**Program Inspection Tool**

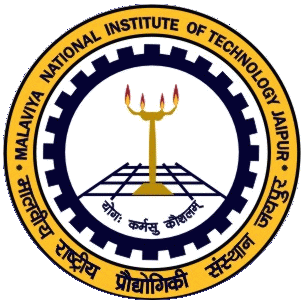
*A dissertation report submitted in fulfilment of the requirements for the degree of Bachelor of Technology*

*by*

Ronak Gadia (2016UCP1185)

Varun Verma (2016UCP1368)

Awadhesh Kumar (2016UCP1441)



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MALAVIYA NATIONAL INSTITUTE OF TECHNOLOGY

APRIL 2020

# Certificate

We,

RONAK GADIA (2016UCP1185)

VARUN VERMA (2016UCP1368)

AWADHESH KUMAR (2016UCP1441)

Declare that this thesis titled, “**PROGRAM INSPECTION TOOL**” and the work presented in it are our own. I confirm that:

* This project work was done wholly or mainly while in candidature for a B.Tech. degree in the department of computer science and engineering at Malaviya National Institute of Technology Jaipur (MNIT).
* Where any part of this thesis has previously been submitted for a degree or any other qualification at MNIT or any other institution, this has been clearly stated. Where we have consulted the published work of others, this is always clearly attributed. Where we have quoted from the work of others, the source is always given. With the exception of such quotations, this Dissertation is entirely our own work.
* We have acknowledged all main sources of help.

Signed:



Date:



Dr. Vijay Laxmi

Professor

April 2020 Department of Computer Science and Engineering

Malaviya National Institute of Technology Jaipur

# Abstract

The project aims at detecting various errors which occur at runtime in a C/C++ application. Memory errors such as invalid memory access, uninitialized memory, dangling pointer, heap overflow, stack overflow, etc cause various security issues and it may slow down the system eventually. Multi-threading issues such as deadlock, datarace may affect the consistency of the data and may prove fatal. So it's important to detect these issues before-hand and be able to fix them. We are working towards developing a tool that determines statically as well as dynamically about any of these issues and post warning messages about the same.

Using different detection algorithms and libraries, we aim to package them together into a single such application that warns us about the issues present and the lines of code that caused the problem. Such a tool greatly helps the application developing team to develop an error free, non-vulnerable code and ensures security to the system as well as the data.

ii

# Acknowledgements

It is a matter of great pleasure and privilege for us to present oue B.Tech Project report on “Program Inspection Tool”. I wish to express my deep sense of gratitude to Dr Vijay Laxmi, our project supervisor for her unparalleled guidance and motivation. A constant push and constructive feedback made this possible and helped us to have hands-on experience with the new technologies.

Ronak Gadia Varun Verma Awadhesh Kumar

(2016UCP1185) (2016UCP1368) (2016UCP1441)

iii

**Contents**

[**Certificate**](#_3oaac5ifw1jo) **1**

[**Abstract**](#_rpax1d1yxz1i) **2**

[**Acknowledgements**](#_ro9okeq8d17u) **3**

[**List of Figures**](#_4w4eptofjg1d) **6**

[**1. Introduction**](#_x7w1tt1qlgud) **1**

[1.1 Static Analysis](#_qbqu4gkc27fs) 1

[1.2. Dynamic Analysis](#_48kksv72ea0g) 2

[**2. Important Terms and Concepts**](#_7dves359lb4e) **3**

[2.1. Errors](#_8tqvz5x6qrjz) 3

[2.2. Thread Errors](#_mqp37mk0x0z7) 3

[2.2.1. Datarace](#_la1ioq3nnptw) 3

[2.2.2. Deadlock](#_8x7kfke478pm) 4

[2.2.3. Livelock](#_k94u9kqwkmu1) 4

[2.2.4. Starvation](#_2ypbkel5pkgp) 4

[2.3. Memory Errors](#_vcry266tvor5) 4

[2.3.1. Uninitialized pointers](#_malphrgbxdk) 5

[2.3.2. Uninitialized memory](#_v1dm682m4yrc) 5

[2.3.3. Array index out of bounds](#_pk8hqperjwac) 5

[2.4. Address Errors](#_iu16g3xluqtm) 5

[2.4.1. Dangling pointer](#_977gpc2qv4wf) 6

[2.4.2. Stack overflow](#_mnb88zfbjggx) 6

[2.5. Wait-for Graph](#_pqvul63iq8ak) 6

[2.6. Thread Affinity](#_jwpqdmdcz50i) 7

[**3. Tools and Technologies Used**](#_hm22kzcbm2x0) **8**

[3.1 Intel-Inspector](#_bnmbuq4jolem) 8

[3.1.1 Launch the Intel Inspector](#_184alkkuciy3) 8

[3.1.2. Choose/Create Project](#_e62u9iqhgg2x) 8

[3.1.3. Configure Project](#_rw0ocxrpx72f) 9

[3.1.4 Configure Analysis](#_422qehwhet63) 9

[3.2 Thread Sanitizer](#_w7tqqze53fkt) 10

[3.3 Address Sanitizer](#_dkcc401uszwz) 10

[3.4 Memory Sanitizer](#_xj3jhjjz0dqu) 11

[**4. Proposed Method**](#_dfmhsrq1gh2z) **12**

[4.1. Detecting Errors Using Intel-Inspector](#_9709sss4ydsb) 12

[4.1.1. Detecting DataRace Errors](#_trg4zxgsf8hc) 12

[4.1.2 Detecting Deadlock Errors](#_yf684ayn4g2r) 13

[4.1.3. Detecting Uninitialized pointers Errors](#_mlhedx2r2u1l) 13

[4.2 Detecting deadlock using Wait for graph algorithm](#_aby04xc11uhw) 14

[4.3. Data-race detection using ThreadSanitizer](#_kvwqc7i66bwb) 16

[4.4. Address error detection by AddressSanitizer](#_e3wnkeeghw04) 16

[4.5. Memory error detection by MemorySanitizer](#_g8scbelkx7ac) 17

[4.6. Error Detection by Program Inspection Tool](#_izcyvdbcmqpv) 17

[4.6.1 About the script install\_and\_setup.sh](#_b5xlyrk5k90h) 17

[4.6.2 About the script inspect.sh](#_qlf9e31oft44) 18

[**5. Experimental Results**](#_6gdjw0tzqr54) **20**

[5.1. Results of detected errors using intel-inspector](#_69wxh01qucro) 20

[5.1.1. Results of detected data-race](#_jr349xd2fp61) 20

[5.1.2. Results of detected memory error](#_89ql8n1xxgg4) 21

[5.1.3. Results of deadlock detection by Intel-Inspector](#_9hnistuoh0m6) 21

[5.2. Results of detected deadlock using wait for graph algorithm](#_8w90l82evk2d) 22

[5.3. Results of detected data-race using ThreadSanitizer](#_3pw35nf2p1km) 22

[5.4. Results of detected address error by AddressSanitizer](#_3009o4xm35qd) 23

[5.5. Results of detected Memory error by MemorySanitizer](#_reu2roav9h9s) 23

[5.6. Results of error detection by Program Inspection Tool](#_26ns8m5dqxfi) 24

[**6. Conclusion**](#_q0w5mxhwd9jk) **25**

[**References**](#_brcghb4er52v) **26**

v

# List of Figures

Figure 1 : Code for Datarace 12

Figure 2 : Code for deadlock 13

Figure 3 : Code for uninitialized pointer 14

Figure 4 : Deadlock detection code (part-1) 14

Figure 5 : Deadlock detection code (part-2) 15

Figure 6 : Deadlock detection code (part-3) 15

Figure 7 : Data Race detection using ThreadSanitizer 16

Figure 8 : Data Race detection using AddressSanitizer 17

Figure 9 : Data Race detection using MemorySanitizer 17

Figure 10 : install\_and\_setup.sh script file 18

Figure 11 : inspect.sh script file 19

Figure 12 : Data race detected using Intel-Inspector 20

Figure 13 : Stack call trace of swap1() and swap2() 20

Figure 14 : Memory error detected using Intel-Inspector 21

Figure 15 : Stack trace of memory error 21

Figure 16 : Deadlock not detected 21

Figure 17 : Deadlock detection result using wait-for graph 22

Figure 18 : Datarace detected using Thread Sanitizer 22

Figure 19 : Address error detected using Address Sanitizer 23

Figure 20 : Memory error detected using Memory Sanitizer 23

Figure 21 : Analysis by Program Inspection Tool 24

vi

**Chapter 1**

# **1. Introduction**

Error is an unlawful activity performed by the client which brings about unusual working of the program. Programming mistakes regularly stay undetected until the program is assembled or executed. A portion of the mistakes hinder the program from getting accumulated or executed. Accordingly mistakes ought to be evacuated before assembling and executing.[1]

Debugging serial and parallel programs can be very tedious. The typical debugging process is: 1) start with compilation and rectify the compile-time errors, 2) then execute and correct the run-time errors issued by the run-time system and 3) then use a debugger and/or print statements to find and correct the rest of the errors, i.e. the errors not detected at run-time and logical errors. Accumulate time mistakes can typically be amended rapidly since compilers ordinarily issue great blunder messages. So also, for the most part run-time blunders can be remedied rapidly without a debugger and print proclamations when the run-time framework accurately analyzes the mistake and issues a decent mistake message. In any case, rectifying different blunders can be very tedious.[2]

When you want to check your application is reliable, secure and accurate, you have to use a lot of different tools which are based upon static and dynamic analysis.

