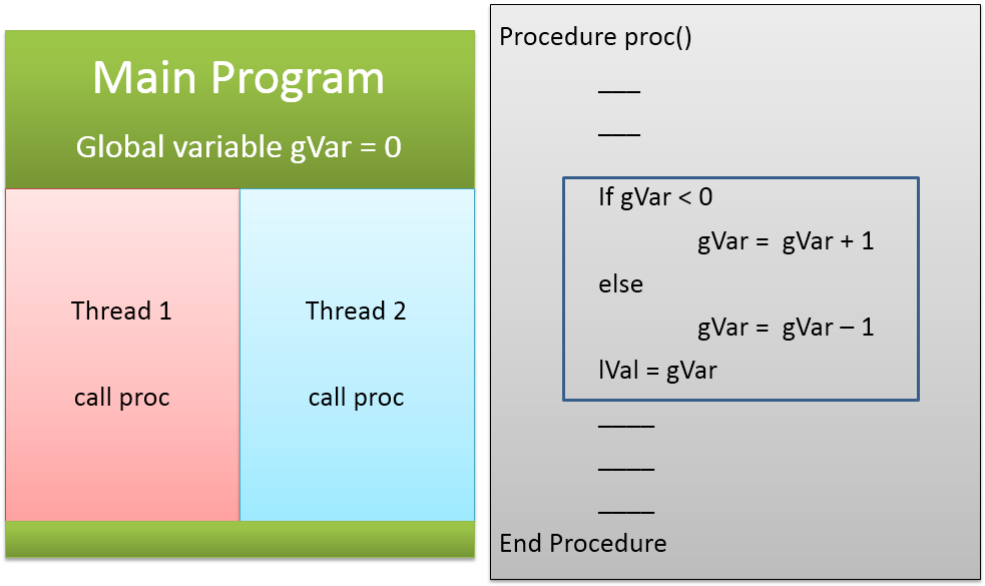

Figure 1.4: Critical Section

In the figure shown above, 'gVar' is a global variable of the procedure proc(). Procedure is associated with two threads i.e. Thread1 and Thread2. Now, when execution starts both threads may access the global variable 'gVar' concurrently, hence 'gVar' is a Critical Section for the given procedure.

1.1.4 Race Condition

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 [9].

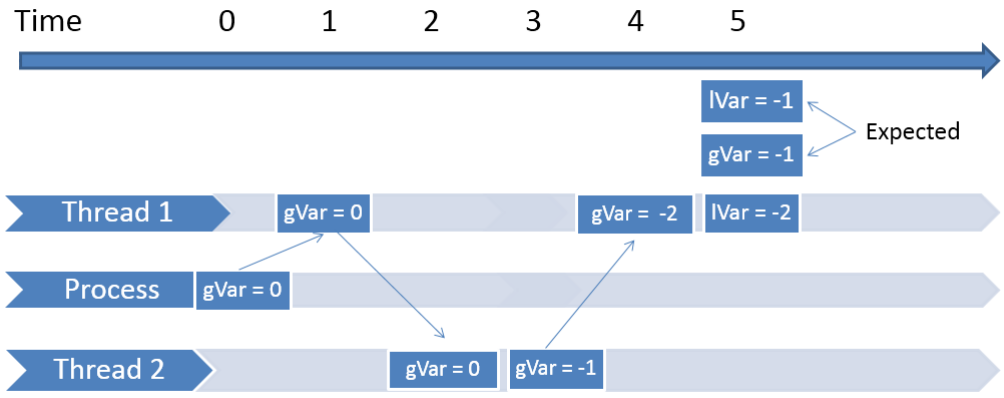

Figure 1.5: Race Condition

By considering the code snipest shown in fig 1.1: Critical Section, one possible case of execution is shown above in the figure of Race Condition.

1.1.5 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 smallscale multiprocessing systems - in multithreaded environments and multiprocessor computers - and in distributed computers consisting of thousands of units [9].

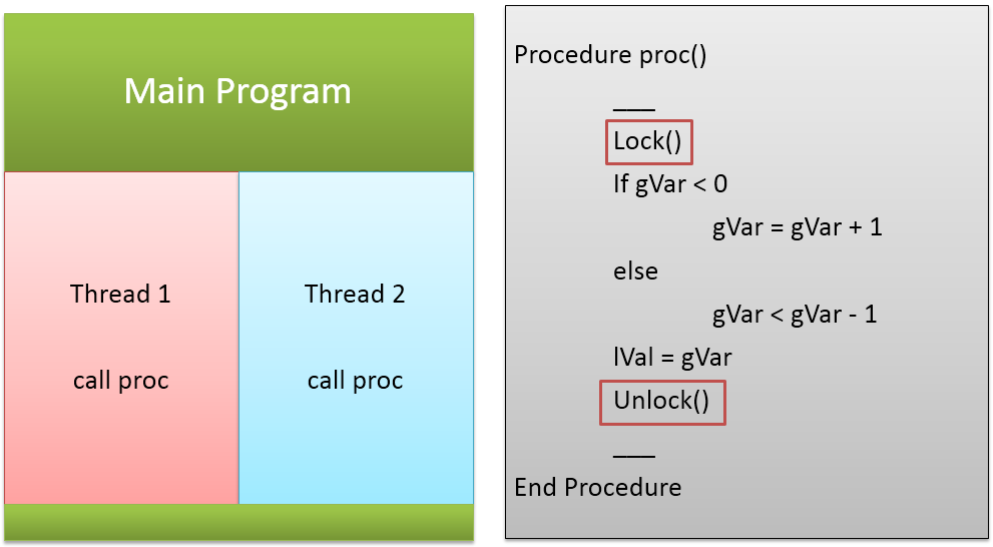

Figure 1.6: Synchronization

To avoid the race condition, synchronization mechanism is used in multi threaded programming.Synchronization mechanism may depend on the application logic, hence it differe in many cases.

1.2 Need

When large multithreaded programs are written, its difficult to keep a track of the critical sections in it. This may inturn lead to the occurence of unexpected results or may lead to synchronization bugs. Our project aims to find out these critical sections and introduce a locking mechanism to avoid race conditions in the program[9].

And current version of GCC does not provide or support any feature to find out the critical sections or synchronization bugs in the multitheaded programming.

1.3 Motivation

Motivation behind Critical Section Detection tool is contribution to open source technology, because now a days there are many technologies which are there already implemented based on GCC or Open Source technologies but each of them was developed considering different parameters in front of them.

Our system is motivated not only considering detecting critical sections but different parameter which are mandatory now a day, like initially we are checking for error free C source code and after that tokenizing and parsing the same source code.

So it seems that we need to study all the phases compiler in details and as fat our knowledge about GCC concerns, GCC doesn't provide this featue of detecting critical sections for given source code.

GCC Extension for detecting Critical Sections in a Multi threaded Environment is motivated with respect to many parameter under one basic idea, considering one scenario where one peraon have written a lorge multithreaded code say 1 KLOC and another person tries to work on the same code but unfortunately there are some synchronization bugs, so in such a case it is so difficult to new person to browse that code and detect the suspected critical section or race condiiton. Hence if the tool that we have developed is used by that new person, would become very easy to detect suspected critical section.

1.4 Objective

To develop a GCC Extension for detecting Critical Sections and providing locking and unlocking mechanism or synchronization mechanism which may lead to race Condition or Synchronization bugs in a Multithreaded Environment.

The main objective of the system is, the tool should automatically detect the suspected critical sections which may lead to race conditions or synchronization bugs in a multitheaded program.

Chapter 2

Literature Survey

There are two ways to detect critical sections as follows:
1) Static analysis
2) Dynamic analysis

There are very few tools available which detect critical section in multi-threaded environment. These tools have used one of these two methods.

2.1 Static Analysis

This methods for race detection examine only the program text, and assume that all execution paths through the program are possible. Under this assumption, determining whether sections of code may execute con-currently (or in some other order) requires examining only the explicit synchronization in the program (this assumption would always be correct if the program contained no conditionally executed code). Static analysis therefore examines how the programs synchronization might allow such potential orderings. Using this ordering information, and a conservative analysis of which shared variables may be read and written in each section of code, data races and general races can be detected.

First, some methods traverse the space of all possible states that the program may enter. This state space can either be constructed explicitly, by building a graph, or implicitly, by constructing a representation of the state space (such as a formal language or a petri-net). In the general case, these methods have exponential time complexity, and in some cases, exponential space complexity as well.

Second, other static analysis methods perform a data-flow analysis of the program to discover potential event orderings. These methods have polynomial time and space complexity, but are less accurate than the state-space methods, sometimes reporting races that the program could never exhibit (and that the state-space meth-

ods would never report).

Static analysis has also been used to complement dynamic methods. Static analysis can sometimes rule out the possibility of races between some sections of the program, precluding the need for tracing these program sections for dynamic analysis. Static analysis can also compute input data that might manifest a race, allowing dynamic analysis to attempt to verify the existence of that race.

Advantages:
1) Easy to implement
2) No need to analysis dynamic nature of a program.
Disadvantages:
1) Static analysis doesn't predict perfectly nature of a program.
2) It requires to use diferent API's of different libraries.

