Crash Analysis Tool

Profiling Applications in MULTITASKING Environment

Final Evaluation Report

Faizan Ullah

15026424

Dr. Adnan Iqbal

Dr. Anjum Naveed(EXT.)

Abstract

Software applications are prone to crash due to resource snatching or resource unavailability. Many applications don’t have the mechanism to restart itself and try to reacquire unavailable resource in many cases the resource is available, but it is being held by a stray lock. Issues like these take days in manual debugging. This potential issue must be looked after by developing an application that robustly checks the locks; Whether they are susceptible to system crash, power failure and permanent locking. This will considerably improve software performance and assurance of its working. A software utility capable of analyzing crashes of multitasking application. With its help, developers can reassure that their application has the ability to handle anomalies like power off, sudden kill, accidental resources snatching, race-condition, resources unavailability, permanent locking and etc. As the common effects of these exceptions are, crashing, not restarting of the application, not able to score a particular r­­­­­ecourse, a particular feature not responding, and many more. In general, the purpose of this software is to pinpoint any locking functions that might cause the application to behave oddly. Tools utilized in developing this software is kernel utility named as ptrace and libunwind (stack unwind library).

Acknowledgments

My utmost acknowledgment is for Allah Almighty as He who helped me the most all along when I was helpless. I’m very thankful to Dr. ANJUM NAVEED for believing in me and guiding me at every stage of this project. I would like to thank Dr. Adnan as without his guidance and constant push I won’t be able to complete this project. Special thanks to my family members who had been praying for my success and believing in myself more than I did.

Table of Contents

[1 Introduction 7](#_Toc8671731)

[1.1 Problem Description 7](#_Toc8671732)

[1.2 Background 7](#_Toc8671733)

[1.3 Proposed Solution 8](#_Toc8671734)

[1.4 Aims and Objectives 8](#_Toc8671735)

[1.5 Features 8](#_Toc8671736)

[1.5.1 Software Execution 8](#_Toc8671737)

[1.5.2 Attaching to running process 8](#_Toc8671738)

[1.5.3 Syscalls detection/tracer implementation 9](#_Toc8671739)

[1.5.4 Lock Detection 9](#_Toc8671740)

[1.5.5 Printing Backtrace 9](#_Toc8671741)

[1.6 Document Structure 9](#_Toc8671742)

[2 Literature Review 10](#_Toc8671743)

[2.1 Overview 10](#_Toc8671744)

[2.2 Literature Comprehension 10](#_Toc8671745)

[2.3 Issue Identification 11](#_Toc8671746)

[2.3.1 Example 11](#_Toc8671747)

[2.4 Useful Tools Available 11](#_Toc8671748)

[2.5 API’s 12](#_Toc8671749)

[2.6 Related work 12](#_Toc8671750)

[2.7 Related Work Limitation 13](#_Toc8671751)

[3 Requirements and Analysis 14](#_Toc8671752)

[3.1 Project Description 14](#_Toc8671753)

[3.2 Required Features and Analysis 14](#_Toc8671754)

[3.2.1 Software Execution 15](#_Toc8671755)

[3.2.2 Process Attachment 15](#_Toc8671756)

[3.2.3 Communication Logging 15](#_Toc8671757)

[3.2.4 Potential Exception Detection 15](#_Toc8671758)

[3.2.5 Crash Analysis 15](#_Toc8671759)

[3.2.6 Report Generation 15](#_Toc8671760)

[4 Design and Implementation 16](#_Toc8671761)

[4.1 Overview 16](#_Toc8671762)

[4.2 Architecture and component 16](#_Toc8671763)

[4.2.1 Functional Design Identification 17](#_Toc8671764)

[4.2.2 Flow diagram 26](#_Toc8671765)

[4.3 Non-Functional Design Description 27](#_Toc8671766)

[4.3.1 Platform 28](#_Toc8671767)

[4.3.2 Scope 28](#_Toc8671768)

[4.3.3 Output 28](#_Toc8671769)

[4.4 User Interface 28](#_Toc8671770)

[4.4.1 Flags available 28](#_Toc8671771)

[4.4.2 Program to Test 29](#_Toc8671772)

[4.4.3 Report generation 29](#_Toc8671773)

[5 Testing 30](#_Toc8671774)

[5.1 Overview 30](#_Toc8671775)

[5.2 Usability testing 30](#_Toc8671776)

[5.3 Functional Testing 31](#_Toc8671777)

[5.3.1 System Calls Test 31](#_Toc8671778)

[5.3.2 Futex Kill Test 31](#_Toc8671779)

[5.3.3 Stack unwinding 32](#_Toc8671780)

[5.4 Unit testing 32](#_Toc8671781)

[6 Results and discussion 33](#_Toc8671782)

[6.1 Overview 33](#_Toc8671783)

[6.2 Testing Software 33](#_Toc8671784)

[6.3 PostgreSQL Database Server TPC-C testing using Hammerdb 33](#_Toc8671785)

[6.3.1 HammerDB 33](#_Toc8671786)

[6.3.2 TPC-C 33](#_Toc8671787)

[6.3.3 Testing 34](#_Toc8671788)

[6.3.4 Schema Building 34](#_Toc8671789)

[6.3.5 Pattern 38](#_Toc8671790)

[6.3.6 Driver Script 39](#_Toc8671791)

[6.3.7 Pattern 48](#_Toc8671792)

[6.4 Apache Tomcat 48](#_Toc8671793)

[6.4.1 Testing Pre-requisites 48](#_Toc8671794)

[6.4.2 Pre-testing Info 48](#_Toc8671795)

[6.4.3 Test Summary 49](#_Toc8671796)

[6.4.4 Testing Detail 49](#_Toc8671797)

[6.4.5 In Further Iteration 50](#_Toc8671798)

[6.5 FUSE 50](#_Toc8671799)

[6.5.1 Testing Pre-requisites 50](#_Toc8671800)

[6.5.2 Test Summary 50](#_Toc8671801)

[6.5.3 Testing Detail 50](#_Toc8671802)

[6.6 Web Browsers 51](#_Toc8671803)

[6.6.1 Opera 51](#_Toc8671804)

[6.6.2 Firefox 53](#_Toc8671805)

[6.6.3 Chrome 54](#_Toc8671806)

[6.7 OpenStack 55](#_Toc8671807)

[6.7.1 DevStack 55](#_Toc8671808)

[6.7.2 Installation 55](#_Toc8671809)

[6.7.3 Openstack Services 55](#_Toc8671810)

[6.7.4 Testing 56](#_Toc8671811)

[6.7.5 Nova 57](#_Toc8671812)

[6.8 Discussion 64](#_Toc8671813)

[7 Epilogue 65](#_Toc8671814)

[7.1 Conclusion 65](#_Toc8671815)

[Bibliography 66](#_Toc8671816)

**Chapter One**

# Introduction

This chapter defines the application being developed, the rationale behind choosing this problem and objectives to attain from it; along with the structure of this dissertation.

## Problem Description

End users vigorously use software applications in a different environment. Sometimes application gets crashed and unable to get started. It needs a whole system reboot or software reinstallation. This shouldn’t be the solution of any malfunction occurred. If end users have tech knowledge; strace could be one solution to check any system call that compels application to crash. Most of the time issue lies in the required resource being locked. These locked resources are the main issues. Applications use locks to synchronize tasks, but if exceptional handling isn’t done right the application couldn’t unlock the acquired resource in case of sudden kill or power failure. This issue makes application unable to get the same resource and then they couldn’t get restarted without the system reboot or sometimes need software reinstallation.

## Background

In multitasking application, threads/processes use locks to notify other tasks about the resource they are currently occupying. Sometimes the lock remains unhandled as the task had to stop abruptly or it was killed by the kernel. These stray locks compel the software to behave oddly. In many cases, the utility might not even start or needs a complete reboot. Sometimes the lock is permanent which needs extensive manual debugging before the user can use again the acquired resource. MYSQL Database is one example of it that uses file lock to acquire a socket which has, besides some plus points, more adverse effects. In case of system failure, the DBMS won’t get the chance to leave the resource, so in future, the file is already there, and system considers that socket already being locked, and not allowed to acquire that.

This is one example and there are documented and undocumented many examples to make oneself familiar with the complication that an incautious behavior of a programmer arises. To explore this problem furthermore, I have done manual testing on many software like Chrome, Opera, Firefox, PostgreSQL, SQLite-3, Level-DB, OpenStack, CloudStack, Apache-Hadoob, Apache-Tomcat, FUSE and Angular-JS in the course of developing this tool to programmatically find the troublesome problematic lock and pinpoint the function on stack that might be responsible for the exception. The testing results of the above software has been documented and will be explained later. Keeping in mind the problems and the efforts end user might be needing to put in, our tool will make rigorous disruption in the locking mechanism of software and show the developer exact spot where the problem is arising. It will help the programmer to check whether the application that is being developed has some potential unhandled locking exceptions that could affect the software performance in the future.

## Proposed Solution

Our solution comprises of tracing every system call the multitasking software of interest is making using ptrace and then handpick the futex system calls for further appraisal. The same futex call then nullified by kprobe kernel module, this will create a hypothetical condition in which software locking mechanism is robustly stimulated/tested which will generate different exceptions like resource unavailability, resource locked, etc. This makes the software prone to crash, which helps to do the crash analysis. When the software gets crash ‘libunwind’ could help in identifying the methods that result in a crash so, the programmer can do the due tweak if the application was malfunctioning.

## Aims and Objectives

Our solution aims to develop a diagnostic and debugging utility for Linux. It can be used to monitor the interactions like system calls, signal deliveries and process states between application and kernel. System administrators, diagnosticians, and trouble-shooters might find it useful for solving the issues where the source code isn’t available as the program’s recompilation isn’t needed. Further, it can be used to robustly check locks before the software release; that might be problematic in the future. And give the programmer a pinpoint location of code where he/she might need to do exception handling.

## Features

### Software Execution

Our software will have the ability to run other programs under its observation. The software utility needs to run the program in a traced environment in order to achieve the below features.

### Attaching to the running process

The software must have the ability to attach with the running process using its process id only. This makes it easier for the tester to test any running program at a certain time.

### Syscalls detection/tracer implementation

A diagnostic utility that can assure the communication between the kernel and the application needs to keep track of every system call it’s every thread is making. Our software needs to show the system calls along with the pid of the thread. These system calls could help the user in identifying the functionality that causes any problem.

### Lock Detection

As a debugging utility, our software will have an optional feature to robustly check the locks in order to identify the potential problematic situation that needs tweaking in code.

### Printing Backtrace

To identify the specific location where the code needs exception handling the stack trace will come to rescue. Without looking at the code programmer could see where the code demands to tweak.

## Document Structure

Next chapters in this dissertation will contain the discussion of the literature related to this project. Moreover, a fine requirement analysis will be put down to make implementation steps clear and final. After requirement analysis, a chapter comprising design phase where software’s intended design is discussed and following that implementation chapter discusses the implementation of the software. After Implementation we’ll cover testing of our final product this will also contain manual testing done before the implementation.

**Chapter Two**

# Literature Review

This part of the dissertation specifies the related work, projects, researches done in this domain and overview of the knowledge obtained. Moreover, it also states the need due to which this utility was made.

## Overview

In this chapter, we will cover the contents related to the literature we have followed or will follow over the journey of this development. We will start by discussing the problem from which we get the motivation from, following the tools that are doing similar but not specific struggle. In further going down to this chapter we will see kernel API’s that are going to be used in this utility’s development. In the end, we’ll see some related work in this field of study. Every fact and claim will be cited herein in the Harvard referencing style.