## **1.1 Static Analysis**

In static code analysis, we determine various mistakes in the source code of different applications.There is no need of breaking the program because the inspection will be performed on the accessible code base. static inspection is performed with the help of alleged code survey and exception of that static examination because alleged code survey is having the nearest relationship and it is also having computerized adaptation of code audit.

The main advantages of static analysis:

1. Bugs can be detected at an early stage of the software development life cycle. The earlier a bug is detected in SDLC, the lower is the cost to rectify those errors.
2. It greatly helps to precisely pinpoint the potential bug in the program code.
3. Even if the code has a loop which gets executed many number of times, the static analysis checks the entire line of code i.e. it provides the full code coverage.
4. As the static analysis does not happen at runtime, no input data is needed to be made available.
5. Static analysis detects typing and plagiarism related mistakes pretty easily.

To be able to detect such errors, the runtime execution and analysis of the program is important. But such analysis is very difficult and requires a lot of memory and CPU time. Static analysis is able to detect only simple programming error cases. Dynamic analysis is able to detect memory releases and related errors.[3]

## **1.2. Dynamic Analysis**

Dynamic code analysis is the inspection that is performed on a program at run time. First of all you have to convert your source code into an executable file when you want to perform dynamic analysis.This analysis is performed using a different set of input data . The quality and quantity of the test input data describes the effectiveness of dynamic analysis because this data determines the extent of code coverage at the end of the test.

The main advantages of dynamic analysis:

1. Dynamic analysis is performed on an executable program so we don't need to access the program's source code.
2. indexing beyond array bounds and memory leaks will be detected easily.
3. It can also detect memory,address and threading errors such as datarace and deadlock on executable code .

Most usage of dynamic analyzers don't create bogus positives since errors get captured as they happen. Along these lines, an admonition gave by a dynamic analyzer isn't an expectation made by the tool dependent on the examination of the program model however a unimportant explanation of the way that an error has happened.[3]

**Chapter 2**

# **2. Important Terms and Concepts**

## **2.1. Errors**

Much of the programming errors do not terminate the program but they cause unpredictable results. This outcome is then utilized for ensuing project computations and may not bring about a recognizable program error until some other time, making the trackdown of the error hard to find. The errors also cause vulnerability and reduce the security of the program.[4] The different errors which are lethal during runtime execution of program are:-

* Thread Errors
* Memory Errors
* Address Errors

## **2.**2. **Thread Errors**

Threads in a similar process share the same memory space. This permits simultaneously running code to couple firmly and advantageously trade data without the overhead or intricacy of an IPC. At the point when shared between threads, be that as it may, even straightforward data structures become inclined to race conditions on the off chance that they require more than one CPU guidance to refresh: two threads may wind up endeavoring to refresh the data structure simultaneously and discover it out of the blue evolving underneath. Bugs brought about by race conditions can be hard to duplicate and isolate.There are numerous errors that can cause string errors, for example, data race, deadlock, livelock and starvation.[5]

### 2.2.1. Datarace

A data race occurs when:

* When many threads concurrently share the same memory space
* At least one of those threads perform a write operation in the shared memory space
* The resources i.e. the memory locations are not locked/synchronized by the threads

When the above conditions hold true, then the program performs conflicting reads/writes which leads to unpredicted results.[6]

## 

### 2.2.2. Deadlock

In parallel programming, a deadlock occurs when a thread is waiting for a resource which is being held by another thread and that thread is waiting for a resource which is being held by the first thread. Deadlock is inevitable if the program has multithreading. In multithreaded programming and parallel computing where software and hardware locks are used to share resources and perform process synchronization.

A deadlock can take place when a process or thread enters a waiting state because a requested resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process. If the process remains in a state for a long period of time or indefinite period of time and it is not making any progress in terms of programming, then the system is said to be in a deadlock[7]

### 2.2.3. Livelock

In livelock threads or processes will not do any useful work . When two or more threads or processes repeat the same interaction in response to changes in the other processes without doing any useful work,it is called livelock . These processes or threads are in the ready position , and they are running concurrently, but they are not doing any useful work. [8]

### 2.2.4. Starvation

When an event related to the Priority scheduling algorithms, in which a process ready to run for the CPU can wait indefinitely because of low priority it is called starvation or indefinite blocking. A steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU in a heavily loaded computer system.[9]

## **2.3. Memory Errors**

Memory errors implies an inappropriate or erroneous review, or complete misfortune, of data in the memory system for a particular detail or potentially occasion. Memory errors may incorporate occasions that never happened, or they happened uniquely in contrast to the manner in which they really ought to occur. Memory error is caused due to uninitialized memory/pointer and array out of bound.[10]

### 2.3.1. Uninitialized pointers

Uninitialized pointers are called as wild pointers in C which point to random memory locations. This wild pointer may lead a program to act wrongly or to crash.[11]

### 2.3.2. Uninitialized memory

Initially all the memory when the program begins its execution is uninitialized. The value of each bit in each of the memory locations cannot be determined that it is 0 or 1 at this time. If this uninitialized memory is used or referenced in the program then it may lead to unpredictable results.[11]

### 2.3.3. Array index out of bounds

This programming error is a part of stack or heap buffer overflow. This error is caused due to the access of index values which exceeds the size of the array. As the stack has allocated the memory upto the specified length then accessing any value greater than its size would result into accessing any heap value and hence this error is displayed.[12]

## 2.4. Address Errors

A location error happens at whatever point a program has a read or a write association with an invalid location. Address errors are especially normal in C in light of the fact that the C programming language gives the developer direct access to the program's memory, which can help execution yet in addition permits programming imperfections to get to irregular memory locations. Run of the address errors incorporate too far out array records, support flood, dangling load pointers (getting to a location of store assigned memory after the memory has been liberated), dangling stack pointers (getting to a pointer to a neighborhood variable of a function) and the utilization of pointers cast from off base numeric qualities[13]

### **2.4.1. Dangling pointer**

Dangling pointers in writing computer programs are pointers that don't highlight a legitimate address of the proper kind. These are uncommon instances of memory infringement. All the more for the most part, dangling references will be references that don't set out to a legitimate goal.

Dangling pointers emerge during object obliteration, when an item that has a reference is erased or deallocated, without altering the estimation of the pointer, so the pointer despite everything focuses on the memory area of the deallocated memory. The framework may reallocate the recently liberated memory, and in the event that the program, at that point gets to the dangling pointer, eccentric conduct may result, as the memory may now contain totally irregular data.[14]

### **2.4.2. Stack overflow**

A stack overflow happens if the call stack pointer surpasses the stack bound. The call stack may comprise a limited measure of address space, regularly decided toward the start of the execution of the program. The size of stack relies upon numerous variables, including the programming language, system, multithreading, and measure of memory accessible. At the point when a program endeavors to utilize more space than is accessible on the call stack, the stack is said to overflow, typically bringing about a program crash[15]

## 2.5. Wait-for Graph

A wait-for graph in writing computer programs is a directed graph utilized for stop identification in operating frameworks and social database frameworks.In programming, a framework that permits simultaneous activity of various threads and bolting of assets and which doesn't give technique to keep away from or forestall gridlock must help a strategy to distinguish deadlocks and an algorithm for recuperating from them. One halt recognition algorithm utilizes a wait-for graph to follow which different procedures a thread is right now hindering on. In a wait-for graph, threads are spoken to as hubs, and an edge from thread Ti to Tj infers Tj is holding an asset that Ti needs and in this way Ti is waiting for Tj to discharge its lock on that asset. The possibility of a stop is suggested by graph cycles in the developed wait-for graph. There is no straightforward algorithm for distinguishing the possibility of stop in the last case .[16]

## 2.6. Thread Affinity

Thread affinity gives a path to an application thread to tell the OS precisely where its threads might want to run. The scheduler doesn't have to contribute a lot of vitality load altering the framework since application threads are starting at where they ought to be.. Thread affinity permits programming threads to execute inside the extent of explicit handling assets. For instance, when running two MPI errands on a 4-core hub, one might need to tie all threads from task 1 to the main core and the threads from task 0 to the subsequent core and no threads n the staying two core..[17]

**Chapter 3**

# 3. Tools and Technologies Used

## 3.1 **Intel-Inspector**

Intel Inspector is used for detection of memory error and thread error , to have reliable, secure and accurate C/C++ applications.