2.2 Dynamic Analysis

Unlike static analysis, dynamic analysis detects the race conditions exhibited by a particular execution of the program. Dynamic analysis is especially useful for debugging since precise information about manifestations of particular bugs is available. Below the authors[3] outline previous work on dynamic data race and general race detection. They first discuss the similarities and differences of the various methods and then present the unique details of each. Finally, they present an example that focuses on the simple type of analysis that is common to all methods. They show why this analysis can report race artifacts, and discuss other aspects of previous work.

There are many tools exist which are used to determining possibility of critical section in multi-threaded program.These tools are as followed:
1) Valgrind
2) gdb

These tools are explained below in detailed:

2.2.1 Valgrind

It is a GPL licensed programming tool for memory debugging, memory leak detection, and profiling.Valgrind was originally designed to be a free memory debugging tool for Linux on x86, but has since evolved to become a generic framework for creating dynamic analysis tools such as checkers and profilers. It is used by a number of Linux-based projects. There are multiple tools included with Valgrind (and several external ones). These are as followed.

Memcheck

It is default and most used tool in valgrind,It is used for detecting memory leakage. The memcheck is mostly used for:
1) Use of uninitialized memory
2) Reading/writing memory after it has been freed
3) Reading/writing off the end of malloc'd blocks
4) Memory leaks

Helgrind

Helgrind is a Valgrind tool for detecting synchronisation errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives. The main abstractions in POSIX pthreads are: a set of threads sharing a common address space, thread creation, thread joining, thread exit, mutexes (locks), condition variables (inter-thread event notifications), reader-writer locks, spinlocks, semaphores and barriers. Helgrind can detect three classes of errors, which are discussed in detail in the next three sections:
1) Misuses of the POSIX pthreads API.
2) Potential deadlocks arising from lock ordering problems.
3) Data races – accessing memory without adequate locking or synchronisation.
Problems like these often result in unreproducible, timing-dependent crashes, deadlocks and other misbehaviour, and can be difficult to find by other means.

Helgrind is aware of all the pthread abstractions and tracks their effects as accurately as it can. On x86 and amd64 platforms, it understands and partially handles implicit locking arising from the use of the LOCK instruction prefix. On PowerPC/POWER and ARM platforms, it partially handles implicit locking arising from load-linked and store-conditional instruction pairs. Helgrind works best when your application uses only the POSIX pthreads API.

Following those is a section containing hints and tips on how to get the best out of Helgrind. Then there is a summary of command-line options. Finally, there is a brief summary of areas in which Helgrind could be improved.

Detected errors: Misuses of the POSIX pthreads API. Helgrind intercepts calls to many POSIX pthreads functions, and is therefore able to report on various common problems. Although these are unglamourous errors, their presence can lead to undefined program behaviour and hard-to-find bugs later on. The detected errors are:
1) unlocking an invalid mutex
2) unlocking a not-locked mutex
3) unlocking a mutex held by a different thread
4) destroying an invalid or a locked mutex
5) recursively locking a non-recursive mutex
6) deallocation of memory that contains a locked mutex
7) passing mutex arguments to functions expecting reader-writer lock arguments, and vice versa
8) when a POSIX pthread function fails with an error code that must be handled
9) when a thread exits whilst still holding locked locks
10) calling pthread_cond_wait with a not-locked mutex, an invalid mutex, or one locked by a different thread
11) inconsistent bindings between condition variables and their associated mutexes
12) invalid or duplicate initialisation of a pthread barrier
13) initialisation of a pthread barrier on which threads are still waiting
14) destruction of a pthread barrier object which was never initialised, or on which threads are still waiting
15) waiting on an uninitialised pthread barrier
16) for all of the pthreads functions that Helgrind intercepts, an error is reported, along with a stack trace, if the system threading library routine returns an error code, even if Helgrind itself detected no error
Checks pertaining to the validity of mutexes are generally also performed for reader-writer locks. Various kinds of this-can't-possibly-happen events are also reported. These usually indicate bugs in the system threading library.

2.3 Automatic Critical Section Discovery Using Memory Usage Patterns

2.3.1 Summary

The authors[2] have introduced a new heuristic to infer critical sections using the tempo- ral and spatial locality of critical sections and provide empirical results showing that the heuristic can infer critical sections in shared memory programs. A programmer can use the reported critical sections to inform his addition of locks into the program. They present evidence for a new way to discover a type of parallel programming bug called an atomicity violation, which occurs when a programmer expects that a region of code will be executed without interference from other simultaneously executing code regionsthat is it must exe- cute atomically. While several tools can analyze a programs memory usage, paper choose to use Pintool. This tool allows it to run on any program using X86 assembly language[2].

2.3.2 Advantages

The Tool can be incorporated as advisor program that informs the programmer of code regions that might need to be critical sections and to incorporate the results into a second program that automatically inserts critical sections into the code.

Around 75% to 80% of static real critical sections were fully covered, fully covered with overlap or partially covered with the best set of thresholds for all benchmarks with the majority being fully covered. For dynamic real critical section coverage, this number jumps to around 90%.

2.3.3 Disadvantages

The authors analysis incorrectly included instructions that were inside pthread library functions, because Pin was unable to completely identify both the dynamic beginning and end of all pthread library functions.

Analysis was limited by the fact that we only had enough computing power to analyze heap memory references for inferring objects.

2.4 Automatic Lock Insertion in Concurrent Programs

2.4.1 Summary

The authors[4] consider the problem of lock insertion to enforce critical sections required to fix bugs like atomicity violations. This can be accomplished in a trivial manner by sim- ply encapsulating the desired regions of code within lock/unlock statements. However, enforcing critical sections is often not the sole criterion to be satisfied during lock insertion. Indeed, adding mutexes may introduce new deadlocks. Thus a key goal is to guarantee deadlock-free lock insertion, i.e., no new deadlocks are introduced. The authors[3] have designed an algorithm which concentrates on the correctness and per- formance. These contraints may be of conflicting nature but are indeed key components that is needed to ensure a deadlock-free program, while keeping the critical sections as small as possible. Enforcement of mutually atomicity, deadlock freedom, optimality. The authors[4] have concentrated on the concepts of nested locks, which avoids global analysis and ensures large real-life problems[4].

2.4.2 Advantages

The technique presented by the authors have considered the case where prior locks exist, and how to handle them, since removing existing locks and putting them again in a pre-defined order presents many practical obstacles.

It analyzes only a part of program which has the critical section bugs and not the whole program.

Side benefit is that it ensures scalability of the analysis.

2.5 Minimum Lock Assignment: A Method for Exploiting Concurrency among Critical Sections

2.5.1 Summary

A naive lock assignment approach associates one lock to each shared memory location, and the lock set of a critical section is the set of locks assigned to memory locations it accesses. This approach, however, may use more locksthan necessary, and introduce excessive over- head on lock acquisition and release. To control the locking overhead, the authors[5] would use the minimum number of locks which is

necessary to preserve the mutual exclusion and fully exploit theconcurrency between critical sections.

Minimum Lock Assignment:Given a multithreaded program with a set of critical sections, find the minimum number of distinct locks that are needed for controlling the critical sec- tions such that (a) Two critical sections are assigned disjoint sets of locks if they are concurrent and they do not access any common location, or if they access a common location then none of them writes to the common location. (b) Two critical sections are assigned at least one common lock if they are concurrent and they access some common location and at least one of them writes to the common location.

Chapter 3

Proposed Work

3.1 Problem Statement

The compiler is made to identify the critical section in a multi-threaded program for the synchronization bugs, which currently is not a feature in gcc. Also, the compiler should automatically take care of the critical sections by introducing helper statements (comments) for Lock and Unlock function calls in a multi-threaded program without involvement of the programmers.

3.2 Scope

The project has been split up in two modules.

Firstly, the project aims to nd out critical sections in a program which may cause race conditions. The reasons for Race conditions occurence are many. For example, simaltaneuos writing on shared variables(like global, extern, volatile), les, pipes. We shall be nding the critical sections of these.

There are situations where prior locks exist and they need to be taken care of. The approach generally used is either a lock free program, or the existing locks are removed and then put back. This causes lot of processing overhead, and hence is ineffecient.

We also intended to contentrate only on the critical sections of the code since doing a point-to analysis of the whole program proves out the be process intensive. In some cases wrong Lock and Unlock may the reason for Race Condition so in such cases detecting CS and inserting Lock and Unlock on proper location.

Chapter 4

Research Methodology

4.1 Steps to acquire and process

4.1.1 Parser Tokenize (Flex)

This stage helps us token the program for further processing.

4.1.2 Grammar Generation of tables (Yacc)

Once the tokens are generated we pass the output to this phase, where it goes through our grammar, which helps find out different threads, functions associated with them, parameters, shared memory being used in them. We generate the following tables based on the grammar:-

- Global variable (var_name, location)
- User-defined Functions (func_name, location)
- Parameter table (func_name, all_parameters)
- Thread-Functions table (thread_handle, func_name)
- Function local variable table (func_name, var_name)
- Variables' log table (var_name, func_name, location)
- Critical Section table (var_name, location)

4.1.3 Analysis

Once the tables are generated, we check if any global variables are being used in multiple functions, and whether they are handled using sem_wait() and sem_post() function calls.