## Literature Comprehension

Our tool ‘crash analysis tool for profiling multitasking application’ owe us some explanation. Let’s start with demystifying the name to unfold the concepts.

It’s a profiling tool for multitasking application that is specifically designed for Software Crash Analysis. Profiling is a broader term used to define Dynamic program analysis. It is done by executing the program in a fabricated environment. The testing program is then subjected to different predefined inputs and to ensure the executing of a specific critical section of code has obtained by code profilers (Myers, 2004). Here profiling refers to statistical data collected from the software like; resource acquired, locks applied, waiting for processes, etc. We are specifically focusing on multitasking applications. Multitasking; as the name suggests, execution of multiple tasks at a time. It is a logical extension of multiple programming (Best, 2005). This takes context switching with timesharing and in this manner share common resources. Sharing common resources demands to lock to avoid race condition and to enforce mutual exclusion concurrency control policy (Li *et al.*, 2005). In all this hassle of multitasking, careless handling of locks leads to some serious problem and can cause the application to crash. In the following paragraph, we’ll see how this problem occurs and cause us to analyze that crash.

## Issue Identification

In multitasking systems, software application acquires a resource and lock it because other applications might use it or amend it as the resource is sharable. The application uses and unholds it to make things as is so other application waiting in the queue can use this resource. What if the resource is not being unlocked by the application that was using it; this will starve other application and in result, it will affect the overall performance. (Downey, 2016)

What if the application gets crashed while holding the lock over the resource or someone turns off the power source and the application in result loses its resources but when it turns on the application run the same code in which it makes a file that store the status of the resource: locked or unlocked. It’ll read the same file and in spite of the resource is available it will see that the resource is not usable and is being held by some other system utility. (Kaufman, 1981)

### Example

We have some example here of MYSQL database as it acquires socket and lock it by making a file (semaphore lock) which in result of crash; when gets start, sees the same file and read the resource status as locked and this is where the self-starvation starts, it has the resource available but it can’t have it because of the silly mistake -stray locks.

## Useful Tools Available

Tools to debug the condition discussed above are available and need robust understanding to operate. Worth mentioning are strace, lsof, lslocks and etc.

lsof is an infamous tool available to look at the locked files and their owners. Along with it, we have lslocks that lists the locking files and the process along with their owner that is locking it. Both of these tools can help us identify the problems related to a specific lock. And these tools are useful if you already know that there is a lock related problem (Padala, 2002).

What if you don’t know the problem then there comes a famous tool made for debugging and troubleshooting Linux utilities -Strace. It’s a command line tool similar to the above-discussed tool but has the capabilities like a swiss army knife. It discovers all the system calls to the kernel, all the interprocess communication, and can be used to tamper with them. It is most used when the source code, of the application being tested, isn’t available. You can have a thorough look on the program and see why it isn’t responding and why it is behaving in a similar fashion and why It was crashing because system calls do the math behind your source code and if you can see the system call and their return value you can cure the bug by just by seeing the error it returns. (Strace.io, 2019)

Each line in the trace contains the system call name, followed by its arguments in parentheses and its return value. An example from

stracing the command "cat /dev/null" is:

open("/dev/null", O\_RDONLY) = 3

Errors (typically a return value of -1) have the errno symbol and error string appended.

open("/foo/bar", O\_RDONLY) = -1 ENOENT (No such file or directory)

Here we can clearly see from the example, how we can debug our application.

The problem lies as it is the tool for pros and people with having readily understanding of what it is. Strace along with other tools can be useful when the mission is to find the problem and resolving it, but what if we can cure the problem before it arises.

## API’s

To cure the problem beforehand, we are going to see how these tool work in backstage to identify the problem. There comes the most famous API ever written for Linux -the ptrace system call. It is obvious from its name that it has something to do with trace whereas ‘p’ in its name specifies the process. In short, it’s the infamous tool used by developers, debuggers and similar software to trace a process in order to control it, inspect it and manipulate with the internal states of that process. strace in the backend uses this only system call to do all the deep digging. (Padala, 2002)

## Related work

Now we know the weapon all giants are using, and we have developed some hands-on understanding on it, but we have to see how others in this field are seeing the same problem and how they are trying to solve it. In the field of work, there is a trend of following non-blocking algorithms to write lock-free code this will help the coder to solve locking related problems like contentions, overhead, deadlock and convoying, etc. (Göetz and Professional, 2006). In addition to it, Languages providing libraries featuring concurrency-safe data structure. (Sun Microsystems, 2019)

## Related Work Limitation

Most of the big giant software have legacy code and problem that needs locks. To cure the performance overhead developers, require tools to identify the issues, that specific situations which are hard to recreate, may arise. Our tool could help the developer do a crash analysis of their software before shipping and do time analysis for performance tuning. This will help increase the product popularity in spite of the rival software and end user could happily embrace it with one less problem.

Chapter Three

# Requirements and Analysis

It’s what is being required as we have understood the problem statement and needs of this software. In this chapter, we’ll discuss the requirement that our utility must be capable of.

## Project Description

A Linux user-space utility uses to debug and diagnose any potential locking problem of the software provided. It has two main purposes; debug and diagnose.

The diagnose phase helps the user identify any potential problem of a malfunctioning product. Like if the application is prone to crash when request certain feature then the problem can be detected using our utility without looking at the code. Our utility detects every system call the software of interest is making and at the point where the application crashes the system calls along with its arguments could facilitate the debugger to tweak in some exception handling in the code. At the time of the crash, our software could get the stack trace for the debugger so that the tweaking part won’t be difficult.

The debug phase helps the developer identify potential bugs that could cause the software application to crash in a certain condition. To hypothetically generate certain scenarios our software might help in this case. It will robustly check locking mechanism that is infamously the cause of most of the problems and generate stack trace upon crash so debugger could easily modify the code where it needs to.

The application is a command line utility that either takes the name/path of the software that is to be tested or it’ll get the PID of a running process that needs to be traced. In any case, the application is then being traced by our software and every system call it does will be reported back to us. The functionality of diagnosing or debugging phase could be used with the help of tag introduces in command. The results can be seen on the terminal or a log file can be generated having all the system calls, and stack trace in case the software got crashed.

## Required Features and Analysis

Following are the list of features that’ll be included in the software.

### Software Execution

The software must be capable of executing any other application provided in the arguments. The application must be executed in the traced environment so every communication between the process and the kernel can be logged.

### Process Attachment

The software should be capable of attaching with any other process whose pid is provided as command arguments. After attaching the other process must be executed in a traced environment, so every communication must be gathered.

### Communication Logging

Any communication between the kernel and the traced process must be logged in order to attain the desired consequences. Every system-call, the process is making, every signal it is sending and everything the software of interest is doing must be logged.

### Potential Exception Detection

After the applications system calls are being detected the system call used to implement locking (FUTEX CALL) must be filtered out. A kernel module will then be invoked, and It will disrupt the locks, make the application sudden crash, irresponsive and then will test whether the resource acquire is being released or not. If not, then the lock is unmanaged.

### Crash Analysis

As the application was under test it must have behaved oddly. If the application got crashed the software must be capable of providing the user with the stack trace at the time of the crash. Stacktrace could be very useful in case of the crash as it could spot the functions last called before the application was about to crash. These functions could help the developer in handling any anomalies if the application couldn’t start to behave oddly after the crash.

### Report Generation

After all the detection and crashing, the software must generate a full-fledged report of all the communication it traced between the under-observation application and Linux Kernel. It includes system call name, its arguments, return value and the pid which is making that call as the software we are intended to test are multi-tasking. And if the application was being debugged the stack trace must be provided in the report in case any crash happen.

Chapter Four

# Design and Implementation

Our feature-rich utility must get in the development phase, where it has its functionality implemented. Most of the features have been described earlier, and now their practical implementational structure will be discussed along with the coded subtleties; which explains the ways that we have adopted to solve the problem described previously. This part of the document is where we would illustrate how we have programmatically planned a solve the problem.

## Overview

Devising a programmatically productive solution is discussed in this chapter. Our functional requirement indorses a terminal utility which eradicates the user interface from the equation. The terminal application helps the developer test the program which just the ease of command and some flags. The responsibility of the software is to first observe the system level calls and find those specific calls that handle locking mechanism which then further disrupted with the help of the kernel module. After the call disruption the software that was being tested will be further tested and this time rigorously as it not protecting the critical sections or cannot acquire them. At this stage the result will be collected which help us look at the software’s behavior; whether it gets crashed or able to maintain its existence. In case of a crash, a report would be generated identifying the starving process, the locked resource and the stack trace leading to the crash.

## Architecture and component

Our userspace utility is being written in C-language. The reason in choosing this language is that the kernel libraries are mostly provided in C language. And to blend the code at a deeper level kernel’s language is being used. Further to use libraries intended to provide the features above explained demands c language to compile. Design architecture for this product is files and header based. Each code file comprises of a separate functionality and every code file has a separate header file to work with. Each header file has the signature of methods that needs to be written in the code files. Interfile communication is done by including a specific header file to the code files that want to use those methods. The compilation phase is done with the help of a MAKEFILE. To compile this code a make file must be written that compiles all the files and clears all the dependency needed along the line. These make file will help developer compile code easily and generate an executable file.

### Functional Design Identification

To build the software, one needs to know what kind of functionalities that are required and how they need to be implemented in the desired language of code. What are the types of limitation it generates and how we are going to cope with it?

#### Main File

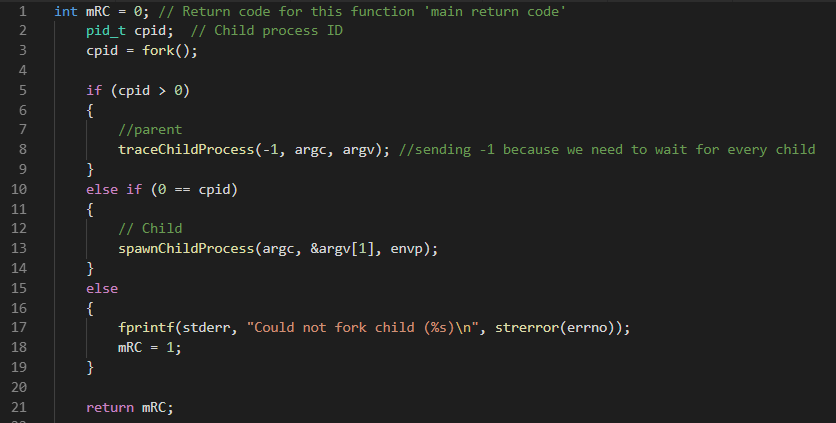
The first function our compiler may call described as following in design and implementation steps

##### Design

A must have a file that starts the code will be written that should fork the process that executes the program that needs to be traced and would call a method that traces the forked program.

##### Implementation

Following code, snippet shows the main functionality that’s been applied via design description.