* Reliability: To detect deadlocks and memory errors that cause locking & crashes.
* Security: To detect memory and threading vulnerabilities used by attackers.
* Accuracy: To have the information about memory corruption and race conditions to have accurate results

Memory error discovery incorporates memory spills, dangling reference pointers, uninitialized factors, utilization of invalid memory references, bungled memory, designation and deallocation, stack memory checks, and stack trace with controllable stack trace profundity

Thread error discovery incorporates race conditions, deadlocks, profundity configurable call stack investigation, diagnostic direction, worked in information on Threading Building Blocks (TBB), OpenMP, and POSIX .

### **3.1.1 Launch the Intel Inspector**

To launch the:

* Intel Parallel Studio XE/Intel Inspector standalone GUI: Run the inspxe-gui command.
* To launch the command line interface: Run the inspxe-cl command.

### **3.1.2. Choose/Create Project**

You have to create or open a project to enable analysis features because Intel Inspector is based on a project paradigm.

Analysis of our project as follows:

* Compiled application
* including suppression rules and search directories
* Container for analysis results and collection of configurable attributes

### **3.1.3. Configure Project**

Our target: In as short a runtime period as could reasonably be expected, execute the same number of ways and the most extreme number of assignments (equal exercises) as you can afford, while limiting the repetitive calculation inside each undertaking to the absolute minimum required for good code inclusion since data set size and remaining task at hand directly affect application execution time and examination speed. For best outcomes we need to choose little and delegate data sets that make threads with negligible to direct work per thread.

We need to make extra data sets to guarantee all your code is examined.

### **3.1.4 Configure Analysis**

Intel Inspector offers a scope of preset memory and threading investigation types (just as custom examination types) to assist you with controlling examination degree and cost. The smaller the extension, the lighter the heap on the framework. The more extensive the degree, the bigger the heap on the framework.

**Run Analysis**

After doing the analysis on intel inspector :

* Runs your application.
* Determines issues that may be fixed .
* Results are the collection of issues .
* All the analysis collected in the form of filenames and line numbers.
* Suppression rules may be applied .
* Duplicate elements eliminated .
* Convert into problem sets .
* You can debug your application using the results of the above analysis.

**Choose Problems**

Intel inspector provides the order of problems that is detected during the analysis of our application. When the particular analysis is completed, intel inspector provides the following results:

* Convert group of detected problems into problem sets
* Provide the priority of problem sets.
* provides the filter of problem sets that may need handling.[18]

## 3.2 Thread Sanitizer

Thread Sanitizer is a library function or a tool which is used to debug data races when more than one threads try to access the same memory area and with at least one write operation in one of those threads. These data races are difficult to debug because they are pretty unpredictable, sometimes function correctly and sometimes not.

* Data race can be easily detected for C/C++ programs using Thread Sanitizer.
* It uses a compile-time instrumentation to check all non-race-free memory access at runtime.
* It provides very precise results with no false positives.
* The warning message provides the thread ID with the line number in code that caused datarace.
* It also reports the total datarace conflicts.[19]

## 3.3 Address Sanitizer

AddressSanitizer is an open source programming tool by Google that detects memory corruption bugs such as buffer overflows or accesses to a dangling pointer. AddressSanitizer is based on compiler instrumentation and directly-mapped shadow memory. If a bug is detected, the program will display an error message to stderr and exit with a non-zero exit code20]

It finds:

* Dangling pointer dereference
* Heap buffer overflow
* Stack buffer overflow
* Memory leaks

## 3.4 Memory Sanitizer

Uninitialized memory reads can be detected by MemorySanitizer in C/C++ programs. Uninitialized values happen when stack or heap-allocated memory is perused before it is composed. It quietly tracks the spread of uninitialized information in memory location, and it likewise creates an admonition report when a code branch is taken or not relying upon an uninitialized esteem.

**MemorySanitizer is bit-exact**: It can follow uninitialized bits in a bitfield. It will permit duplicating of uninitialized memory, and furthermore rationale and math tasks with it. As a rule, it unobtrusively tracks the spread of uninitialized information in memory, and produces an admonition report when a code is taken relying upon a uninitialized esteem.[21]

**Chapter 4**

# 4. Proposed Method

The objective is to detect threading errors, memory errors and address errors. Firstly, we have tried to use intel-inspector as a debugger in order to achieve the above objectives.

## 4.1. Detecting Errors Using Intel-Inspector

### **4.1.1. Detecting DataRace Errors**

Here, we had created multiple programs which used file, database and stack as a resource. We have intentionally put the data race conditions in our program in order to check data race detection by intel-inspector for these resources. As we have discussed earlier about how to set up and configure a project in intel inspector, in the same way, we performed the data race detection for the above programs. The intel inspector dynamically analyzes the program for 7 minutes for different input sets and then outputs the result of inspection.

A code snippet for the datarace is given in the figure below.



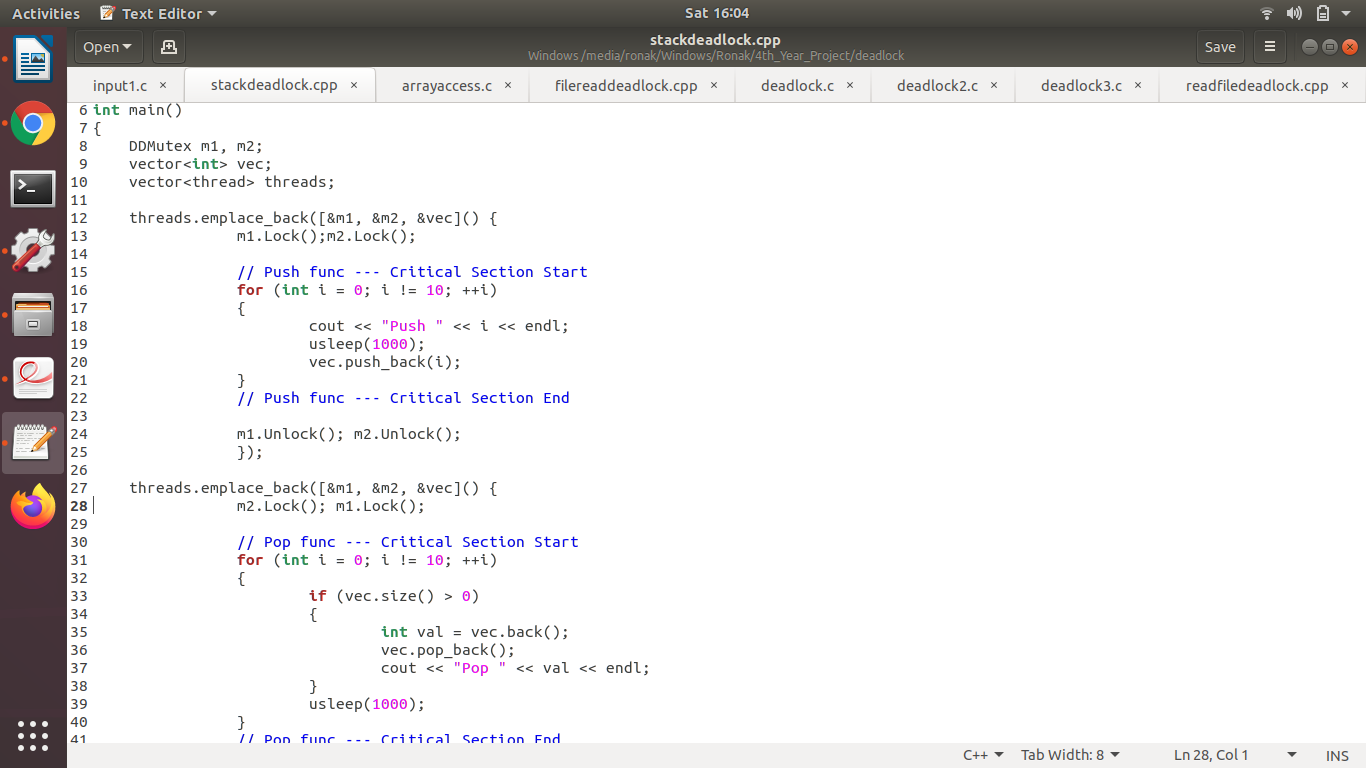
*Figure 1 :- Code for Datarace*

Here data race occurs for the global variable a and b when the two threads run the function swap1() and swap2() concurrently.