Chapter 5

Project Design

5.1 Software Requirement Specification

5.1.1 Hardware Requirements

- 2.4GHz Celeron Processor
- 256MB Memory
- 10GB Disk Space

5.1.2 Software Requirements

- [Operating System]: Linux Based Operating System
- [Build-essential]: Build-essential is required to build Debian packages, starting with dpkg (>= 1.14.18)
- [Necessary packages for building GCC]: `http://gcc.gnu.org/install/prerequisites.html`

5.1.3 Technologies Used

- C, lex, yacc.

5.2 U.M.L. Diagrams

5.2.1 Use Case

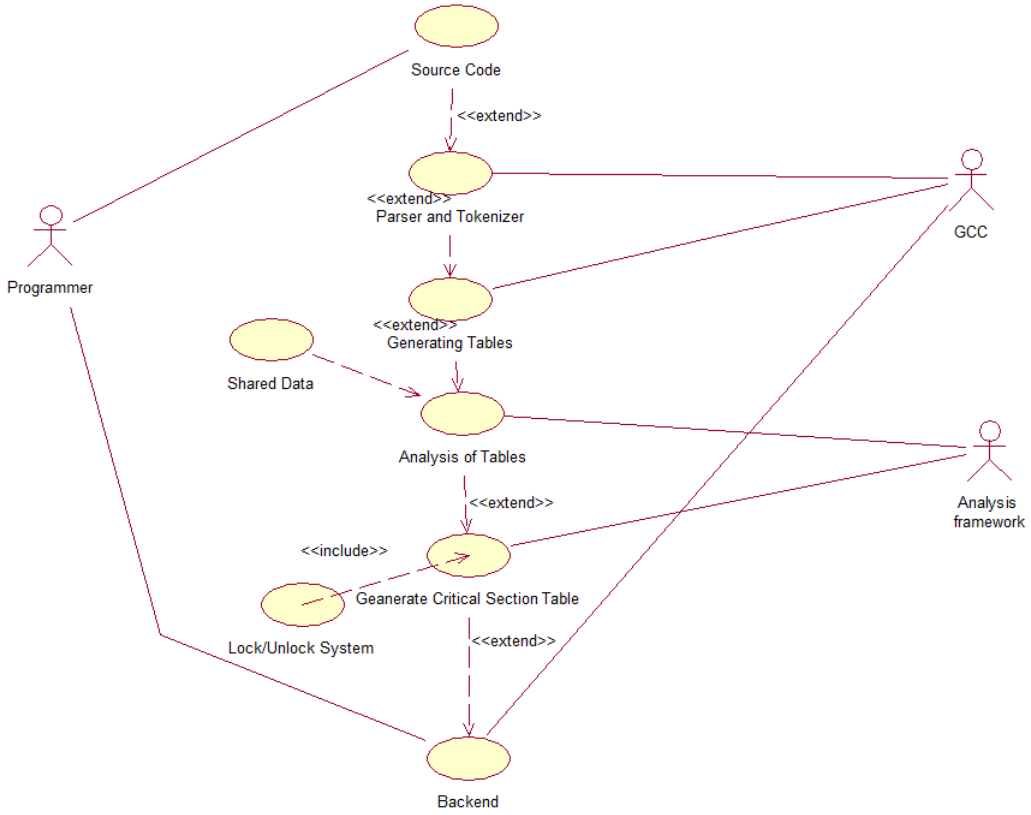

Figure 5.1: Use case diagram

From use case point of view our system contains one actor as user or programmer, in which programmer has to specify a C source code (Source code should be error free and Multithreaded for better results). And on the other hand GCC does the parsing and tokenization with the help of patterns and grammar that we have written, Analysis framework does the detection of critical sections in a source code provided by the Programmer as a input.

5.2.2 State Chart

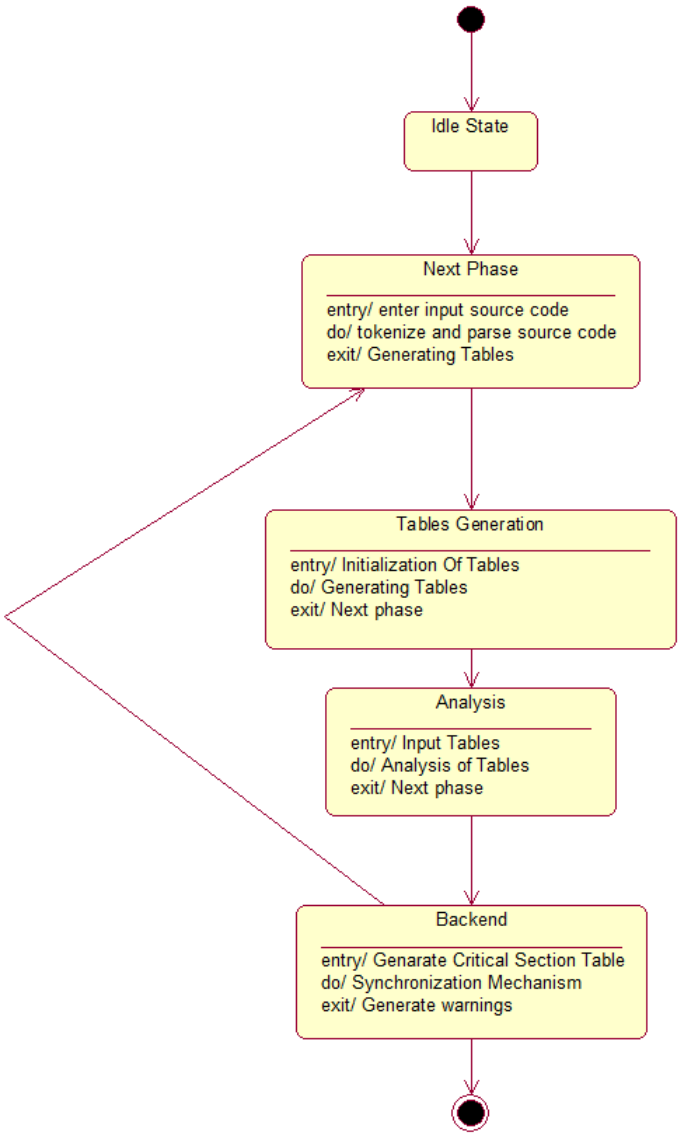

Figure 5.2: State chart diagram

Control Flow of the Critical Section Detection System. Above figure shows the state transisions of our system, this shows that how actually the system is going to transfer from eact step to next step after providing certain input. There are certain steps in each state. Flow of the Critical Section Detection System.

5.2.3 Sequence Diagram

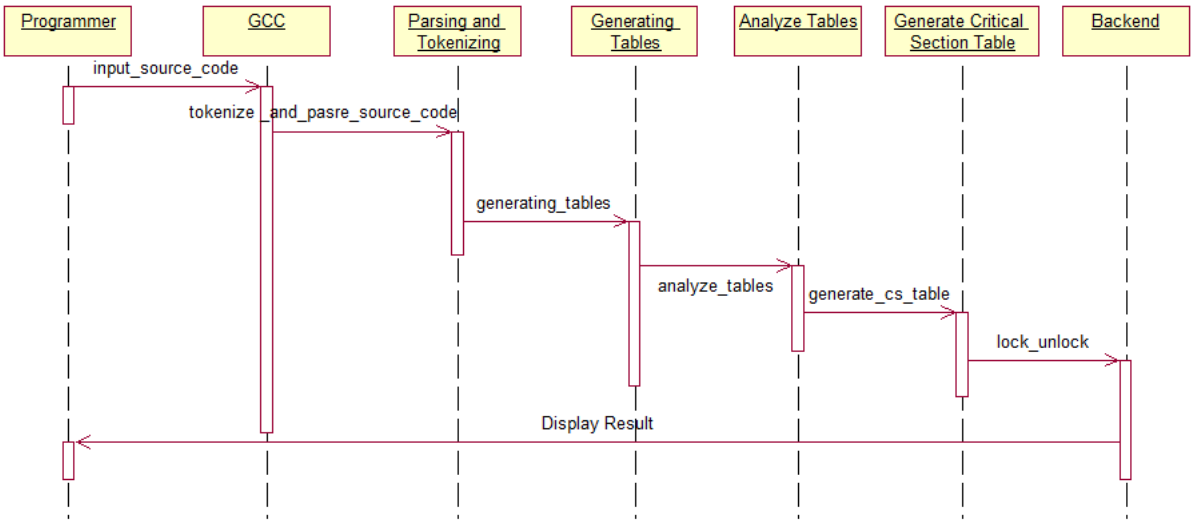

Figure 5.3: Sequence Diagram

Shows the sequences of operations from starting to last one. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario i.e Programmer, GCC, Analysis Framework etc. and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Scenario could be whole sequence of operations till the end result.