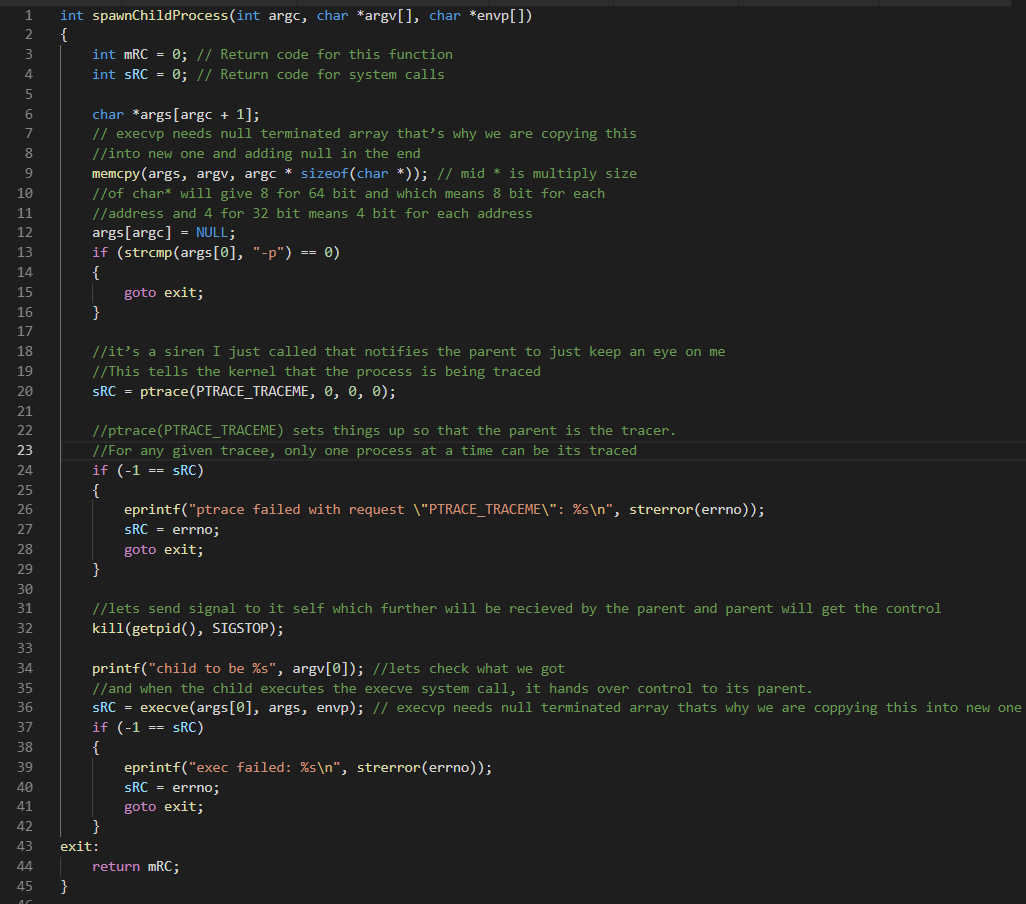
#### Spawn Process

##### Design

A code file that’s responsibility is to exec the program that needs to be traced or attach to the process that demands consideration. This code file doesn’t have much responsibility to exec a new process and ends.

##### Implementation

This part of the implementation starts a new process and calls PTRACE\_TRACEME and notify waiting for the parent process.



#### Trace Spawn Process

##### Design

The main code of our program must be written here. It is the file that waits for the program to spawn and start tracing it. Ptrace is the library used for this purpose as it the kernel implemented system call. A wizardly system call that empowers its beneficiary process to manipulate other processes. It’s a swiss knife of a debugger, which is having all the tools of controlling the executing process along with peeking and poking the vitals. It provides a mechanism by which you can write a program that can attach to a running process as a parent and then can-do certain things like, stepping through the code one line at a time, set breakpoints, have access to the entire address space, stack, and registers, patch running code of the target etc.

From the man page of ptrace()

The ptrace () system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

In order to get manipulated, the Tracee must get attached to its tracer. Each tracer can get attached to only one thread of a multi-threaded application/process. It takes a magic request (from the list of enums) in its first parameter and doing completely different things depending on its value.

**Ptrace Prototype**

In order to include use the ‘#include <sys/ptrace.h>’

Syntax of the call.

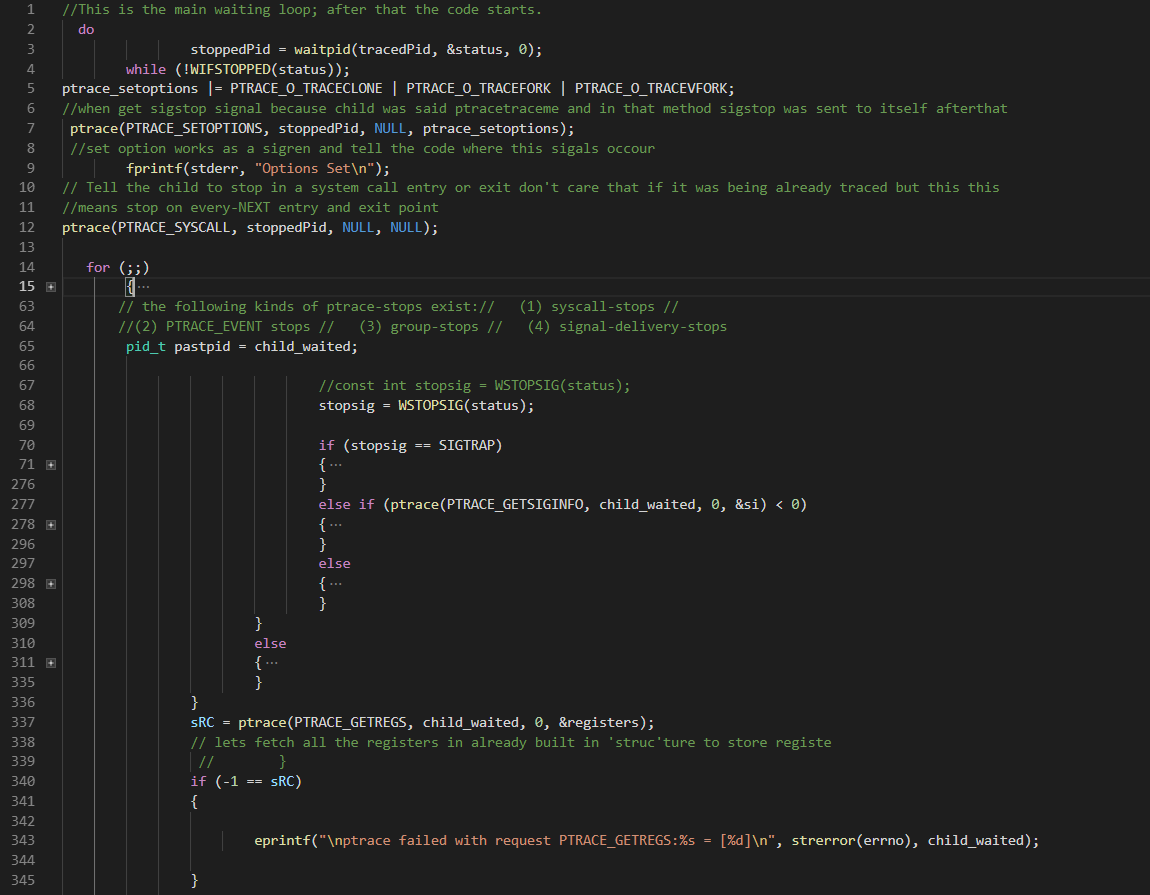
long ptrace(enum \_\_ptrace\_request request, pid\_t pid, void \*addr, void \*data);

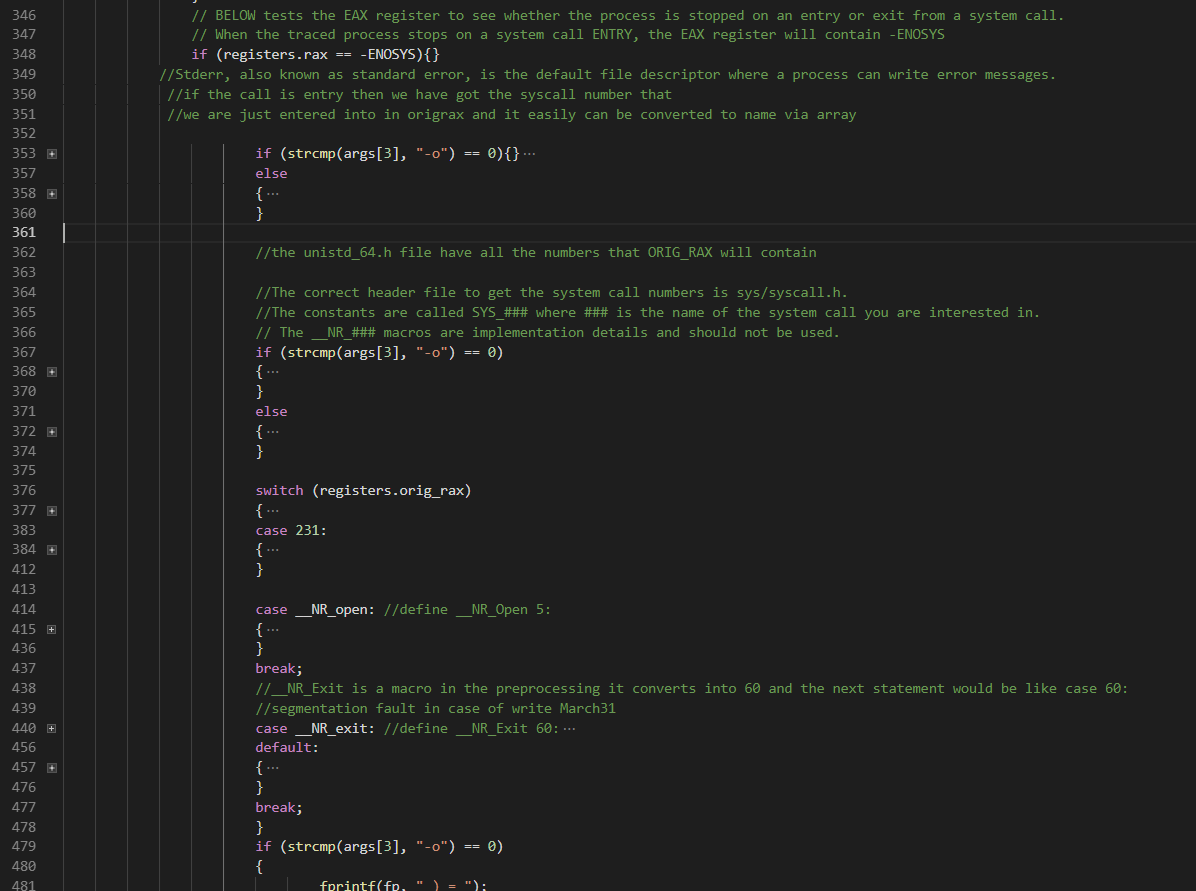
* Request augment decides what is to be done.
* Pid is the process ID of the process that is to be traced.
* Addr is the offset in the user space. It’ll be used on the basis of the request.
* Data something that is to be written to the tracee, if told so. It depends on the request argument.

##### Implementation

It will be attached with the desired executed program and the manipulation of that program will be starting. First, the options will be set with ptrace\_setoptions this ensures that the kernel will notify us about any options that we have requested. Example of this is PTRACE\_O\_TRACECLONE this will explicitly tell the kernel to notify with clone signal when the traced program clones a new process. There are other options that can be looked after in order to add functionalities in the code. After options setting, we will use the same API to tell the kernel that notify us at every entry and exit of a system call using PTRACE\_SYSCALL argument for ptrace.

This code file is now responsible to gather the communication between the process and the kernel, in this case, an infinite loop is started that will wait for the stop signal as we have told the kernel to send us a signal each time the system call in entered or exited. After the stop, we must see what is the signal that we have and after which event we have got this. If the event was fork then we will repeat the above steps like setting option and Syscalls entry-exit notifier. But if the event defaulted then we will see in the ENOSYS register whether this is system call entry stop or exit. If the stop is entry we will explore the orig\_rax register to explore the system call number. A data file is needed here that matches the system call number with the system call name. We will then explore another register for the arguments. In the next iteration, we’ll do the same and this time check for system call exit to get the return value.