### **4.1.2 Detecting Deadlock Errors**

Here, we had created multiple programs which used file, database and stack as a resource. We have intentionally put the deadlock conditions in our program in order to check deadlock detection by intel-inspector for these resources. As we have discussed earlier about how to set up and configure a project in intel inspector, in the same way, we performed the deadlock detection for the above programs. The intel inspector dynamically analyzes the program for 12 minutes for different input sets and then outputs the result of inspection.

A code snippet for the deadlock is given in the figure below.

****

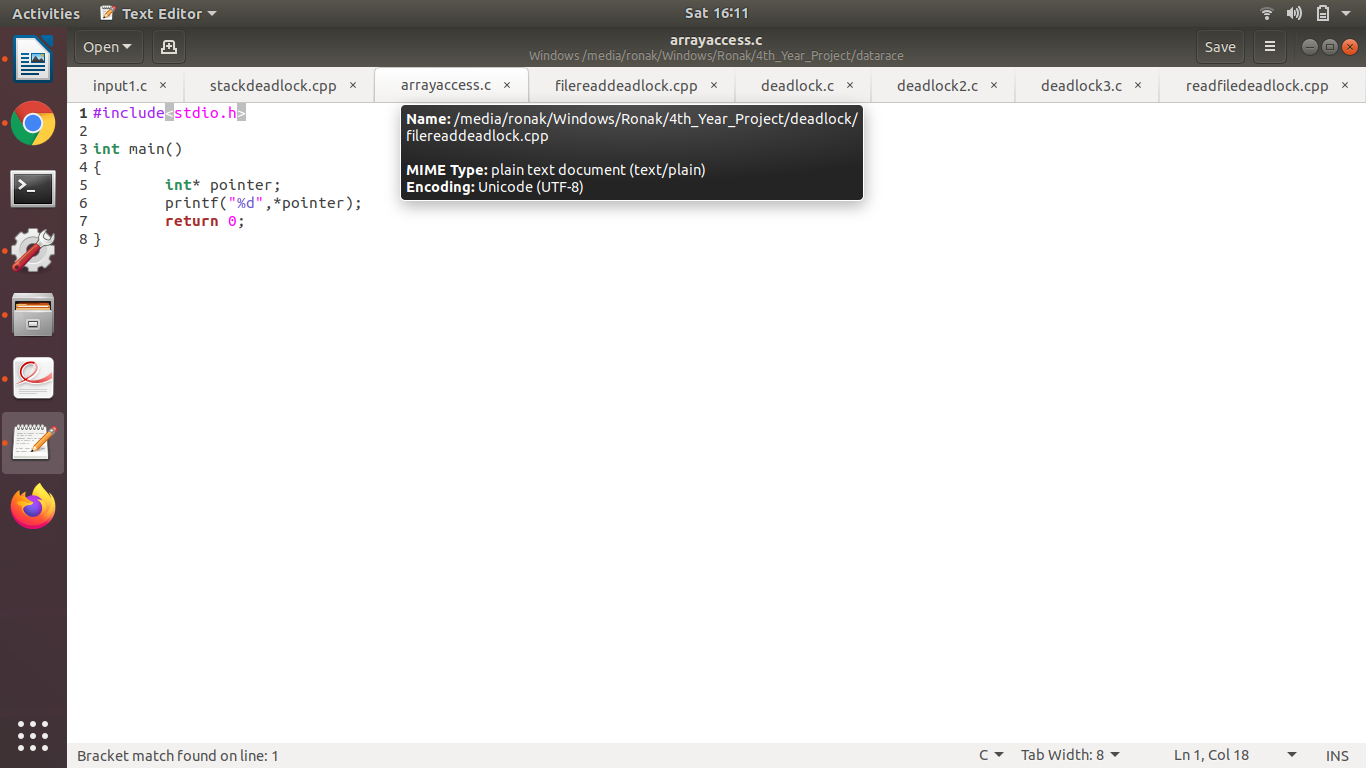
*Figure 2 :- Code for deadlock*

Here deadlock occurs for the vector vec which is passed as a resource. When the mutex in the two threads wait for the other mutex, then deadlock occurs.

### **4.1.3. Detecting Uninitialized pointers Errors**

Here, we had created multiple programs which used file, database and stack as a resource. We have intentionally put the Uninitialized pointers in our program in order to check Uninitialized pointers detection by intel-inspector for these resources. As we have discussed earlier about how to set up and configure a project in intel inspector, in the same way, we performed the Uninitialized pointers detection for the above programs. The intel inspector dynamically analysis the program in a few seconds for different input sets and then outputs the result of inspection.

A code snippet for the uninitialized pointer is given in the figure below.

****

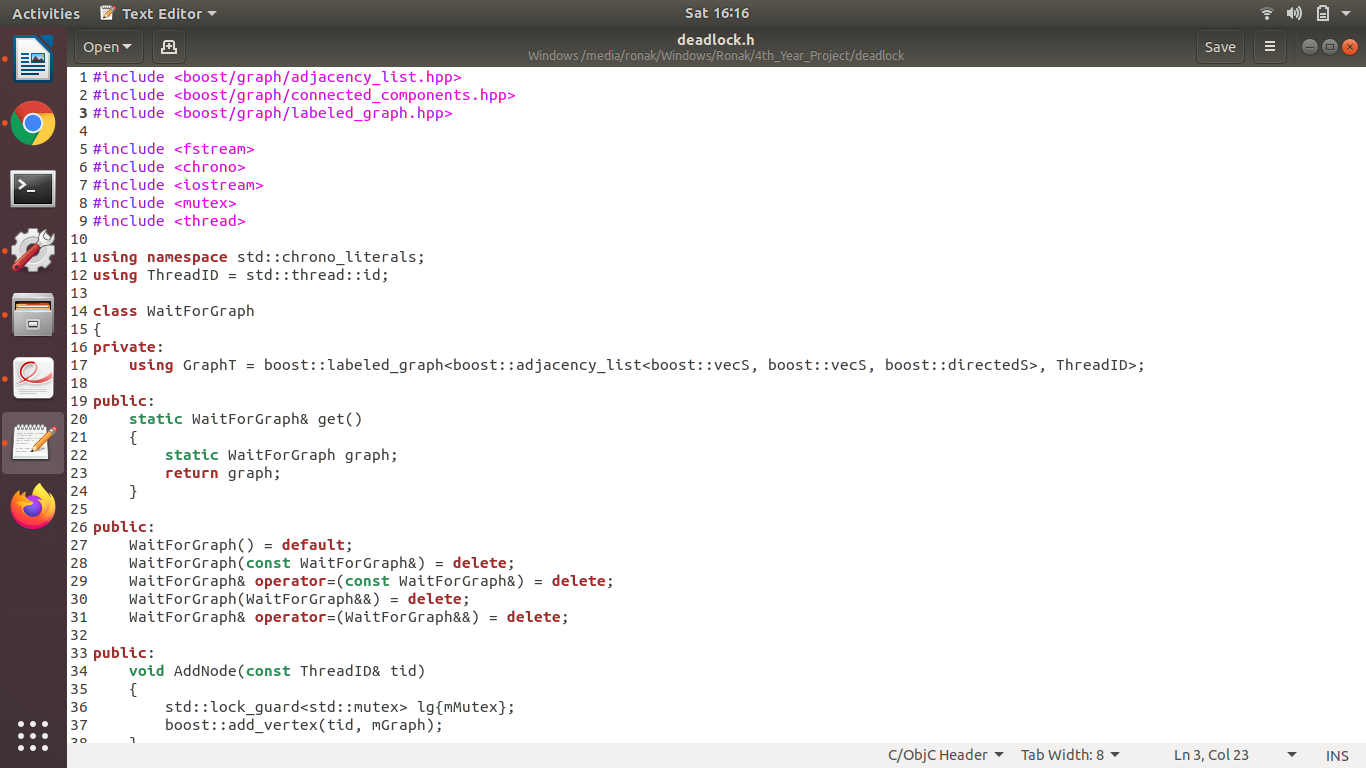
*Figure 3:- Code for uninitialized pointer*