5.2.4 Collaboration Diagram

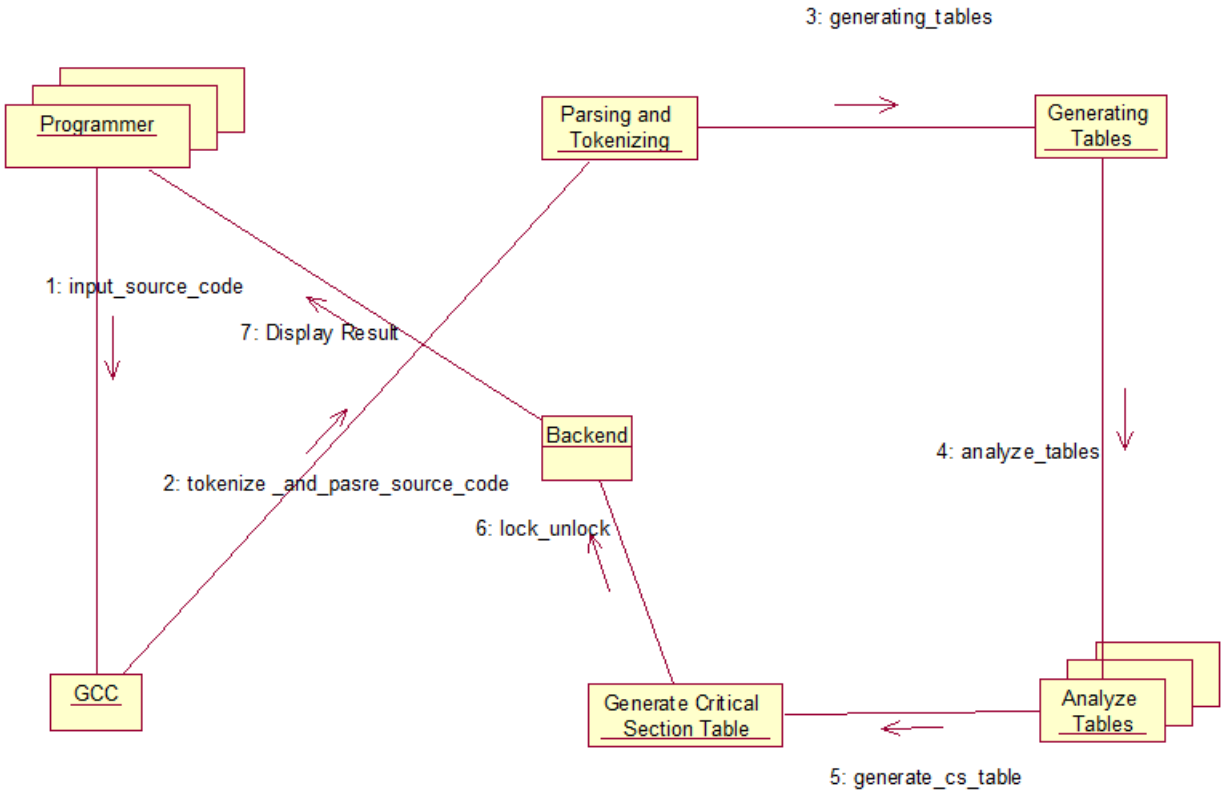

Figure 5.4: Collaboration Diagram

From Collaboarative perspective of our system, it covers the same scenario as shown in Sequence diagram.Indeed the collaborative view of the system is generated by using sequence diagram.

5.2.5 Activity Diagram

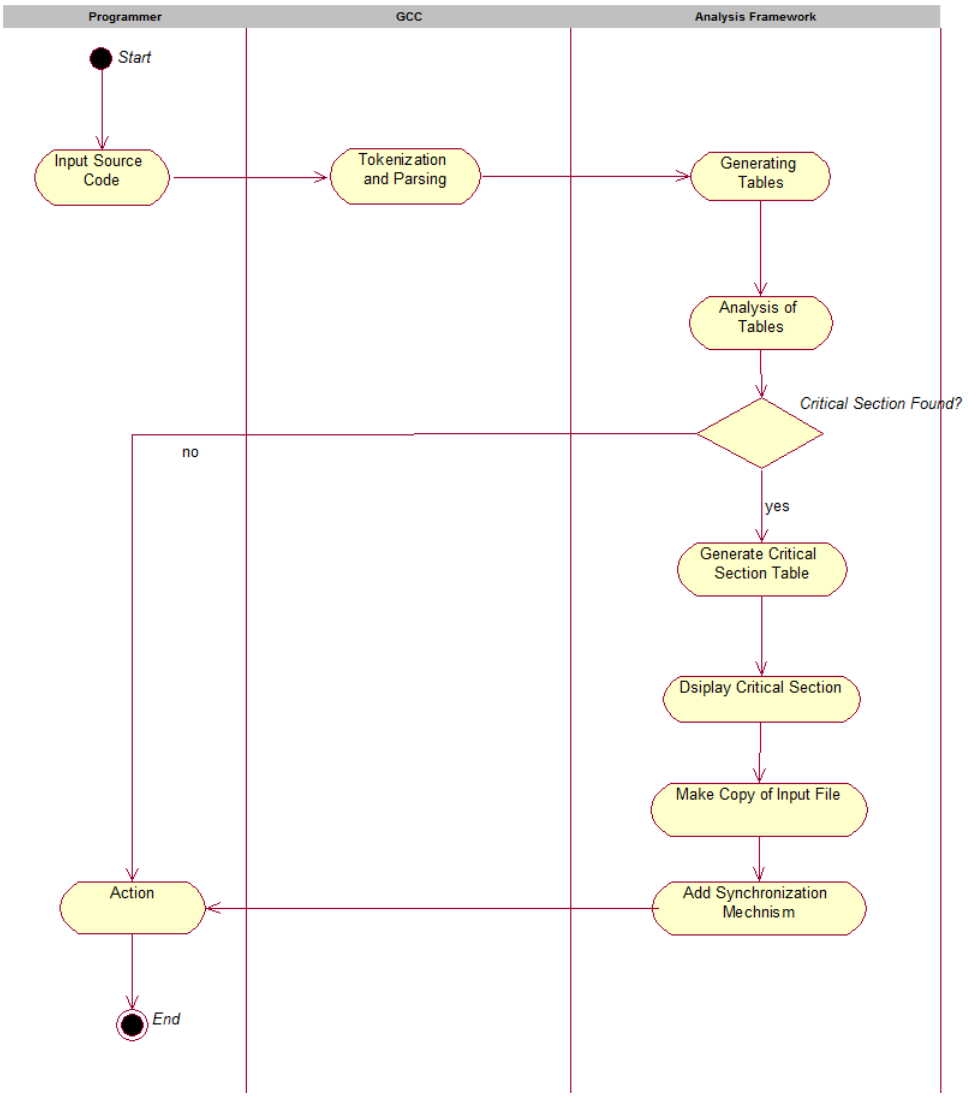

Figure 5.5: Activity Diagram

From programmers prespective activity diagram shows which are the diffrent activities performed by Programmer, GCC and the Analysis Framework.

5.2.6 Component Diagram

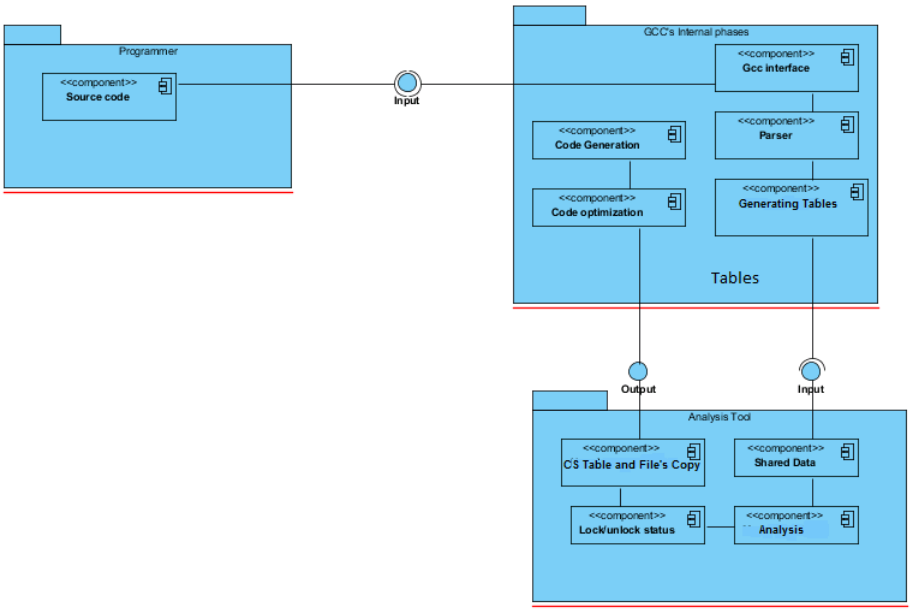

Figure 5.6: Component Diagram

From Component point of view our system having different components used in system like system is Table Generator which generates the tables like Global Symbol's Table, Local Symbol's Table, Log Table, Semaphore Table, Function Table and Thread Table etc.