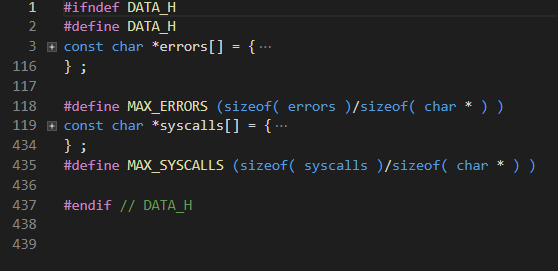
*Disclamation: Above code, the snippet is the minimalistic version of the basic command to describe in this document.*

#### Data File

##### Design

This must be the file that contains all the data our code needs to check each time an error occurs, any system call detects, or any signal captures. This file is a kind of a database but efficiently fast.

##### Implementation



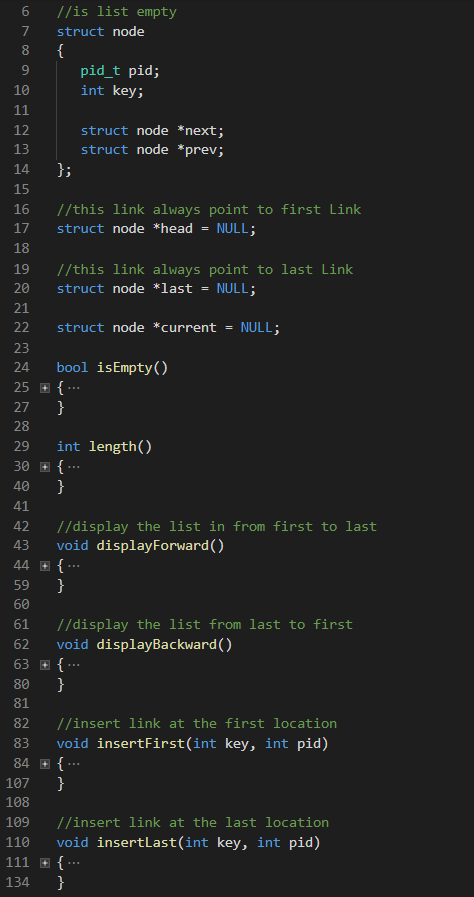
#### Data storing

##### Design

A list must be implemented that is responsible to store the pids of each newly spawned process. This will help in identifying the system call and to generate the process wise stack trace when an anomaly occurs.

##### Implementation

This double linked list helps to store the pids of each spawned process



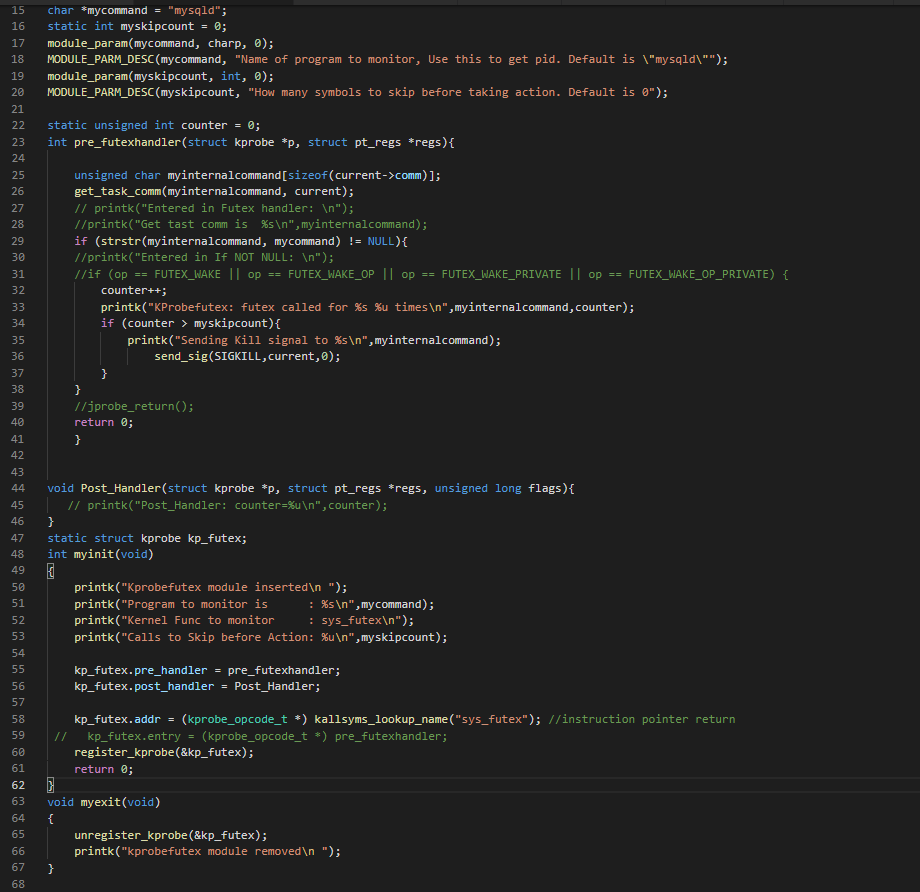
#### FUTEX handler

##### Design

A kprobe futex handler will be written that infuses into the kernel at the start of the software and kill each futex system call when every it gets detected after the preassigned skip count.

##### Implementation

Kernel probes help debugger collect performance information, add functionality at runtime without the need of kernel compilation. The developer may add break-point instruction that may invoke the module upon any trigger. To write such kernel probe one should write a kernel module that will have a init and exit routine.

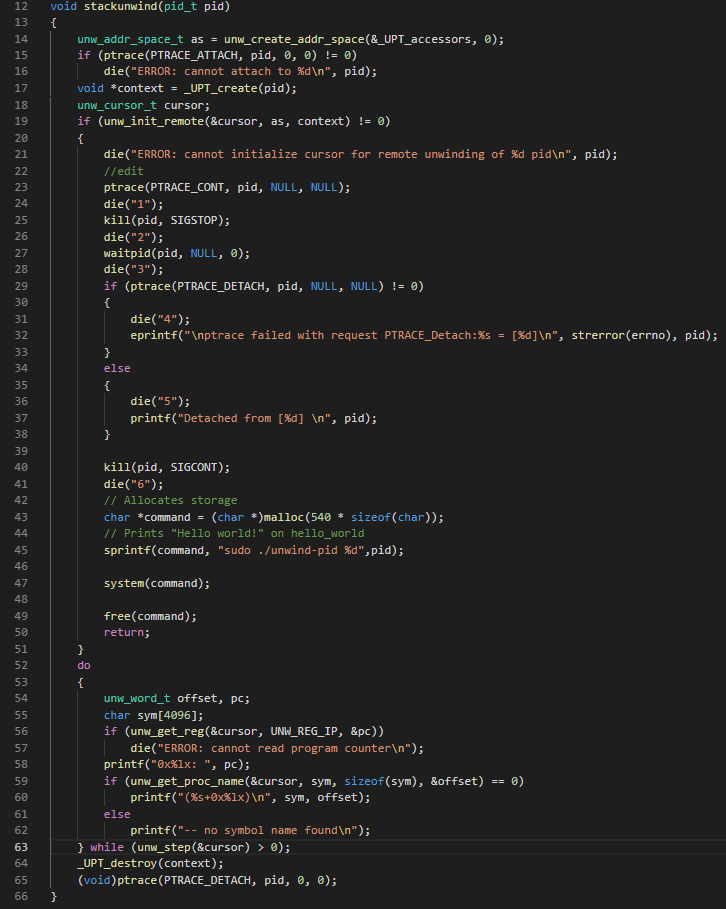


#### Stack trace

As the software crashes, the programmer needs to know what method results in the crash and where it needs to tweak in order to do exceptional handling. This requirement needs libunwind library that provides a stack trace of a remote process.

##### Implementation

Libunwind’s ‘remote unwinding will be used in this case. A new address-space object will be created for the process that is to remotely unwind. For this purpose, a library method known as *unw\_create\_addr\_space*() will be used. This method takes two values: firstly a pointer to adjust the accessors, secondly an integer that notifies the byte order of the process that is to be explored. Further, there are methods to check and edit the memory or registers of the given process at any instance. After the creation of address space unwinding can be generated by a routine call to *unw\_init\_remote*(). This very call takes address space object as an argument and an opaque pointer. This method is used to extract the memory state. After all, a cursor needs to be instantiated with *unw\_init\_remote*(). Now different methods can be used to move up and down in the stack.



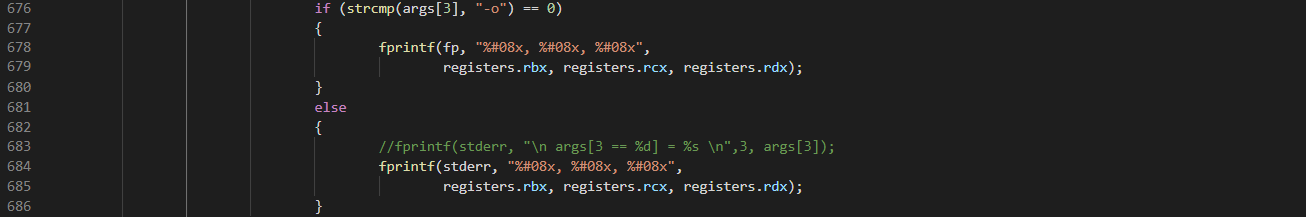
#### Report Generation

##### Design

Every information that can assist the developer to debug the code will be printed on the terminal. To save all the information of the terminal the user can give -o as an argument followed by a name. A file will be generated in the /home directory having all the outputs like stack-trace, locked resources, starving process, etc.

##### Implementation

A file pointer has been created and in the whole code and fprintf method is used to print every relevant information in the designated file.



### Flow diagram

As we zoom in into the software, we see multiple methods implementing functionalities which a user will be needing. It shows how the code structure must be implemented in the back end to generate the requirement of the software

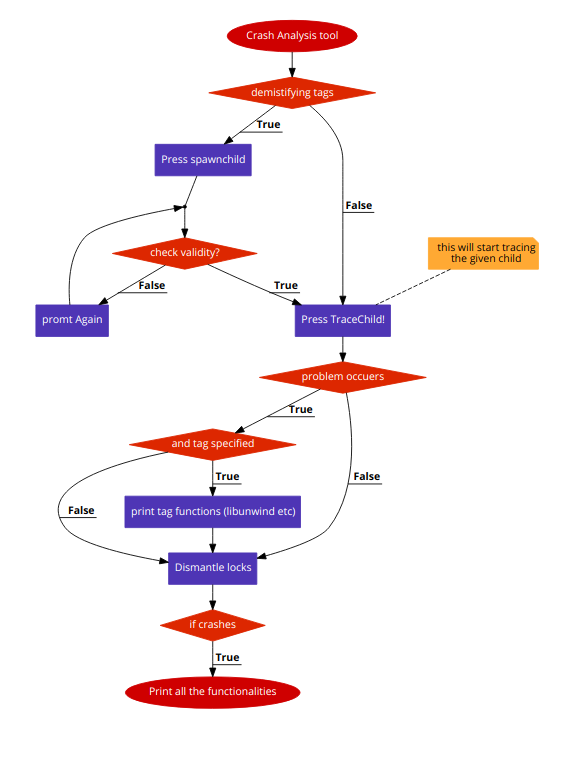
This flow chart encompasses all the classes we are going to need in order to generate user specified results

Figure 4.1

## Non-Functional Design Description

The nonfunctional requirement consists of non-technical features that constraints on how the system would work. Our system will be used to check whether the developers have formulated a technique that infers a way of avoiding crash when a requested resource cannot be acquired. This is the terminal utility so, it has no interface. The terminal application would start with the command ‘crashtrace’ followed by the arguments for each function as described above.