## 4.2 Detecting deadlock using Wait for graph algorithm

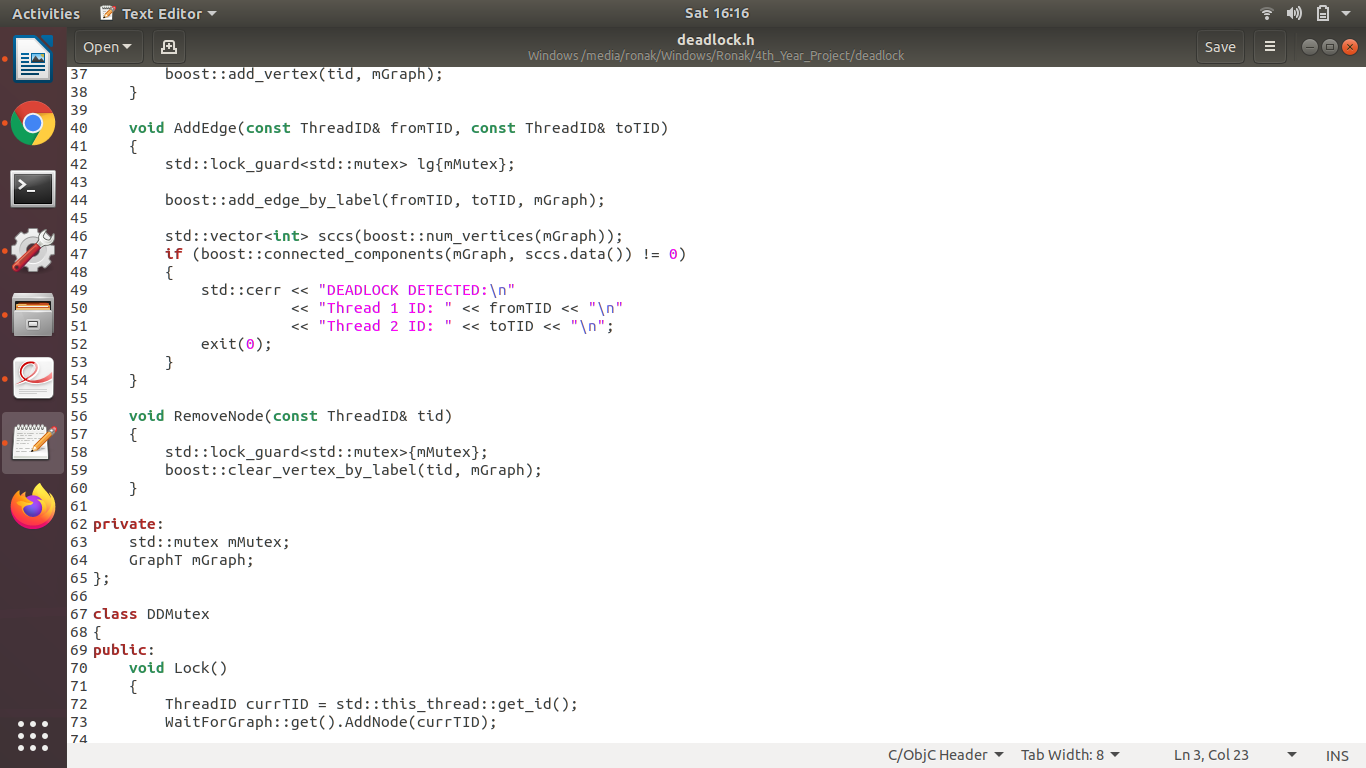
Intel-inspector is not able to detect deadlocks very efficiently so we used a wait-for graph algorithm to detect the deadlock. We implemented the Wait-for graph algorithm in C++. The code was able to detect deadlock for file as a resource, database as a resource

and stack as a resource.

The Wait-For Graph code implemented in C++ is given below:-



*Figure 4 :- Deadlock detection code (part-1)*

****

*Figure 5 :- Deadlock detection code (part-2)*

****

*Figure 6 :- Deadlock detection code (part-3)*

## 4.3. Data-race detection using ThreadSanitizer

As intel-inspector is unable to find the line of code that resulted in data-race so we have used ThreadSanitizer tool for this purpose.

Here, we Simply compile the program with -fsanitize=thread and link it with -fsanitize=thread. To get a reasonable performance add -O2. Use -g to get file names and line numbers in the warning messages.

When you run the program, ThreadSanitizer will print a report if it finds a data race. Here is an example:

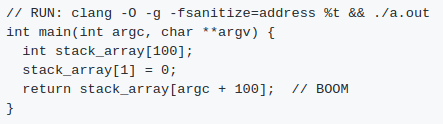


*Figure 7 :- Data Race detection using ThreadSanitizer*

## 4.4. Address error detection by AddressSanitizer

As intel-inspector is unable to detect dangling reference error,stack overflow and heap overflow so we have used AddressSanitizer tool for this purpose. AddressSanitizer is also providing the information about the line of code that is causing the above errors.

In order to use AddressSanitizer, compile the code and link the program using clang with the -fsanitize=address switch. To get a reasonable performance add -O1 or higher. To get nicer stack traces in error messages add -fno-omit-frame-pointer

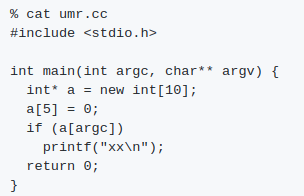
****

*Figure 8 :-**Data Race detection using AddressSanitizer*

## 4.5. Memory error detection by MemorySanitizer

As intel-inspector is unable to detect uninitialized memory,and array index out of bound so we have used MemorySanitizer tool for this purpose. MemorySanitizer is also providing the information about the line of code that is causing the above errors.

To use MemorySanitizer, compile and link your program with -fsanitize=memory -fPIE -pie. To get any stack traces, add -fno-omit-frame-pointer.



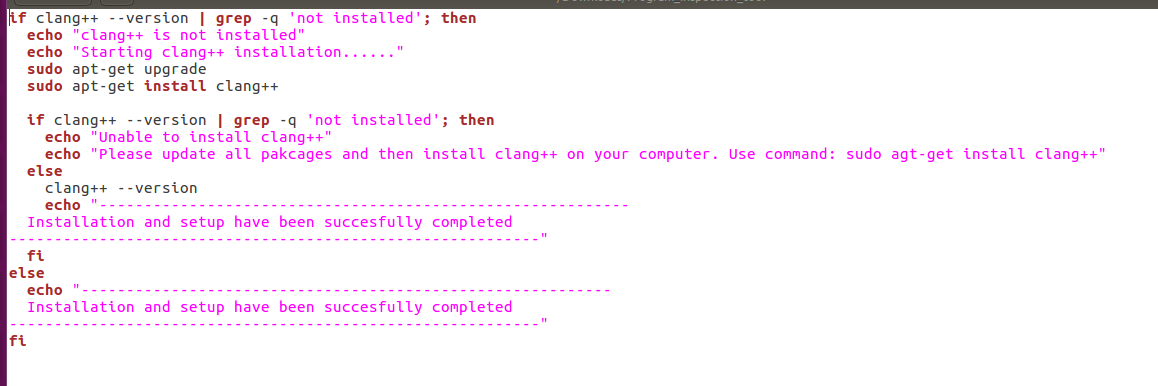
*Figure 9 :- Data Race detection using MemorySanitizer*

## 4.6. Error Detection by Program Inspection Tool

So, finally we have packaged all the above error detection tools and algorithms into a single module to simplify the procedure of detecting various multithreading and memory errors. Here, it consists of two script files, install\_and\_setup.sh and inspect.sh.

### **4.6.1 About the script install\_and\_setup.sh**

To be able to use the sanitizer tools, we need the clang++ library. This script’s function is to help us install the clang++ library into our computer. The code in script is given below.