5.2.7 Deployment Diagram

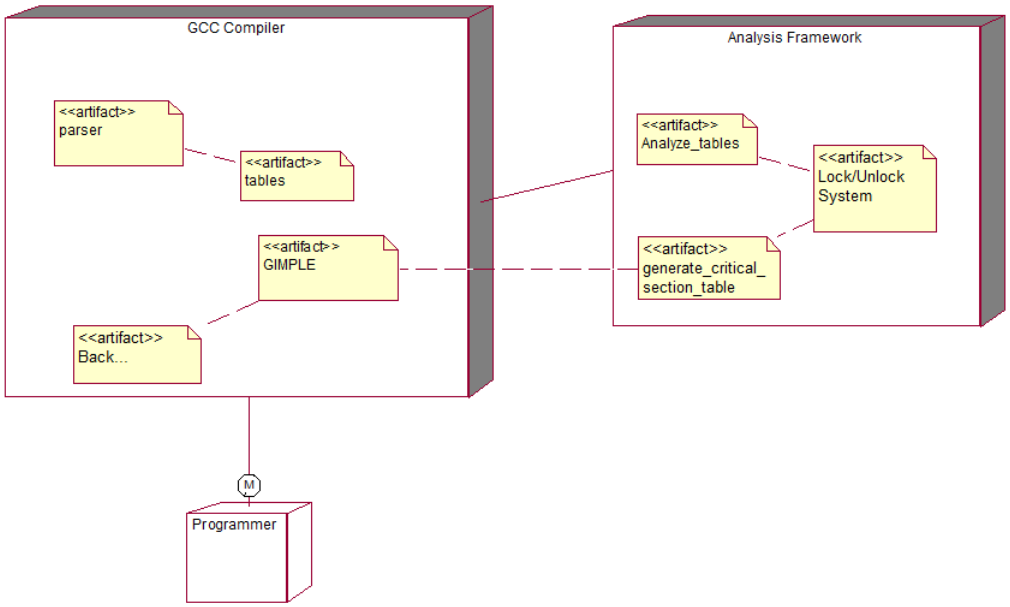

Figure 5.7: Deployment Diagram

From deployment point of view our system contains following hardware and software component which are required for system execution are like GCC Compiler for compiling the source code, lex and yacc for Tokenization and Parsiing and the analysis frmaework for detecting the critical sections and adding synchronization mechanism.

Chapter 6

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.

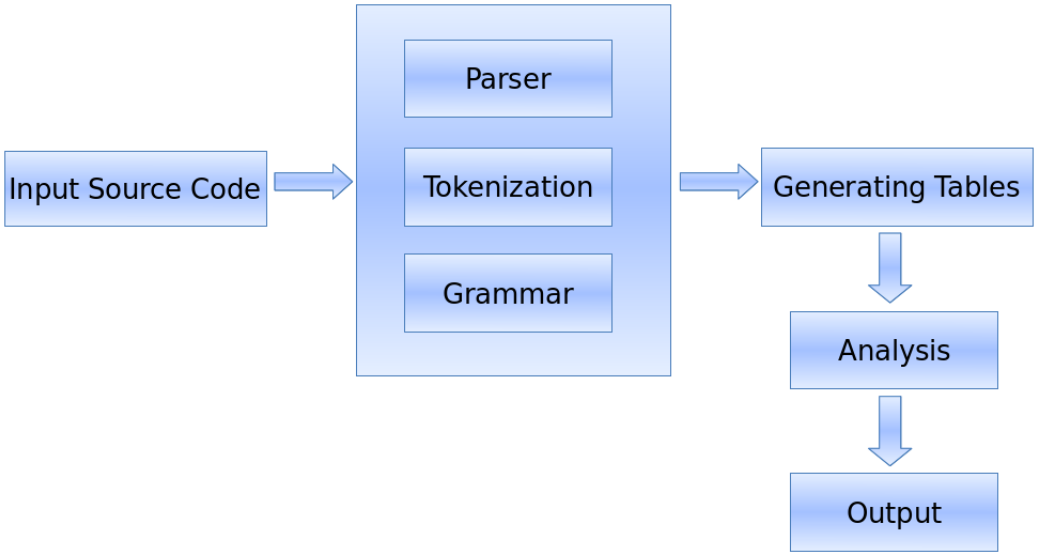

Figure 6.1: Architectural Diagram

6.1 Architecture

6.1.1 Input Source Files

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.

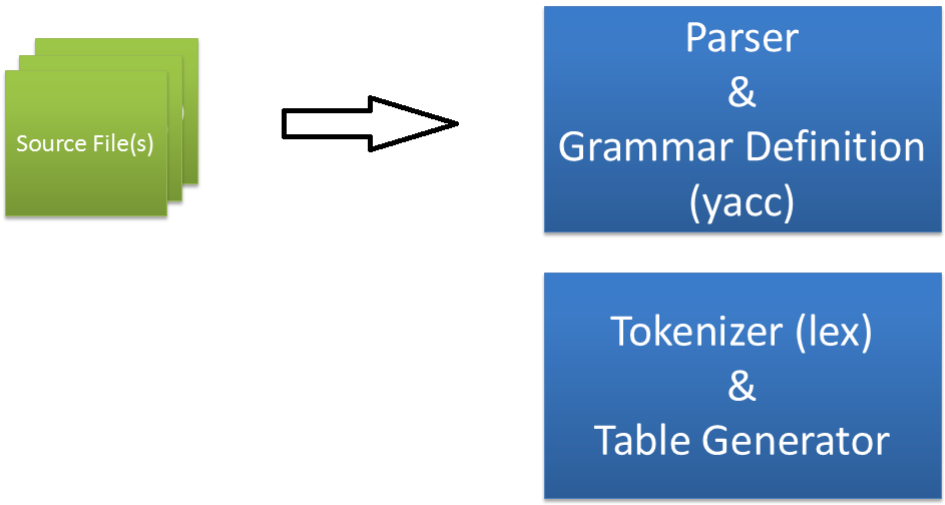

Figure 6.2: Source File(s)

6.1.2 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

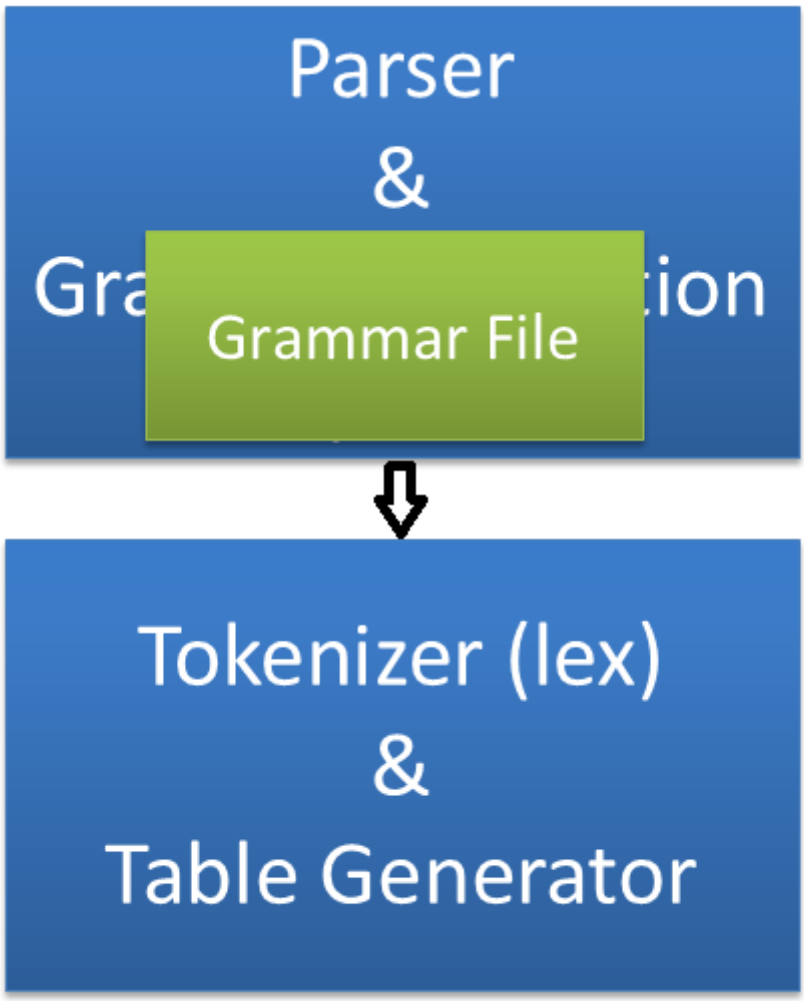

Figure 6.3: Parser, Tokenizer, Grammar

6.1.3 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 usersupplied 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
- Shared Memory 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, first_location, last_location, function_name)

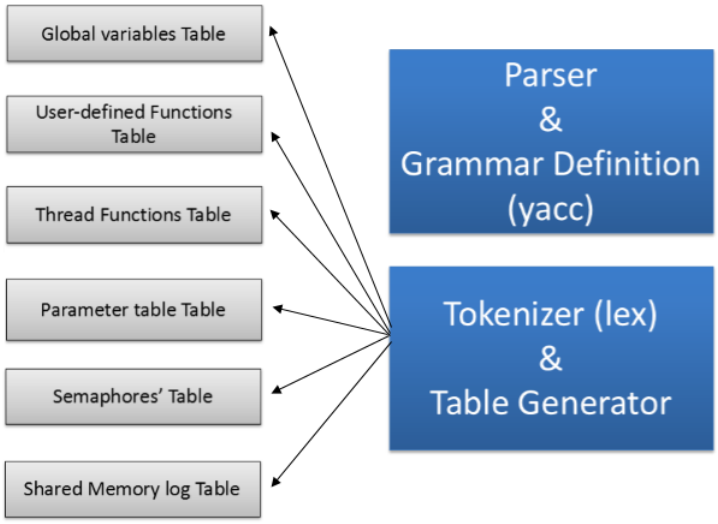

Figure 6.4: Generated tables

6.1.4 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.