### Platform

This utility is capable to operate on every 64-bit Linux distro as it will be based on POSIX structure. Our application will be installed in the operating system with a single command and then will capable to perform with full functionality.

### Scope

This app is capable to test any Multitasking software and other software work on a single thread. Any software runs in a Linux environment can be subjected to ‘crashtrace’ test.

### Output

In case of a crash, the application’s stack trace along with starving process’s details and locked resource description will be published on a text file that assists the developer in debugging and handling the crash.

## User Interface

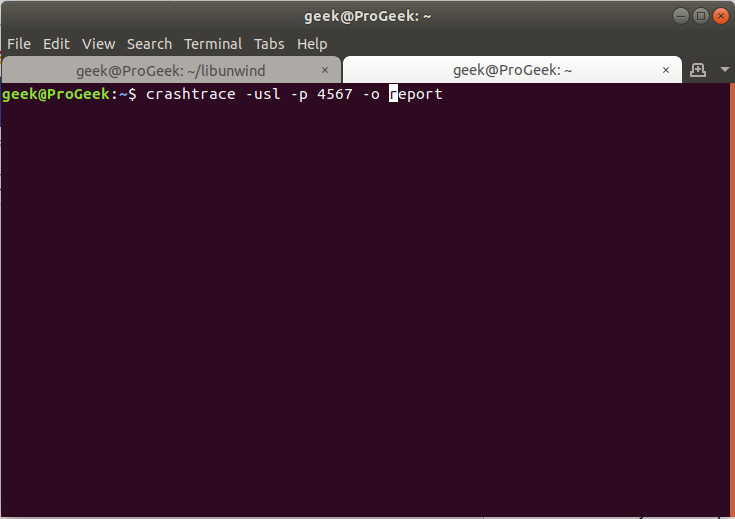
How to interact with our application is stated here. The user would only need to run the command ‘crashtrace’ with subjective flags that describe the user’s requirement.

Figure 4.2

### Flags available

Flags aid the utility to function according to the user requirement.

-p: if the intended software is already running then this tag will be used with pid of the process

-o: to print all the log and details into a separate file -o tag will be used. It outputs every information into the code file.

-k: to do debugging the software needs to be tested with locks. This tag will load the kernel module is used. It must be used with the name of the software which futexes is to be tested and an integer value that signifies the skip count.

The flags will be acted upon with respect to the possibility as many application has restricted access.

### Program to Test

The program can be passed as an argument so it can be started by the crashtrace or an already running application can be passed by its PID number using -p conventional flag following the PID.

### Report generation

All of the results get printed at the terminal if -o flag does not use. -o flag following a name helps the utility generate a file and pipe all the output to it so the user can have access to it after the application gets closed.

Chapter Five

# Testing

This chapter holds a significant value in the development of this project. There are three main types of the test applied to test this product. Usability testing, Functional testing, Unit testing. All three have helped us in the development of this tool that could help programmers easily test their software.

## Overview

A product is incomplete when it is not well tested. We did three types of testing with the software developed and we’ll explain how each has its individual benefits. In the end, well explain how this product can’t be programmatically unit tested as the results of any given command could provide different results each time as the product is dependent upon other software (it test other software) which provides unexpected but understandable results. This makes it difficult to test the product but different tricks have been applied to get the desired results.

## Usability testing

As the product would be of use of the developers. We have been asking developers to use this product to test their software malfunctioning behavior before any manual debugging. Like strace, they have learned to use this tool with ease and was a very enthusiast in testing their products and finding potential bugs.

Following were the questions we told them to apply.

* Running their application in ‘crashtrace’ environment.
* Apply to lock disruption mechanism and see if their application crashes.
* Were they able to restart the application? If yes then was it running perfectly and if no then was the stack trace helpful.

Below table Shows the responses from the four users.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| USERS | Crashtrace | Crashed | Recoverable | Stacktrace | Comments |
| 1 | Ran Successfully | Yes | YES | N/A | Locks were applied correctly |
| 2 | Ran Successfully | Yes | YES | N/A | Locks were applied correctly |
| 3 | Ran Successfully | No | N/A | N/A | Locks were applied correctly |
| 4 | Ran Successfully | Yes | No | Helpful | File locking is used and exceptional handling wasn’t done. Stacktrace helped in identifying the routine. |

Table 5.1

Their response backs my stance of developers using locks without any possible overseeing.

## Functional Testing

This type of testing signifies the importance of individual functional features. In this type of testing, we seek the output as it needs to be. In our case, we run other software in the traced environment, and we get different output in every case. For this type of testing, we wrote different test ‘HelloWorld’ programs that ensure that the output results must be the same. And to check whether the result is reliable we test the program against our code and then with another potential program or intuitions.

### System Calls Test

We ran our written test program in this software and then compares the output against the results given by strace on the same test program. Every time the results were the same that signifies the reliability of our software.

### Futex Kill Test

After detecting reassuring system calls we needed to differentiate the futex calls and start the application kill those. We wrote a kernel module and test it efficiently on different software products. The results could be seen by ‘dmesg’ as kernel level printing. The kernel module was working just fine on every software in every condition. This module is being individually loaded on the start and unloaded when the application is closed. This type of module shouldn’t be loaded in the kernel forever as it could create problems.

### Stack unwinding

The same approach of system calls testing is used and in this case, the function is written which needs a pid to unwind the program. The routines displayed by the libunwind function has the same routines written in the test program.

## Unit testing

This type of product that generate different results at every run cannot be implemented by any framework that specifies the unit test. Every Unit is interrelated, and its result is dependent on the software it runs. The software under scrutiny generates different results as their code isn’t available so their behavior cannot be predicted. So unit testing is futile in our case.

Chapter Six

# Results and discussion

This chapter contains the achievement of our software, results in it produces and along with critical discussion related to it. This chapter access how significantly our software helps in testing other software. We will discuss how our software fulfills the requirement of the end user who wants to test its product.

## Overview

We will test this application on certain another important set of software and their results will be explained. We’ll also see how these applications react in that traced condition and how they recover from a crash. We also explain if that application is unable to restart then what could be the potential issue behind it. If the issue persists, we’ll explain how a programmer can do exception handling in order to solve it.

## Testing Software

Below is some software that has been tested using the tool created.

## PostgreSQL Database Server TPC-C testing using Hammerdb

### HammerDB

HammerDB is the best performing database benchmarking and load testing tool for PostgreSQL and other databases with built-in benchmarks based on the industry standard TPC-C and TPC-H specifications and native builds for Linux and Windows.

### TPC-C

TPC Benchmark C is an online transaction processing (OLTP) benchmark.TPC-C involves a mix of five concurrent transactions of different types and complexity either executed on-line or queued for deferred execution. The database is comprised of nine types of tables with a wide range of record and population sizes. TPC-C is measured in transactions per minute (tpmC). While the benchmark portrays the activity of a wholesale supplier, TPC-C is not limited to the activity of any particular business segment, but, rather represents any industry that must manage, sell, or distribute a product or service.

### Testing

Our testing doesn’t demand database server benchmarking rather It’s a general test which checks the futex calls the server made and what happen when we try to kill the calls during multiple processes. Usage of hammerdb was to give an almost real scenario to the server so we can see the futex calls in the backend. So there is no need to make a reference to hammerdb’s result but we will definitely discuss the scenario under which the database server was tested.

#### **Test One – Schema Build**

Test one comprises of loading schema and raw data into the database. We used 10 warehouses and 4 virtual users to build the schema.

#### **Test Two – Driver Script**

Test two comprises of driver script that makes rigorous queries according to tpc-c standards. We used timed driver script for this testing. Two minutes were allotted for ramp-up time and five minutes for the test duration. A number of virtual users for this process were five and total transactions per user were 200000.

### Schema Building

#### Test Case One

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Not Loaded | 4 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | No | 1 |

##### Comments

Everything was perfect.

#### Test Case Two

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 0 skip count before building schema | 4 | 10 |

##### Results

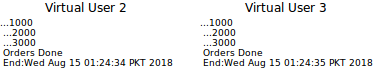
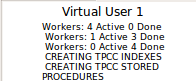
|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | Yes: Error in Virtual User 1: SSL SYSCALL error: EOF detected | 0 (the single futex call was killed) |

##### Second Try

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes (all the data was loaded) | Yes: Error in Virtual User 1: SSL SYSCALL error: EOF detected | 0 (the single futex call was killed) |

##### Comments

Everything (data) was in the database and nothing strange was observed.



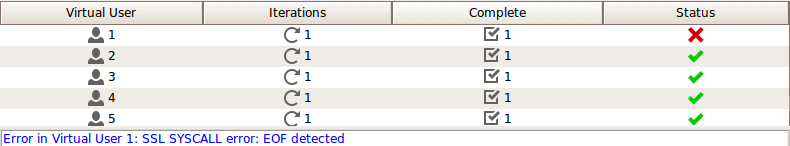


Figure 6.1

stats of the rest of the two users were the same.

#### Test Case Three

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 0 skip count After 5 seconds of running ‘build schema’. | 4 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| No | Connecting to HammerDB Agent @ localhost:0  Testing Agent Connectivity...SocketError  Connection failed verify hostname and id @ localhost:0  Error in Virtual User 1: SSL SYSCALL error: EOF detected | N/A |

##### Second Try

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes (all the data was loaded) | Yes: Error in Virtual User 1: SSL SYSCALL error: EOF detected | 0 (the single futex call was killed) |

##### Comments

Same as discussed in case two everything (data) was in the database and nothing strange was observed.

#### Test Case Four

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 1 skip count before building schema | 4 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | No | 1 |

##### Comments

Everything was perfect.

#### Test Case Five

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 1 skip count After 5 seconds of running ‘build schema’. | 4 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | No | 1 |

##### Comments

Everything was perfect.

#### Test Case Six

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 2 skip count before building schema | 4 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | No | 1 |

##### Comments

Everything was perfect.

#### Test Case Seven

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 2 skip count After 5 seconds of running ‘build schema’. | 4 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | No | 1 |

##### Comments

Everything was perfect.

### Pattern

Same results were observed for test case eight, nine, ten and eleven. Eight and nine were given skip count 10 whereas ten and eleven were given skip count 100. The scenario and the results were the same even the number of futex calls were consistent that is One.

### Driver Script

#### Test Case One

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Not Loaded | Total 5 (One Monitor and 4 Virtual Users) | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | No | 4 |

##### Comments

Everything was perfect.