****

*Figure 10:- install\_and\_setup.sh script file*

### **4.6.2 About the script inspect.sh**

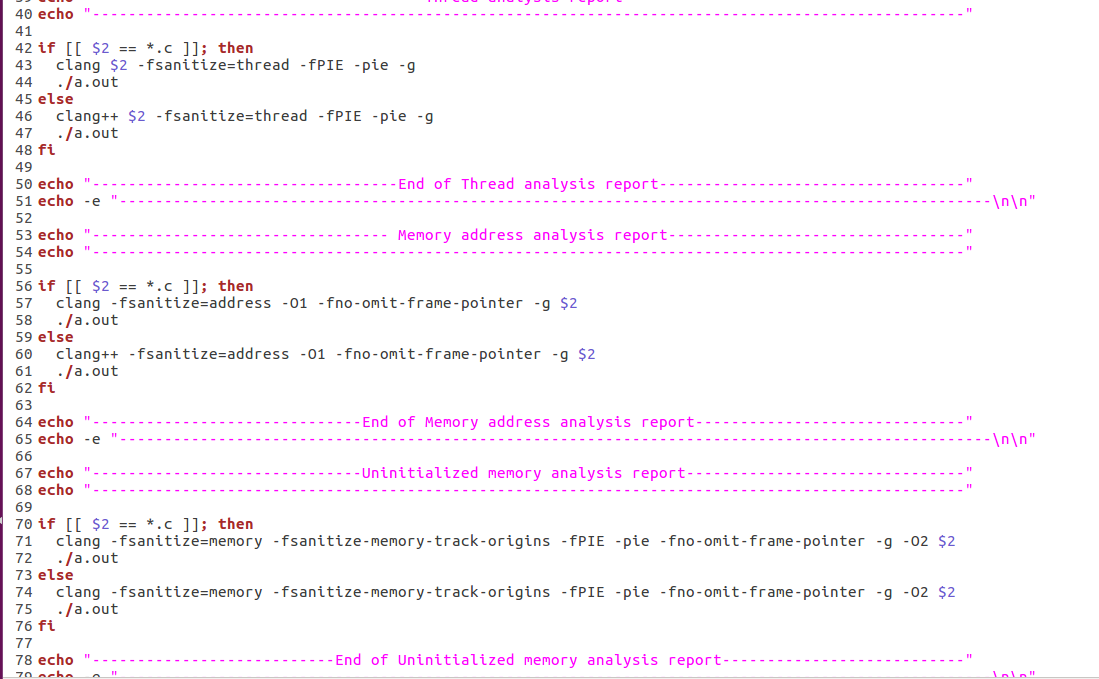
This script performs the analysis on the file which is passed as its argument. Two arguments must be provided, the first argument is the /path/to/directory of the file on which the analysis is to be performed and the second argument is the filename (with extension).

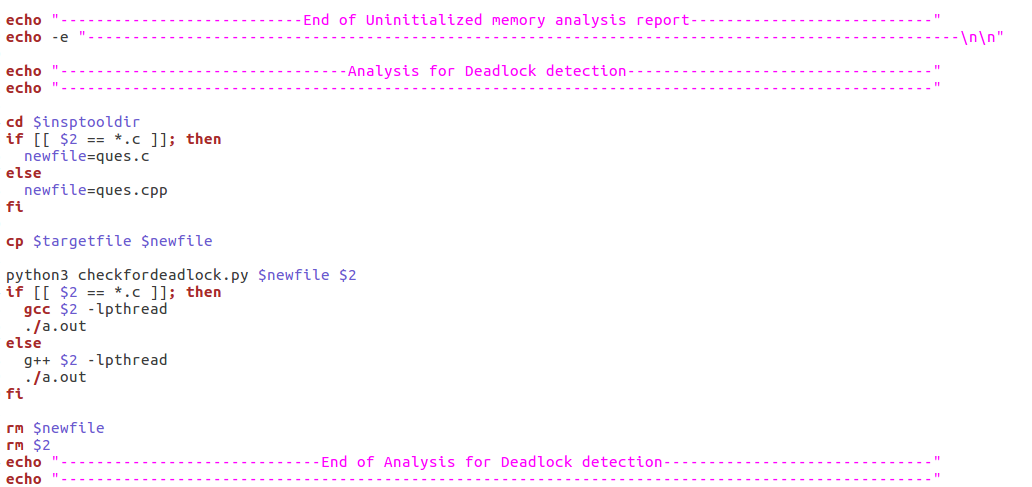
The script performs the following checks:-

* check if there are valid no of arguments
* check if the directory exists
* check if the file is having extension of c/cpp only

After performing the checks, it runs the clang / clang++ command to analyze the file by ThreadSanitizer, AddressSanitizer and MemorySanitizer, one after the other. Finally it checks for deadlock using the Wait-for graph algorithm which is coded in the header file “deadlock.h” in a single go.

The code for the inspect.sh script file is given below:-

****

****

*Figure 11 :- inspect.sh script file*

**Chapter 5**

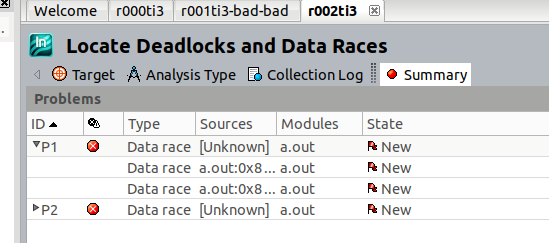
# 5. Experimental Results

We have detected data-race, deadlock, stack overflow etc using different techniques and tools. The results include a detailed report which gives warnings of the errors and the lines of code that caused those specific errors.

## 5.1. Results of detected errors using intel-inspector

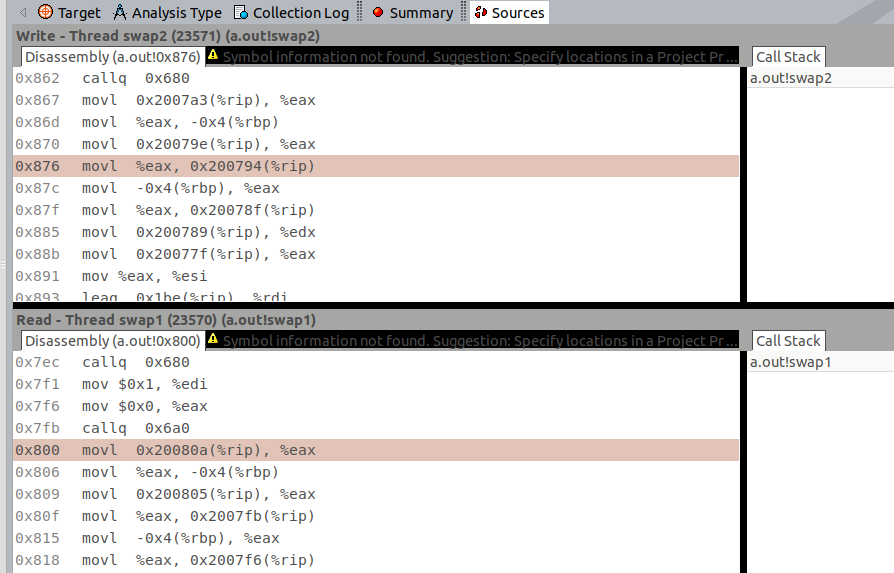
### **5.1.1. Results of detected data-race**

It displays the number of dataraces while doing dynamic analysis of the program

****

*Figure 12: Data race detected using Intel-Inspector*

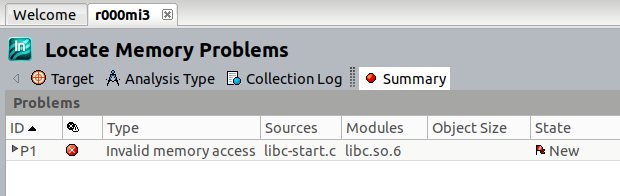
Displays the call stack trace of the threads that cause datarace. The redline points out the assembly code that resulted in data-race.

****

*Figure 13: Stack call trace of swap1() and swap2()*

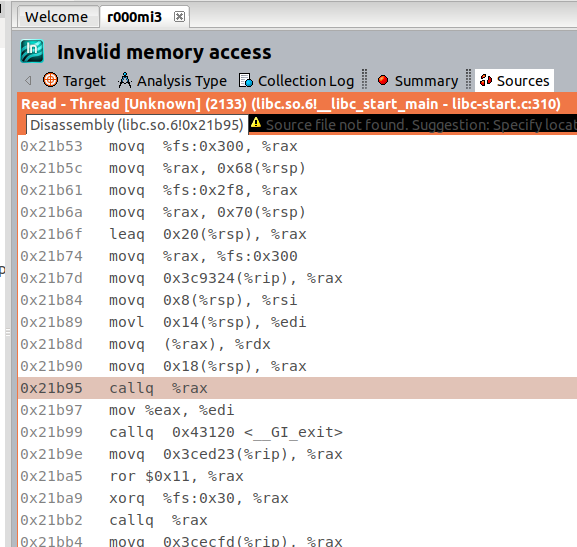
### **5.1.2. Results of detected memory error**

It displays the Memory Error while doing dynamic analysis of the program

****

*Figure 14: Memory error detected using Intel-Inspector*

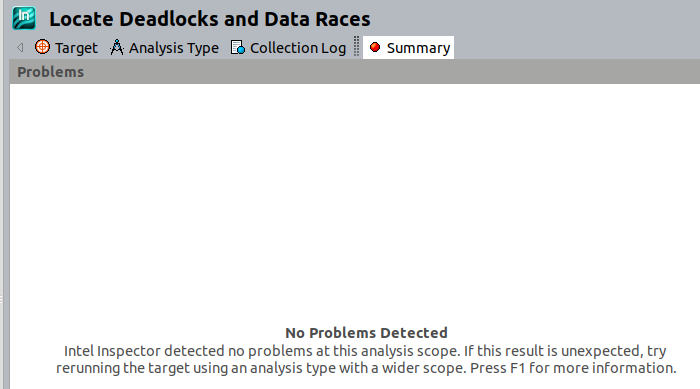
Displays the call stack trace of the threads that cause memory error The redline points out the assembly code that resulted in memory error.