6.1.5 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(),

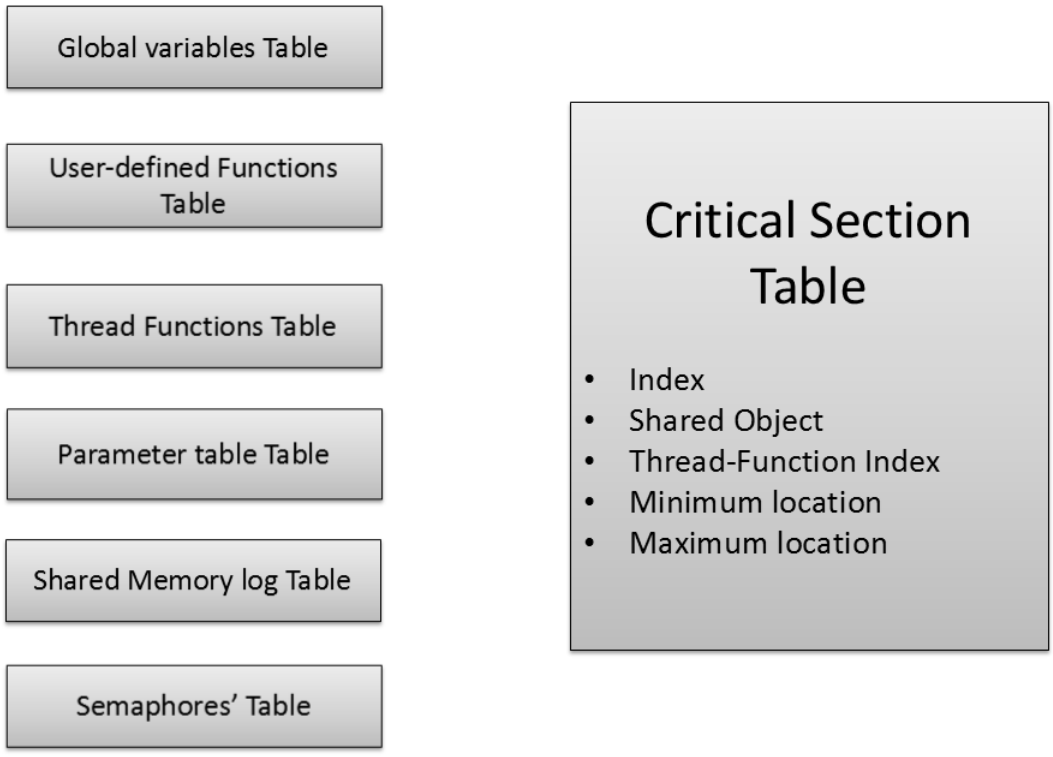

Figure 6.5: Critical Section Table

recompile the code to produce bug free programs.

As par the implementation part is concern we required certain modules to be integrate to form the whole system or tool. Form the implementation point of view this tool is divided into following modules .

- Input Source File
- Tokenization and Parsing
- Generating Tables
- Analysis
- Output

Module of the Tokenization and Parsing

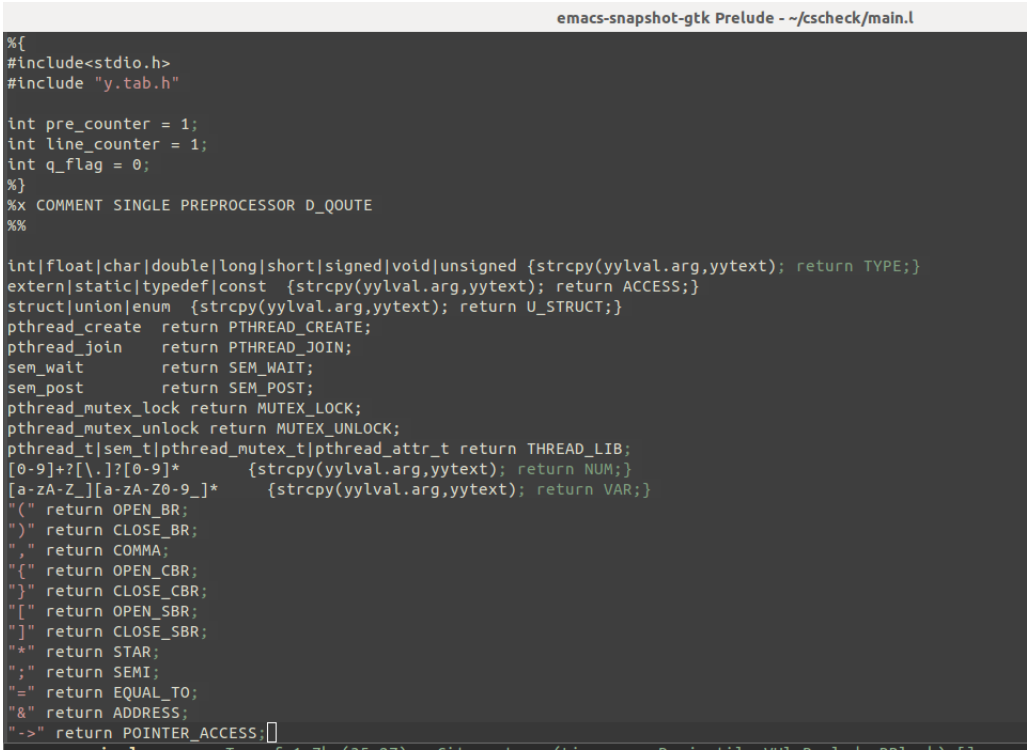

Figure 6.6: Tokens

Figure 6.7: Tokens

6.2 Generating Tables

emacs-snapshot-gtk Prelude - ~/cscheck/common.h

```
/*****************************************************************************
 *      Header contains structures for handling critical section and declarations       *
 *      for genrating tables.                                                *
*****************************************************************************/


#include <stdio.h>

#define SIZE_VAR 50
#define SIZE_FUNC 30
#define NUM_PAR 30
#define SIZE_TYPE 20
#define SIZE_TABLE 100
#define INFINITY 65535

/* Structure contains information about all global variables. */
typedef struct global_symbol
{
    int index;
    char access[SIZE_VAR];
    char sym_name[SIZE_VAR];
    char type[SIZE_TYPE];
    int line_number;
}global_symbol;

/* Parameter structure has information about the parameters stated
   in user-defined functions. */
typedef struct parameter
{
    char type[SIZE_TYPE];
    char par_name[SIZE_VAR];
}parameter;
```

Figure 6.8: Table's structure

emacs-snapshot-gtk Prelude - ~/cscheck/common.h

```
/* symbol_table structure has information about all the local variables
   used in given input (C program). */
typedef struct symbol_table
{
    int index;
    int func_index;
    char sym_name[SIZE_VAR];
    char access[SIZE_VAR];
    char type[SIZE_TYPE];
    int line_number;
}symbol_table;

/* func_def structure has information about user-defined functions */
typedef struct func_def
{
    int index;
    int line_number;
    char func_name[SIZE_FUNC];
    char return_type[SIZE_TYPE];
    int no_of_parameter;
    int no_of_symbols;
}func_def;

/* thread_info structure has information about threads created in
   C program. */
typedef struct thread_info
{
    int index,line_number, pthread_join;
    char thread_obj[SIZE_TYPE];
    char func_name[SIZE_TYPE];
    int func_index;
    char thread_attr[SIZE_TYPE];
    char func_arg[SIZE_TYPE];
    char parent_thread[SIZE_TYPE];
}thread_info;
```

Figure 6.9: Table's structure

6.3 Output

If the input program is one of the following kind, then it can be said to be a correct program:

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

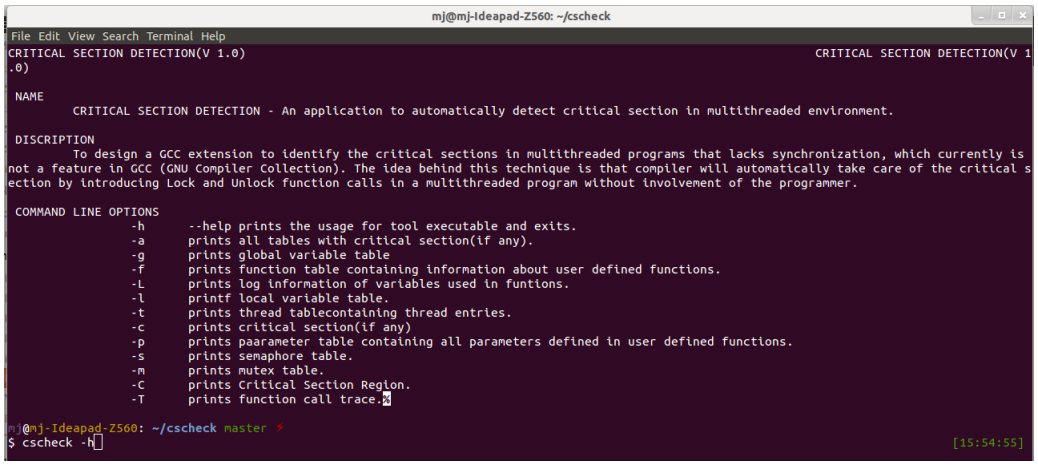

Figure 6.10: All functionality available by different command line flags

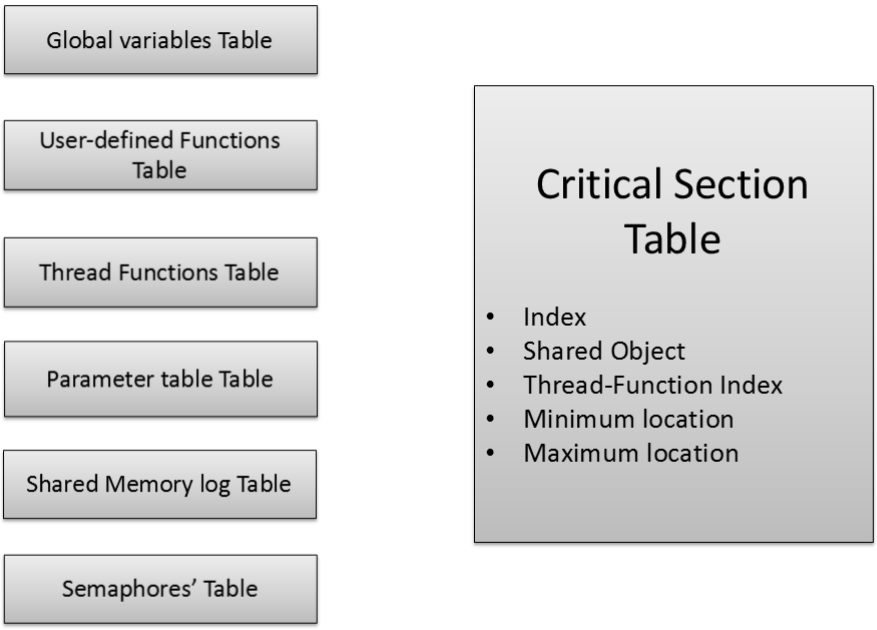

Figure 6.11: Contents of Critical Section Table

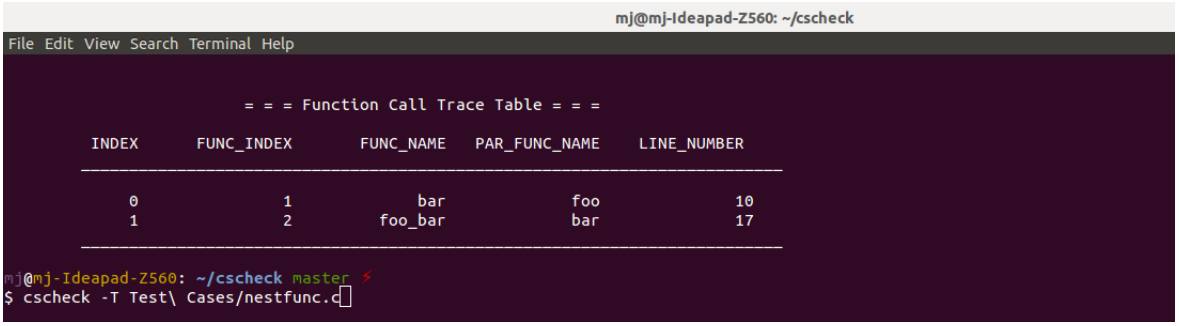

Figure 6.12: Call trace for a thread function

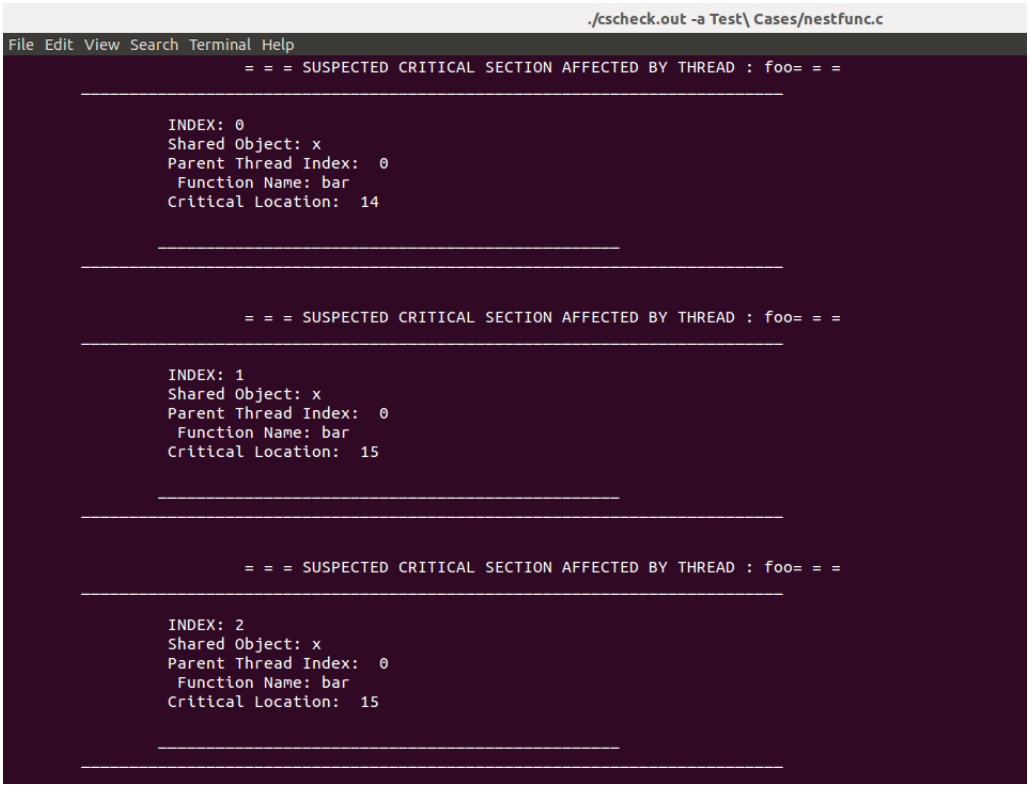

Figure 6.13: Detected Critical Section

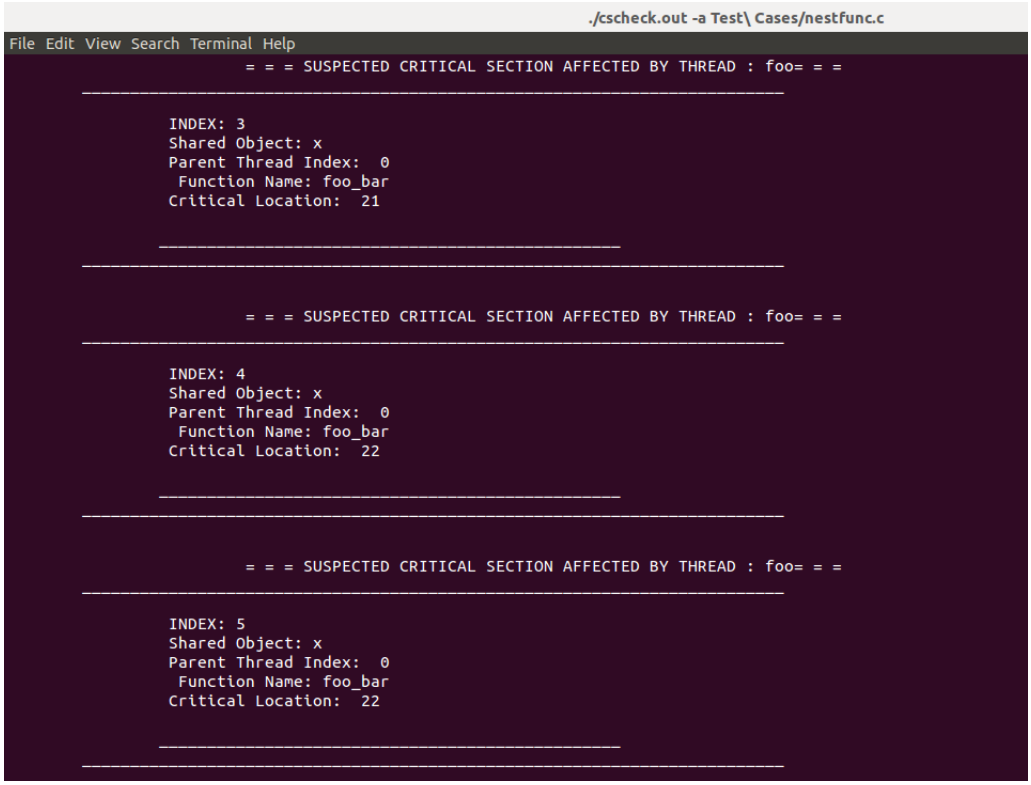

Figure 6.14: Suspected Critical Section

6.4 Manual Testing

Table 6.1: Manual Testing

TestCase ID	Objective	Case	Procedure	Expected Results	Actual Results	Pass /Fail
Check Tools	To check requirements of different tools.	1.	a) Open Terminal and enter gcc.	GCC should be available.	Same as Expected	Pass
			b) Open terminal and enter lex	lex should be available	Same as Expected	Pass
			c) Open terminal and enter yacc	yacc should be available	Same as Expected	Pass
Program Structure	To check nature of program.	1.	a) Program should be errorless.	Compilation of program should be successful	Same as Expected	Pass
			b) Program should contains POSIX threads.	Program should create new threads using POSIX API.	Same as Expected	Pass
Tokenization	To create different tokens as per constraints.	1.	a) Datatypes	Different tokens should be generated according to datatypes.	Same as Expected	Pass

Continued on next page

Table 6.1 – *Continued from previous page*

TestCase ID	Objective	Case	Procedure	Expected Results	Actual Results	Pass /Fail
		2.	a) Function signature should be tokenized.	Fuction name,number of argument and return type should be tokenized	Same as Expected	Pass
		3.	b) POSIX thread's signature should be tokenized.	Thread object,Function name which is executed by thread,number of argument and return type should be tokenized	Same as Expected	Pass
		4.	c) Semaphore signature should be tokenized.	Semaphore name,different semaphore signals should be tokenized	Same as Expected	Pass
		5.	d) mutex signature should be tokenized.	Mutex name,different mutex signals should be tokenized	Same as Expected	Pass
Parsing	Generate different grammer as per tokens.	1.	a) Grammer for variable datatypes.	Valid grammer should be generated for datatype.	Same as Expected	Pass

Continued on next page

Table 6.1 – *Continued from previous page*

TestCase ID	Objective	Case	Procedure	Expected Results	Actual Results	Pass /Fail
		2.	b) Grammer for function signature.	Valid grammer should be generated for functions.	Same as Expected	Pass
		3.	c) Grammer for POSIX thread's signature.	Valid grammer should be generated for threads.	Same as Expected	Pass
		4.	d) Grammer for API of Semaphore.	Valid grammer should be generated for semaphores	Same as Expected	Pass
		5.	e) Grammer for Mutex.	Valid grammer should be generated for mutex	Same as Expected	Pass
Table Generation.	Different table are genereted according grammers.	1.	a) Table for variable including global and local.	Valid tables should be generated for datatype including local and global.	Same as Expected	Pass
		2.	b) Table for function signature.	Valid table should be generated for functions.	Same as Expected	Pass
		3.	c) Table for POSIX thread's signature.	Valid table should be generated for threads.	Same as Expected	Pass

Continued on next page

Table 6.1 – *Continued from previous page*

TestCase ID	Objective	Case	Procedure	Expected Results	Actual Results	Pass /Fail