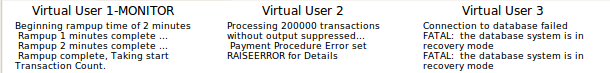
#### Test Case Two

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 0 skip count before running virtual users with driver script | 5 | 10 |

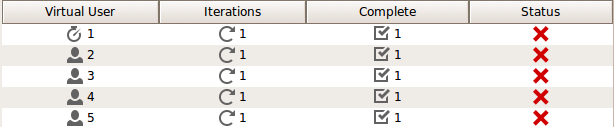
##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| No | Error in Virtual User 2: no connection to the server  Error in Virtual User 3: Connection to database failed  FATAL: the database system is in recovery mode  FATAL: the database system is in recovery mode  Error in Virtual User 4: Connection to database failed  FATAL: the database system is in recovery mode  FATAL: the database system is in recovery mode  Error in Virtual User 5: no connection to the server  Error in Virtual User 1: server closed the connection unexpectedly  This probably means the server terminated abnormally  before or while processing the request. | 0 |

##### Comments



**Figure 6.2**

**Figure 6.3**

#### Test Case Three

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 1 skip count before running virtual users with driver script | 5 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| No | Error in Virtual User 2: no connection to the server  Error in Virtual User 3: Connection to database failed  FATAL: the database system is in recovery mode  FATAL: the database system is in recovery mode  Error in Virtual User 4: Connection to database failed  FATAL: the database system is in recovery mode  FATAL: the database system is in recovery mode  Error in Virtual User 5: no connection to the server  Error in Virtual User 1: server closed the connection unexpectedly  This probably means the server terminated abnormally  before or while processing the request. | 1 |

##### Comments

N/A

#### Test Case Four

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 2 skip count before running virtual users with driver script | 5 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| No | Error in Virtual User 2: no connection to the server  Error in Virtual User 3: Connection to database failed  FATAL: the database system is in recovery mode  FATAL: the database system is in recovery mode  Error in Virtual User 4: Connection to database failed  FATAL: the database system is in recovery mode  FATAL: the database system is in recovery mode  Error in Virtual User 5: no connection to the server  Error in Virtual User 1: server closed the connection unexpectedly  This probably means the server terminated abnormally  before or while processing the request. | 2 |

##### Comments

N/A

#### Test Case Five

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 3 skip count before running virtual users with driver script | 5 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| No | Error in Virtual User 2: no connection to the server  Error in Virtual User 3: Connection to database failed  FATAL: the database system is in recovery mode  FATAL: the database system is in recovery mode  Error in Virtual User 4: Connection to database failed  FATAL: the database system is in recovery mode  FATAL: the database system is in recovery mode  Error in Virtual User 5: no connection to the server  Error in Virtual User 1: server closed the connection unexpectedly  This probably means the server terminated abnormally  before or while processing the request. | 3 |

##### Comments

N/A

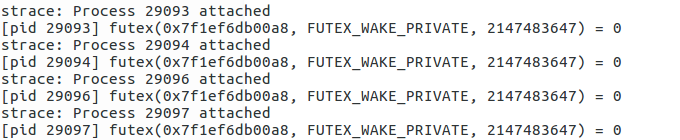
#### Test Case Six

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 4 skip count before running virtual users with driver script | 5 | 10 |

###### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | No | 4 |

###### Comments

****

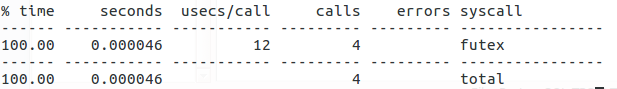


Figure 6.4

#### Test Case Seven

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 10 skip count before running virtual users with driver script | 5 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | No | 4 |

##### Comments

N/A

#### Test Case Eight

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 100 skip count before running virtual users with driver script | 5 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | No | 4 |

Comments

N/A

#### Test Case Nine

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 0 skip count After 3 sec of running virtual users with driver script. | 5 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | No | 0 |

##### Comments

The 3 seconds we delayed is the time, I presume, Hammerdb requires to make the connection of virtual users with the database server.

#### Test Case Ten

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 1 skip count After 3 sec of running virtual users with driver script. | 5 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | No | 0 |

###### Comments

N/A

#### Test Case Eleven

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 2 skip count After 3 sec of running virtual users with driver script. | 5 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | No | 0 |

##### Comments

N/A

#### Test Case Twelve

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 5 skip count After 3 sec of running virtual users with driver script. | 5 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | No | 0 |

##### Comments

N/A

#### Test Case Thirteen

|  |  |  |
| --- | --- | --- |
| Kernel Module Jprobefutexwakekill | Virtual Users | Warehouses |
| Loaded with 100 skip count After 3 sec of running virtual users with driver script. | 5 | 10 |

##### Results

|  |  |  |
| --- | --- | --- |
| Finished | Errors/Abnormalities | Futex calls |
| Yes | No | 0 |

##### Comments

N/A

### Pattern

Four Virtual users only need 4 futex calls to make connections with the database server. No system futex calls were made while the driver script was running. I tested with five virtual users and saw Five futex calls in the summary of strace.

## Apache Tomcat

Apache Tomcat (or simply Tomcat) is an open source web server and servlet container developed by the Apache Software Foundation (ASF).

### Testing Pre-requisites

#### Installation

Installation of Tomcat and setting apache tomcat’s configurations.

#### Sample Servlet

Sample servlet can be installed from [here](https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/).

### Pre-testing Info

Inserting the module requires a name. Since tomcat is a Java-based application it is started using java binary, so always ps will show it as a java process. Adding the name of tomcat in mycommand attribute of kprobefutexwakekill is not possible else we have to give “java” in the kprobefutexwakekill’s mycommand attribute. Worth mentioning that even when you install application servers like WebSphere or WebLogic they too are java programs so Linux will identify them as java process only because that's how java virtual machines are identified. Although in server.xml of WebSphere there is an option to change the executable name to whatever you like which is java by default. I didn’t come across this sort of setting for tomcat.

### Test Summary

It needs 3000 plus futex calls to start and if any of it gets kill by our kprobefutexwakekill module then tomcat won’t start. Operating the sample and example apps don’t make any calls.

### Testing Detail

Inserted module with count 50 while tomcat was running and when I did ‘*dmesg*’ to check I got 50 calls that were skipped and after some time 25 more counts that were killed. Surprisingly the tomcat server was running fine in the backend and I tried to refresh the sample website that I hosted locally on tomcat and it was working fine and soon after 5 minutes, I see 5 more calls that were killed by our module which made the tomcat server die.

Tried this again with more than 5000 skip counts. At first, killed the tomcat server via ‘Kill –9 pid’ and restarted it again. Recorded 5000 calls in a minute and after it reached the skipping threshold we got recorded 66 calls that were killed by kprobefutexwakekill module. I got surprised because the tomcat was running in the background. Removed the module and inserted it again and the recorded no futex calls.

Restarting needs 3000 calls to and if any of it gets killed the program won't start but if we remove the module and then just observer we see that the program tomcat service automatically starts with almost 1500+ more calls.

Then ran all the servlet examples given at <http://localhost:8080/examples/servlets/> and found no further futex calls.

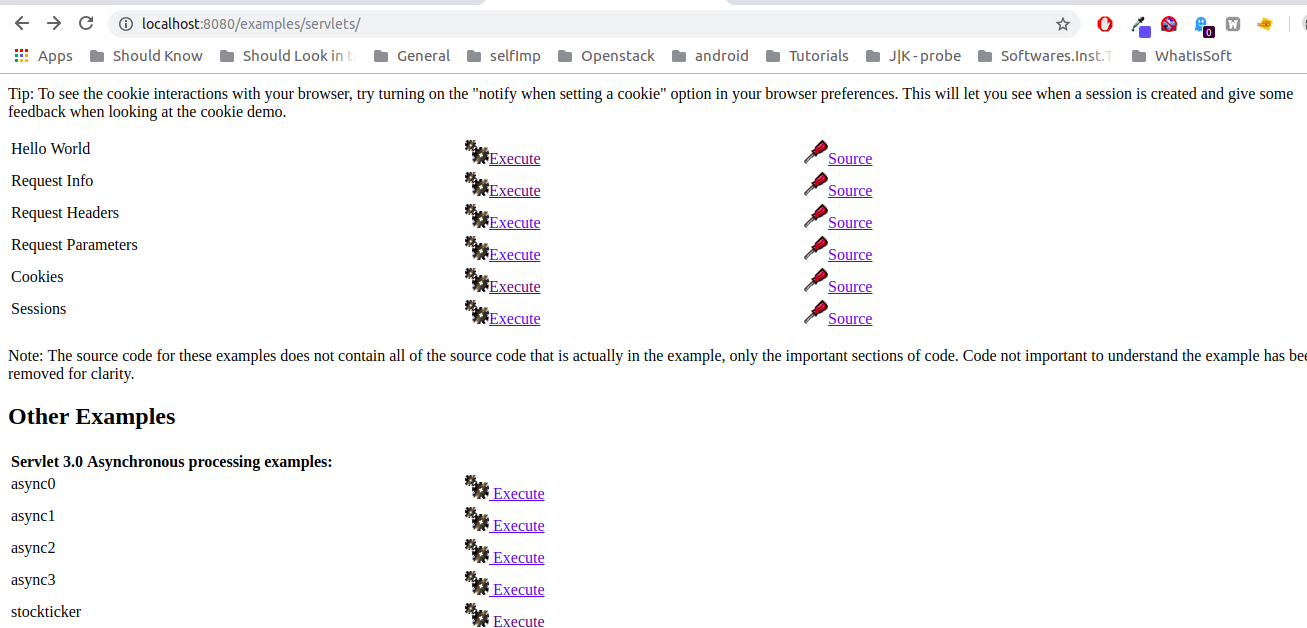


Figure 6.5



Figure 6.6

### In Further Iteration

I have written a custom servlet that store and read data from the database. I have tested this site and got the same results as above. In the next phase, I would test tomcat while benchmarking that site.

## FUSE

FUSE is the Filesystem in Userspace, an operating system extension that allows non-root users to create mountable filesystems. FUSE is available for most UNIX-like operating systems, including OS X.

### Testing Pre-requisites

To test fuse we needed a model, so we chose SSHFS. It is a [FUSE](http://en.wikipedia.org/wiki/Filesystem_in_Userspace) filesystem uses the [SSH File Transfer Protocol](http://en.wikipedia.org/wiki/SSH_File_Transfer_Protocol) (SFTP) as it’s backend. The short of it is that you can mount a remote directory on your local machine with nothing more than SSH access.

### Test Summary

SSHFS; a fuse utility used to access files remotely. It makes futex calls on every step and when any of the calls get killed our connection is aborted with an error that is Transport endpoint is not connected.

### Testing Detail

To start the testing, we inserted the module with the skip count of 500. At first, we observed 6 futex calls and then a minute later 4 more. We mounted the remote directory with the following command.

*geek@linux: sshfs -o allow\_other* [*keeg@172.19.0.4:/*](mailto:keeg@172.19.0.4:/) */mnt/droplet*

*Where 172.19.0.4* is the address of the remote server and */mnt/droplet* is the mount folder.

We observed 16 calls when opened the mnt folder and 474 calls when opened the main folder named as a droplet.

*At this point, our kernel module had started probing because the skip count limit was exceeded. When tried to open the folder home we recorded 6 calls that all were killed by our module and we were only left with the following error.*  
  
error opening directory, /mnt/droplet/home:Transport endpoint is not connected

We tried to access the droplet folder and it was empty. We tried to remount it using the same command but instead of mounting the remote directory we got the following error.

fuse: bad mount point `/mnt/droplet': Transport endpoint is not connected

So, we needed to resolve this error by properly unmounting the remote drive. We used the following command

sudo fusermount -u /mnt/droplet fusermount is a program to unmount FUSE filesystem -u is unmount

If we kill any futex call, we just get Transport endpoint is not connected. It doesn't matter whatever call it is. It could be the first call or five thousand calls.

For testing we opened different folders and files and when the module kills the futex call, our connection gets aborted and then we can’t load the remaining file and cannot be able to access the remaining folders, but already opened images and files are accessible.