****

*Figure 15: Stack trace of memory error*

### **5.1.3. Results of deadlock detection by Intel-Inspector**

The intel-inspector is not able detect deadlock in thread analysis using both mutex as well as semaphore.

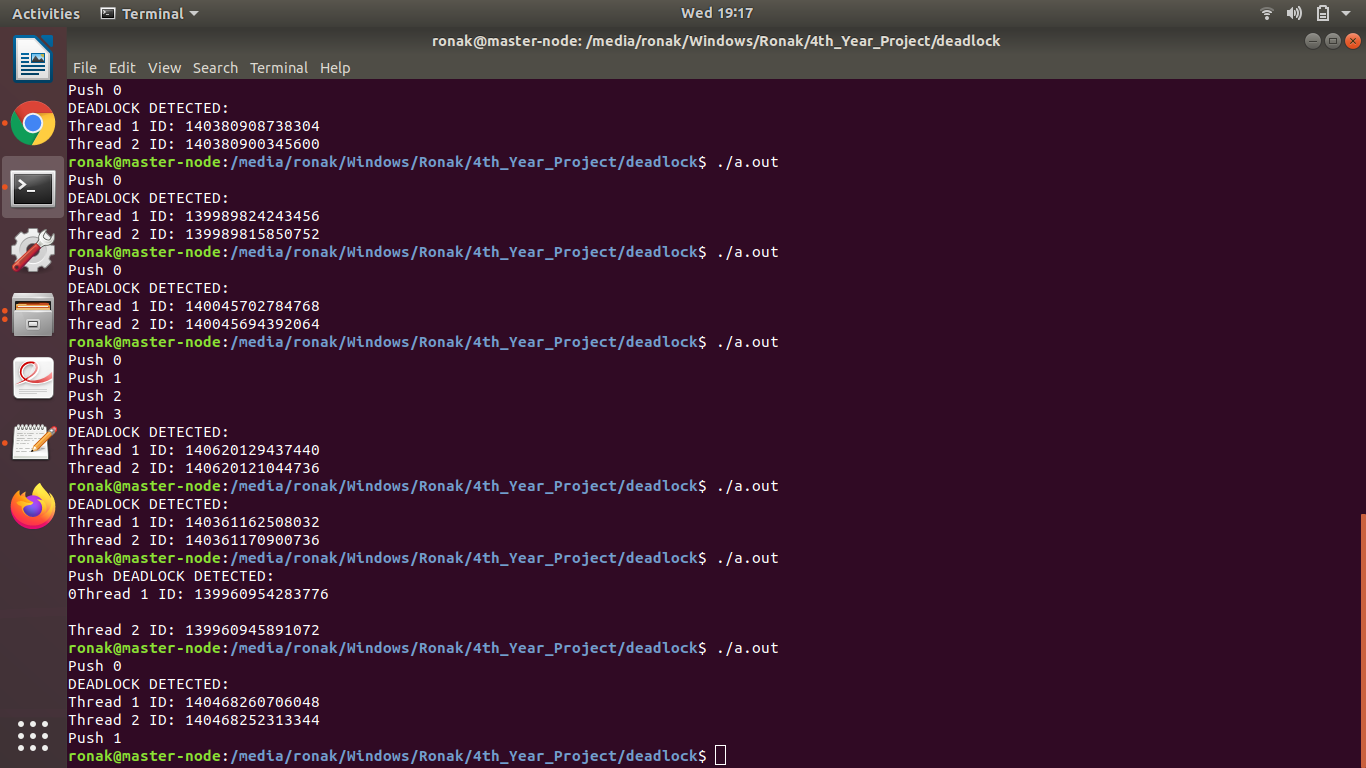
****

*Figure 16: Deadlock not detected*

## 5.2. Results of detected deadlock using wait for graph algorithm

This algorithm was able to detect deadlock for file as a resource, database as a resource and stack as a resource.

The output for stack as a resource is given below.

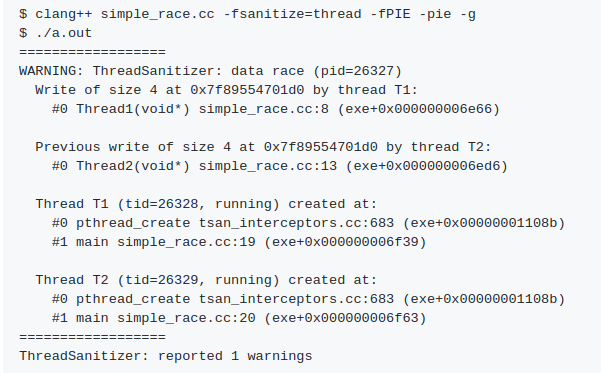
****

*Figure 17: Deadlock detection result using wait-for graph*

## 5.3. Results of detected data-race using ThreadSanitizer

In ThreadSanitizer the warning message provides the thread ID with the line number in code that caused data-race.It also reports the total data-race conflicts.

An example of using clang is given below:-

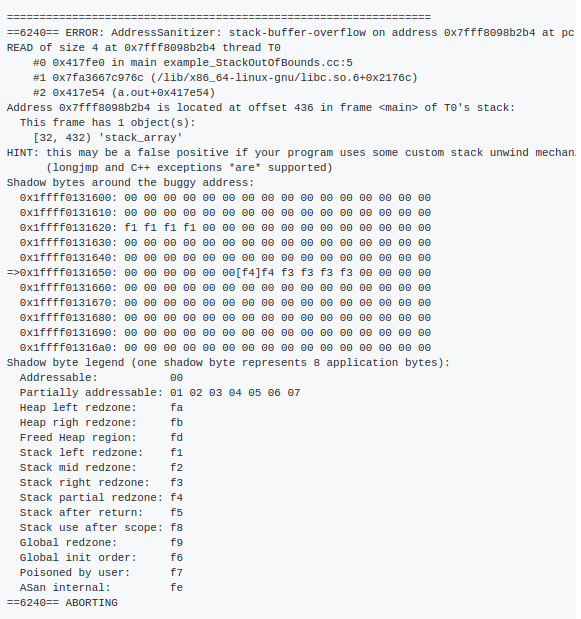


*Figure 18: Datarace detected using Thread Sanitizer*

## 5.4. Results of detected address error by AddressSanitizer

In AddressSanitizer, if a bug is detected, the program will print an error message to stderr and exit with a non-zero exit code.

An example of using clang library for AddressSanitizer is given below:-

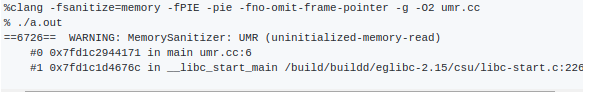


*Figure 19: Address error detected using Address Sanitizer*

## 5.5. Results of detected Memory error by MemorySanitizer

MemorySanitizer silently tracks the spread of uninitialized data in memory, and reports a warning when a code branch is taken or not depending on an uninitialized value.