		4.	d) Table for API of Semaphore.	Valid table should be generated for semaphores	Same as Expected	Pass
		5.	e) Table for shared memory (Log Table)	Valid table should be generated for shared memory	Same as Expected	Pass
		6.	f) Table for Mutex.	Valid table should be generated for mutex	Same as Expected	Pass
Analysis of critical section.	Critical sections blocks are detected	1.	a) Locking and unlocking mechanism.	Check wheter locking and unlocking mechanism is exist for Critical Section.	Same as Expected	Pass
		2.	b) Critical Section detection.	If critical section isn't synchronized then add it.	Same as Expected	Pass

Chapter 7

Scheduling

7.1 Proposed Modules

- Parse, Tokenize
- Tables Generation
- Analysis
- Testing

7.2 Scheduling

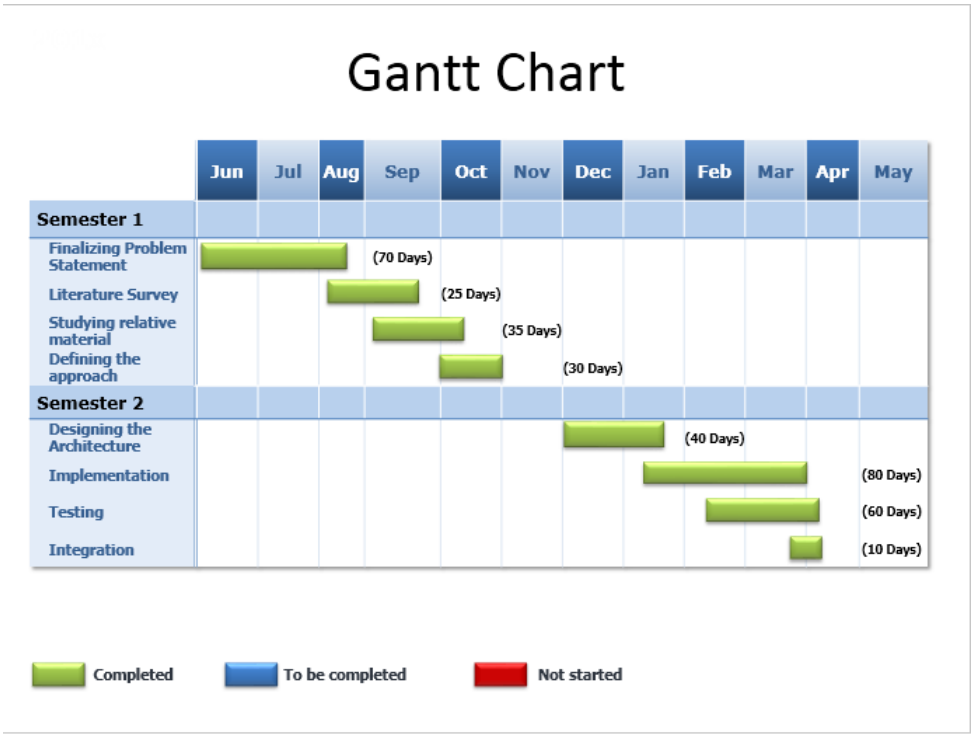

Figure 7.1: Gantt Chart

Chapter 8

Conclusion and Future Scope

8.1 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.

8.2 Future Scope

Although this concept if pretty much full proof, the capabilities can extended further.
As of now, we are only supporting POSIX thread library and semaphore locking mechanism, this project can be extended for other thread libraries and locking mechanism.

- OpenThreads
- Boost C++ libraries http://www.boost.org/
- OpenMP http://openmp.org/wp/
- Intel Cilk Plus http://software.intel.com/en-us/intel-cilk-plus
- ZThreads http://zthread.sourceforge.net/

Since this approach is generic, we can port the same for compilers of other languages. Since critical sections are one of the most important factors of concurrent programming, this port would help programmers of other domain incorporate it with their development and benefit from it.

For now we have only concentrated on shared memory in User space. A possible future enhancement could be that we can tackle shared memory in the kernel space, (for example: kernel level pipes) or handle files as well.

A very difficult (but a possible) concept of adding synchronization mechanisms automatically could be done. However, many complications are associated with it as locking mechanism depends on the application logic, which is practically impossible to identify programmatically.

References

[1] ”Automatic Critical Section Discovery using Memory Usage Patterns”; Lisa Marie Stechschulte, Department of Electrical and Computer Engineering, University of Maryland, 2012

[2] ”Race Condition Detection for Debugging Shared Memory in Parallel Programs”; Robert Harry Benson Netzer, University of Wisconsin - Madison, 1991

[3] ”Automatic Lock Insertion in Concurrent Programs”; Tolubaeva, M., Computational Intelligence for Modelling Control and Automation, 2008 International Conference, 10.1109CIMCA.2008.225

[4] ”Minimum Lock Assignment: A Method for Exploiting Concurrency among Critical Sections”; Yuan Zhang, Vugranam C. Sreedhar, Weirong Zhu, Vivek Sarkar, and Guang R. Gao, Languages and Compilers for Parallel Computing, DOI - 10.1007978-3-540-89740-8-10, Print ISBN - 978-3-540-89739-2

[5] ”Multithreaded Programming Framework Development for gcc infrastructure”; Dr. Niranjan N. Chiplunkar, B. Neelima, Mr. Deepak. 978-1-61284-840-2/11/ 2011 IEEE

[6] https://computing.llnl.gov/tutorials/pthreads/

[7] GNU Compiler Collection; http://gcc.gnu.org

[8] GNU Compiler Collection Internals http://gcc.gnu.org/onlinedocs/gccint/

[9] http://en.wikipedia.org/