## Web Browsers

### Opera

Opera is a web browser for Windows, macOS and Linux operating systems developed by Norwegian Company Opera Software AS. It uses the Blink layout engine.

#### Testing Pre-requisites

We tested Opera against our Kernel module and recorded the results. In this module, there is a parameter myskipcount which means that how many Futex calls we want to skip. We tested opera by giving values of myskipcount from 0 to 1000 and observed the effects.

#### Test Summary

Opera web browser uses a lot of futex calls for its functionality. For properly working Opera makes

continuously futex calls. You can see this by filtering futex calls by using strace utility.

strace strace -Tf opera 2>&1 | grep -v futex

and even if a single futex call of opera gets killed whole opera functionality got effected and opera crashed or do not work properly.

#### Testing Detail

Below table is showing results we got during testing. We tested opera on the same skip count twice

1. First loaded module and then started opera

2. First Opera started and then the module inserted.

|  |  |  |
| --- | --- | --- |
| Scenario | Skip Count | Result |
| Opera not running | 0 | Opera did not even start |
| Opera running | 0 | Crashed instantly |
| Opera not running | 100 | Opera did not even start |
| Opera Running | 100 | Crashed Instantly |
| Opera not running | 1000 | Starts but got killed instantly |
| Opera running | 1000 | Was working for 10-15 seconds |
| Opera not running | 1500 | Starts but instantly killed |
| Opera running | 1500 | Worked perfectly for 5-10 seconds |
| Opera not running | 2500 | Starts and stay alive for 4-6 seconds then killed |
| Opera running | 2500 | Worked for 1 minute and then crashed |
| Opera not running | 5000 | Starts and work perfectly for 1 minute |
| Opera running | 5000 | Works for 1-2 minutes |
| Opera not running | 7000 | Run and work for 2-3 minutes |
| Opera running | 7000 | Loaded youtube video and running for 1-2 minutes |
| Opera not running | 10000 | Run and work perfectly for 2-3 minutes |
| Opera running | 10000 | Run a video for 2-3 minutes and then crashed |

Table 6.2

### Firefox

#### Test Summary

The Firefox browser has done rigorous exceptional handling. Every time the browser gets killed it starts again with all the functionalities and recent browsed data.

|  |  |  |
| --- | --- | --- |
| Scenario | Skip count | Results |
| Firefox is not running | 0 | Instant kill: doesn’t even start |
| Firefox is running | 0 | Instant kill |
| Firefox is not running | 100 | Instant kill |
| Firefox is running | 100 | Instant kill |
| Firefox is running | 1000 | Instant kill |
| Firefox is not running | 1000 | Doesn’t even start |
| Firefox is running | 3500­4000 | Remain open for almost 10­20 sec |
| Firefox is not running | 3500­4000 | Remain Open for about 15 secs and crashes |
| Firefox is running with multiple tabs | 7000 | Remain open for 20+ sec |
| Firefox is not running | 7000 | Remain Open for about 15 secs and crashes |
| Firefox is running with youtube | 10000 | Opens for a while like for 40­50 sec. |
| Firefox is not running | 10000 | Opens but closes instantly after 25 sec |
| Firefox is running | 25000 | Opens for 2­3 minutes. Every button and the option was working correctly. Watched a video on youtube then it suddenly crashed |
| Firefox is not running | 25000 | Opened multiple instances, watched a video pressed buttons and after almost 2 minutes it crashed |

Table 6.3

### Chrome

#### Test Summary

The Chrome browser has done rigorous exceptional handling. Every time the browser gets killed it starts again with all the functionalities and recent browsed data.

|  |  |  |
| --- | --- | --- |
| Scenario | Skip count | Results |
| chrome is not running | 0 | Instant kill after 1 futex call: doesn’t even start |
| chrome is running with 5 tabs | 0 | Instant kill after 21 futex calls |
| chrome is not running | 100 | Instant kill after 102 futex call |
| chrome is running | 100 | Instant kill after 112 futex call |
| chrome is running | 1000 | Instant kill after 1012 futex call |
| chrome is not running | 1000 | Doesn’t even start kill after 1042 futex calls |
| chrome is running | 3500 | Remain open for almost 5 sec killed after 3506 futex calls |
| chrome is not running | 4000 | Remain Open for about 15 secs and crashes after 4103 futex calls |
| chrome is running with multiple tabs and a youtube video | 7000 | Remain open for 20+ sec after 7007 futex calls |
| chrome is not running | 7000 | Remain Open for about 15 secs and crashes |
| chrome is running with multiple instances with multiple tabs but with no external activity | 10000 | Remained open for 5 minutes then I tried to open youtube and it suddenly crashed. |
| chrome is not running | 10000 | Opens for a while like for 20 sec and crashed after 10003 futex calls |
| chrome is running | 25000 | Opens for 5 minutes. Every button and the option was working correctly. Watched a video(4 min) on youtube then it suddenly crashed |
| chrome is not running | 25000 | Opened multiple instances, watched a video pressed buttons and after almost 2 minutes it crashed |

Table 6.4

## OpenStack

**OpenStack** is a set of opensource software tools for building and managing cloud computing platforms for public and private clouds.

### DevStack

**DevStack** is an opinionated script to quickly create an OpenStack development environment.

### Installation

During this testing, we deployed OpenStack on a single multicore machine using devstack. We followed the [official documentation](https://docs.openstack.org/devstack/latest/index.html) of OpenStack. During the installation, we have resolved many issues by editing multiple configuration files and inserting subjective instructions in them.

### OpenStack Services

When we use devstack following services would run in the background to create a basic environment of OpenStack.

#### Nova

Nove is using to implement services and associated libraries to provide massively scalable, on-demand, self-service access to computing resources, including bare metal, virtual machines, and containers. (OpenStack, 2018)

#### Neutron

OpenStack Neutron is an SDN networking project focused on delivering networking-as-a-service (NaaS) in virtual computing environments.

#### Swift

Swift is a highly available, distributed, eventually consistent object/blob store. Organizations can use Swift to store lots of data efficiently, safely, and cheaply. It's built for scale and optimized for durability, availability, and concurrency across the entire data set. Swift is ideal for storing unstructured data that can grow without bound.

It is by default turned off in devstack and can be enabled by adding the following line in the local.conf file available at ~/devstack.

#### Glance

Glance image services include discovering, registering, and retrieving virtual machine images.

#### Keystone

Keystone provides API client authentication, service discovery, and distributed multi-tenant authorization by implementing OpenStack’s Identity API.

#### Cinder

Cinder is a Block Storage service for OpenStack. It virtualizes the management of block storage devices and provides end users with a self-service API to request and consume those resources without requiring any knowledge of where their storage is actually deployed or on what type of device

#### Other services

OpenStack provides different services to its customer which can be integrated and used according to the need of the user.

### Testing

To test OpenStack services we created images, networks and launched multiple instances of CIRROS. We checked all these operations after loading and unloading of our loadable kernel module JPROBEFUTEXWAKEKILL.

Before testing OpenStack services I tested qemu-system-x86 instance. I made a virtual machine of cirros and then straced it and got almost 6000 futex calls when it started and then got 2000 more when logged in cirros. when a single futex call gets killed by jprobefutexwakekill module, it doesn't matter what call it is, it could be 0th,10th,1000 or 8000 the qemu-system-x86 instance just gets killed and we have been shown the status of the shutdown. It also doesn't start after getting killed.

### Nova

This component of OpenStack is used to provide on-demand access to compute resources. It consists of the following subcomponents to make this service work tirelessly.

#### Nova-Conductor

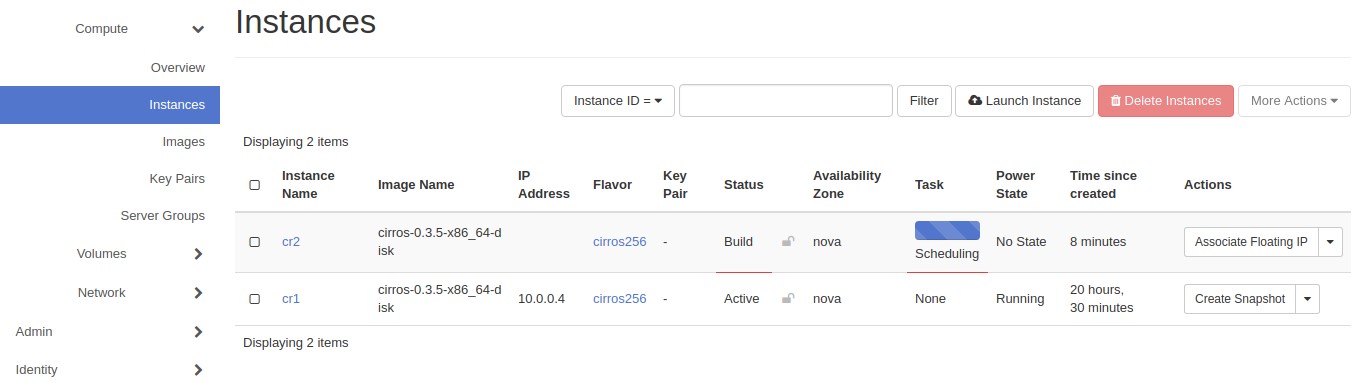
The nova-conductor service enables OpenStack to function without compute nodes accessing the database.

##### Test Summary

Nova conductor needs 12 futex calls to start and if any of it gets killed the service filed to start and then we are unable to launch a new instance.

##### Test Details

So when I tried to check the status of nova-conductor I did this sudo systemctl status devstack@n-super-cond.service command on the terminal and found that the service was perfectly running in the background. After stracing it I got no futex calls so I applied module on it and when I restarted it via *systemctl start devstack@n-super-cond.service* I found 12 futex calls which obviously got canceled by our module and then the service got broke and didn’t start. While it was not working any problem was faced while operating already running instance, So I tried to launch another instance which also got launched but status didn't change after almost 10 mints from *build* to active and the **task** was just showing schedule.



*Image 6.7*

Moreover, I was able to clone the instance by creating its snapshot.

#### Nova-Compute

Nova compute is responsible to provide compute resources. It is the main component of Nova

##### Test Summary

It makes a lot of calls when the instance starts, shutdown and when the instance is spawning. If any of the calls get killed it needs to restart to complete the incomplete work like shutting down or spawn a new instance. After certain limits of futex calls if calls get killed then accessing the console of running instance is futile.

##### Test Details

The module was loaded with the skip-count value of 500. When the instance was running the nova-compute service made no futex calls and when it was ordered to shut down it made a lot of futex calls and killing of which started as described after 500 calls but when 22 calls killed we recorded no more calls and the service was failed to run.

As the service was not running in the background the shutting down instance was hanged there and no further instance was able to launch. So, I tried running it again via following command

sudo systemctl status devstack@n-cpu.service

Suddenly my hanged instance who was shutting down for quite some time had finally got the chance to do so. The nova-compute service when started made 4760 futex calls to start itself and able to shut down my hanged instance.

After more observing it via strace the service was constantly making futex calls although no instance is running. I created an instance of CIRROS and loaded the JPROBEFUTEXWAKEKILL module and with 10000 skip counts and when it reached this limit module killed 10 calls and recorded no more. The strange thing happened when systemctl was showing its status as running but I was unable to access it via console.

##### Nova-Scheduler

Compute uses the nova-scheduler service to determine how to dispatch compute requests. For example, the nova-scheduler service determines on which host a VM should launch.