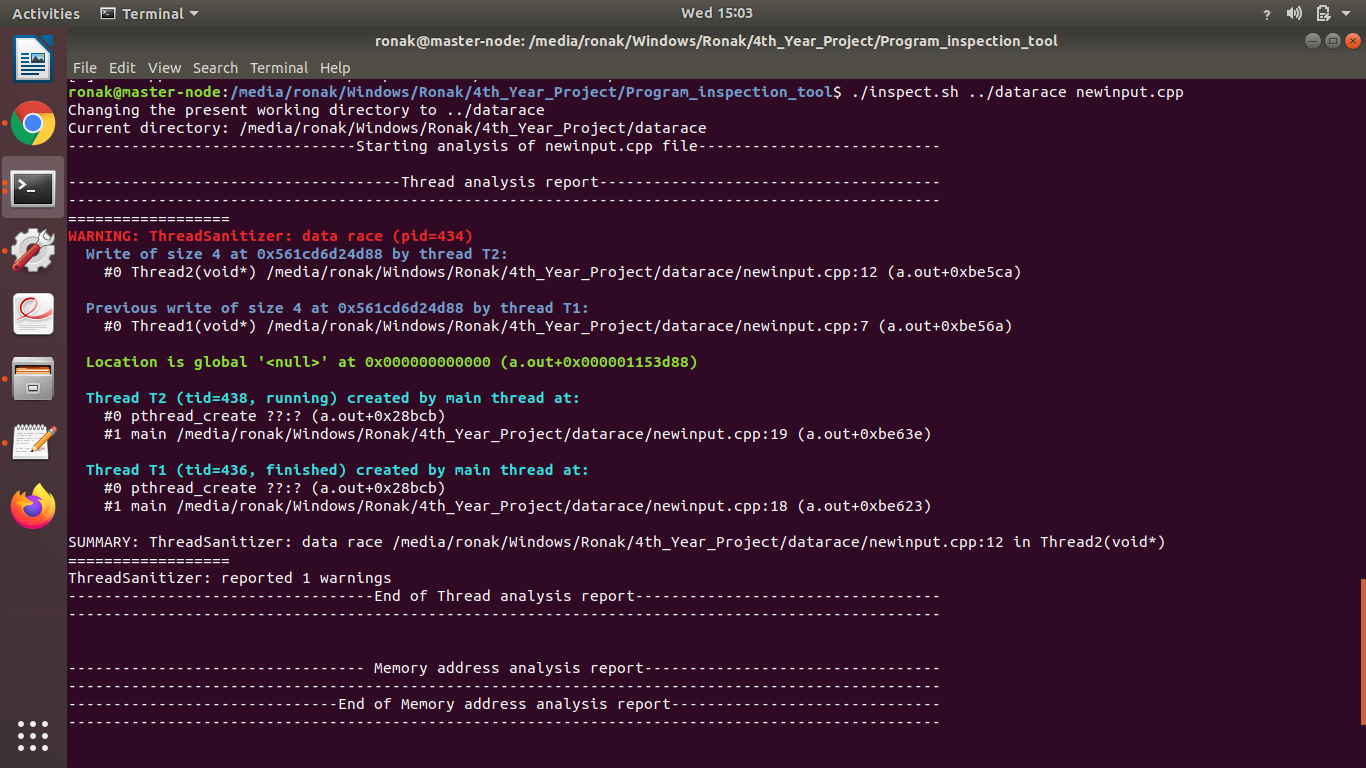
An example of using clang for memory sanitizer is given below:-

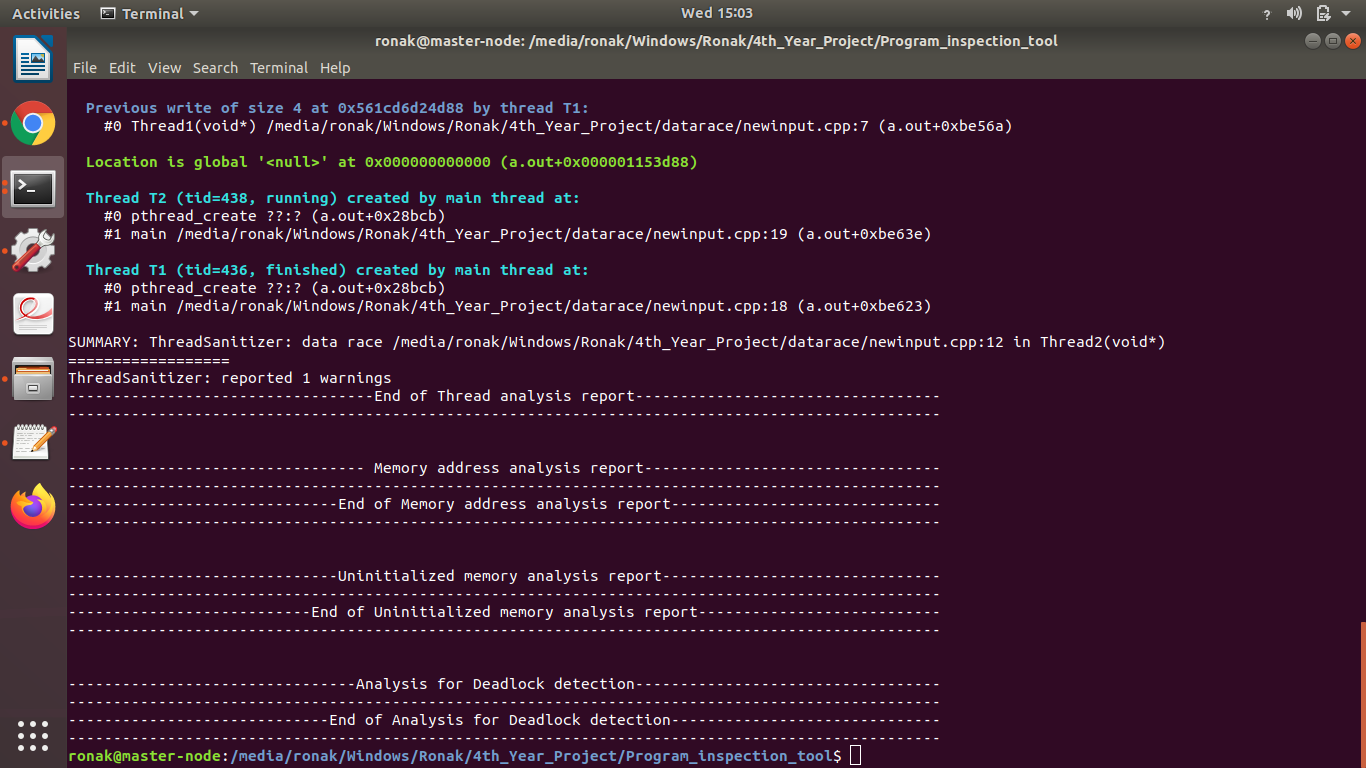
****

*Figure 20: Memory error detected using Memory Sanitizer*

## 5.6. Results of error detection by Program Inspection Tool

Program detection tool detects all the memory, address, threading errors and deadlock in a single go. The output of an example file is given below.

****

****

*Figure 21:- Analysis by Program Inspection Tool*

**Chapter 6**

# 6. Conclusion

Using Intel-inspector, it is possible to detect memory and datarace error dynamically. The experimental results concluded that Intel-inspector is unable to detect deadlock and point out the line of code that caused the above errors. To overcome these issues, we have implemented a Wait-for graph algorithm to detect deadlock. Thread sanitizer, Address sanitizer and Memory sanitizers gives a detailed report of these errors. The detailed report consists of a number of warnings along with the lines of code that have caused those errors. Hence, it has now become easy to locate dangling pointers, stack overflow, heap overflow, uninitialized memory, data races and deadlocks.

Finally, we have packaged all the algorithms and tools which we had researched into a single module which provides the detailed analysis report of the threading and memory errors as well as detects if the given program can cause deadlock or not. The Program Inspection Tool will be of great help to the C/C++ developers who use multithreading in their applications.

# References

[1] https://www.geeksforgeeks.org/errors-in-cc/

[2] https://www.researchgate.net/publication/220918552\_The\_Importance\_of\_Run-Time\_

Error\_Detection/link/0fcfd51053caaf2f9d000000/download

[3] https://www.viva64.com/en/b/0643/

[4] https://www.thecrazyprogrammer.com/2014/08/types-of-errors-in-programming.html

[5] https://www.tutorialspoint.com/major-issues-with-multi-threaded-programs

[6] https://docs.oracle.com/cd/E19205-01/820-0619/geojs/index.html

[7] https://en.wikipedia.org/wiki/Deadlock

[8] https://www.geeksforgeeks.org/deadlock-starvation-and-livelock/

[9] https://www.geeksforgeeks.org/starvation-and-aging-in-operating-systems/

[10] https://en.wikipedia.org/wiki/Memory\_error

[11] https://d3m3vilurr.gitbooks.io/the-unsafe-rust-programming-language/uninitialized.html

[12] https://www.viva64.com/en/t/0071/

[13] https://www.reactive-systems.com/c-runtime-errors.html

[14] https://en.wikipedia.org/wiki/Dangling\_pointer

[15] https://en.wikipedia.org/wiki/Stack\_overflow

[16] https://en.wikipedia.org/wiki/Wait-for\_graph

[17] https://www.sciencedirect.com/topics/computer-science/thread-affinity

[18] https://software.intel.com/en-us/get-started-with-inspector-linux

[19] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Thread\_Sanitizer

[20] https://en.wikipedia.org/wiki/AddressSanitizer

[21] https://github.com/google/sanitizers/wiki/MemorySanitizer