##### Test Summary

Make absolute 12 to 18 futex calls while starting and 1 call while stopping if any call gets killed this service will not start and then the new instance would not launch. Moreover, creating and deleting of instance cost no futex calls.

##### Test details

Need 12 to 18 futex calls to start and to stop it needs 1 call. If any of the calls get killed the service would not start. If the service stop running while the instance is spawning its status will be hanged at scheduling. And when the service came up we are able to create an instance successfully. This service won’t make any futex call during the creating and deleting of the instance.

When tried to create new instance got the following in the status of the process.

Lock "placement\_client" released by "nova.scheduler.client.report.\_create\_client"

Lock "(u'Xroot', u'Xroot')" acquired by "nova.scheduler.host\_manager.\_locked"

Lock "(u'Xroot', u'Xroot')" released by "nova.scheduler.host\_manager.\_locked"

And when I tried to stop

|  |  |  |  |
| --- | --- | --- | --- |
| Acquired semaphore "singleton\_lock" | {{(pid=8758) | lock | /usr/local/lib/python2.7/dist- |
| packages/oslo\_concurrency/lockutils.py:212}}  Releasing semaphore "singleton\_lock" | {{(pid=8758) | lock | /usr/local/lib/python2.7/dist- |
| packages/oslo\_concurrency/lockutils.py:228}}  Acquired semaphore "singleton\_lock" | {{(pid=8757) | lock | /usr/local/lib/python2.7/dist- |
| packages/oslo\_concurrency/lockutils.py:212}}  Releasing semaphore "singleton\_lock"  packages/oslo\_concurrency/lockutils.py:228}} | {{(pid=8757) | lock | /usr/local/lib/python2.7/dist- |

Table 6.5

#### Nova-novncproxy

nova-novncproxy is a server daemon that serves the Nova noVNC WebSocket Proxy service, which provides a WebSocket proxy that is compatible with OpenStack Nova noVNC consoles.

##### Test Summary

Takes futex calls to run and if they get killed it doesn’t start but that doesn’t affect the creation and deletion of instance.

##### Test details

It takes 21 futex calls to run and if jprobefutexwakekill module kills any call except the last two the program doesn’t start. It doesn’t make any futex calls while running so jprobefutexkill module is futile here. We are also able to create and delete instances while it’s not running.

#### Nova-Consoleauth

It is nova’s console authentication service

##### Test Summary

Requires futex calls to start and if they get killed then the instance’s console was unable to get accessed.

##### Test details

Takes 12 calls to start and if any of it gets killed by the module it won’t start. And if it is not running we are unable to access console but launching a new instance and deleting them works fine. When it runs it doesn’t make any futex calls.

#### Glance

Glance image services include discovering, registering, and retrieving virtual machine images.

##### Glance-API

Creates, lists, shows, updates, deletes, and performs other operations on images.

##### Test Summary

It works with a uWSGI program which holds other OpenStack services. uWSGI needs must futex calls otherwise it stops and effects most services of OpenStack.

##### Test details

This service’s daemon runs as uWSGI and to monitor glance-API we need to look for uWSGI

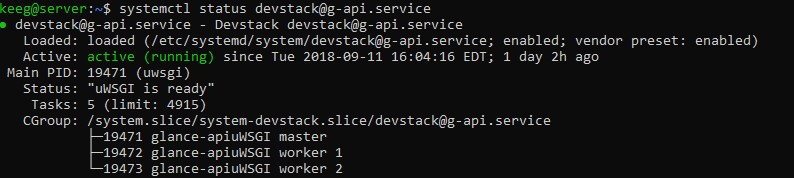


Image 6.8

which takes 65 calls to start and if anyone was killed by jprobefutexwakekill module the program does not starts. Launching new instances were showing unable to retrieve images and those images that were already running was shut down. And after some time as uWSGI was not running, we saw nova-compute stopping and horizon was also stopped working.

#### Glance-Registry

Glance-registry is a server daemon that serves image metadata through a REST-like API.

##### Test Summary

Makes essential futex calls when starts but if gets killed then it doesn’t affect OpenStack operations.

##### Test details

Takes 12 plus 20 futex calls to start killing these calls would not start the service. If it does not run, then we don’t see any effect on creating and deleting instances.

#### Cinder

Cinder is a Block Storage service for OpenStack. It's designed to present storage resources to end users that can be consumed by the OpenStack Compute Project (Nova).

##### Cinder-API

No Futex calls recorded.

3.2 Cinder-Scheduler

Doesn’t making any futex calls while I was testing OpenStack Suddenly recorded 43 futex calls so I tried to recreate all the actions which I did previously but in vain.

##### Cinder-Volume

No futex calls recorded.

#### Keystone

Keystone is the identity service used by OpenStack for authentication (authN) and high-level authorization (authZ). It currently supports token-based authN and user-service authorization.

##### Test Summary

Keystone requires three background process one master which needs one futex call and two workers which require 65 plus futex calls to start. If we load module and don’t allow more than 3 futex calls, we are not allowing worker processes to spawn and if they don’t get futex locks they constantly try for it even after 1000 kills. If the only master process is running, we cannot authenticate ourselves to login into the horizon.

##### Test details

It runs as a uWSGI process in the background which makes no futex calls while running. Keystone service takes 65 futex calls to fully start with one master process and two worker processes. The master process requires 1 necessary futex if gets killed we see the following error.

Job for devstack@keystone.service failed because a fatal signal was delivered to the control process. See "systemctl status devstack@keystone.service" and "journalctl -xe" for details.

After one necessary futex call of the master process, other futex calls are for worker process and if they get killed we see process status active but its workers are just constantly trying to spawn and making constant futex calls which are being killed by jprobefutexwakekill module. It is a never-ending loop and worker processes are constantly trying to spawn.

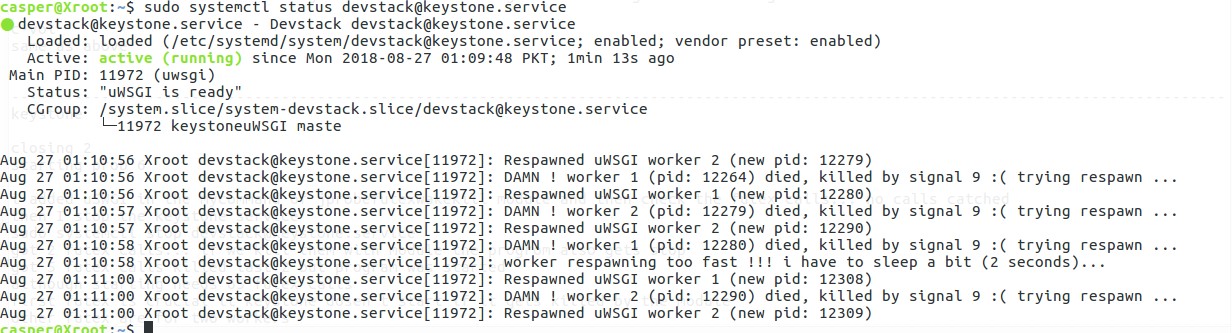


Image 6.9

With only master keystone process running we can’t start the horizon and gets the following error

Gateway Timeout

The gateway did not receive a timely response from the upstream server or application.

Apache/2.4.18 (Ubuntu) Server at localhost Port 80

And when the keystone was properly down/stop we didn't get to see the error else there was a huge error page as follow.

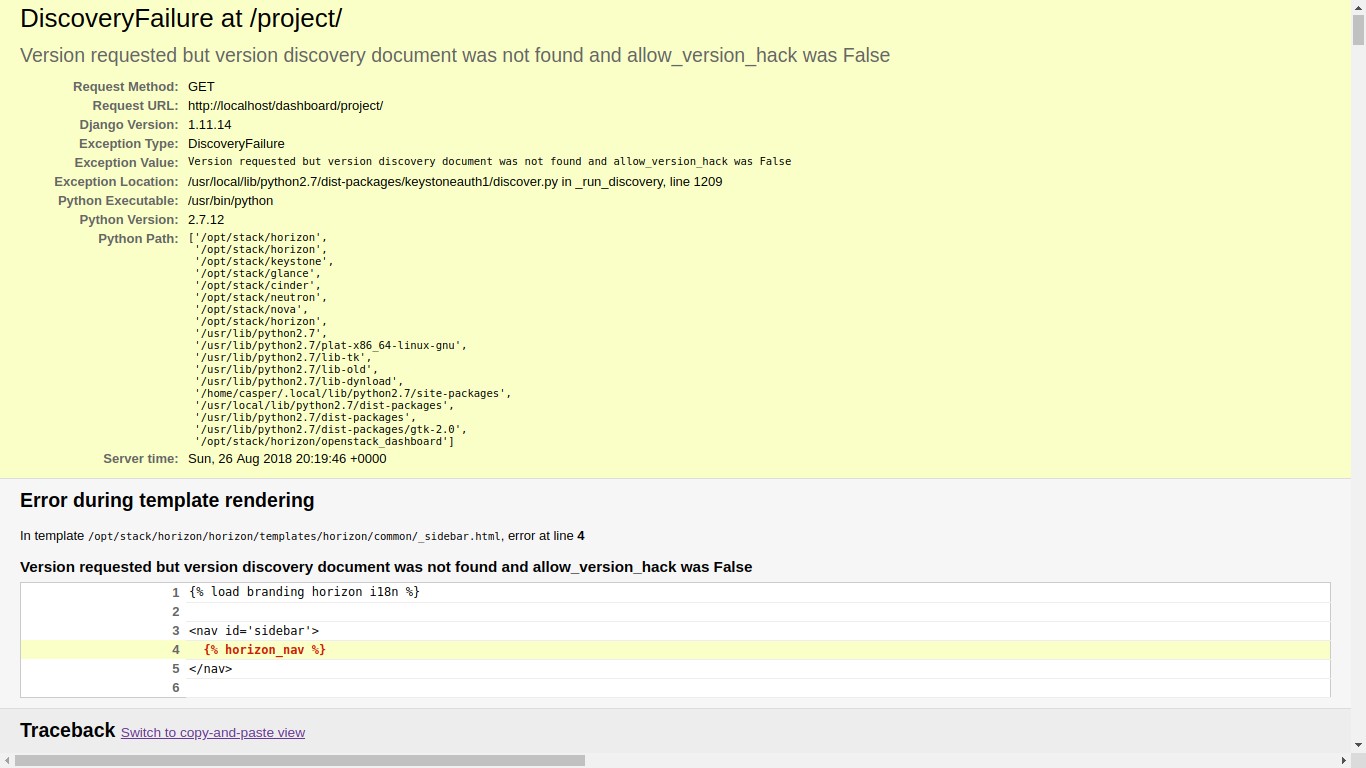


Image 6.9

I tried to run the localhost with incognito mode and I got to see the login screen of the horizon but to get in I need to authenticate which in case of an only master process running I was unable to login even with the correct password. Following error was showed when tried to log in.

An error occurred authenticating. Please try again later.

If the horizon is already running and we load jprobefutexwakekill module and restart keystone and only allow one futex call which runs the only master process, we get no response from the horizon.

Allowing all process to spawn with 65 futex calls makes everything again normal.

#### Placement-API

Nova introduced the placement API service in the 14.0.0 Newton release. This is a separate REST API stack and data model used to track resource provider inventories and usages, along with different classes of resources. For example, a resource provider can be a compute node, a shared storage pool, or an IP allocation pool. (0penstack, 2018)

##### Test Summary

Placement-API is a necessary part for creating an instance. It runs as uWSGI with three processes one master which require 3 futex calls and two workers which require 65 futex calls. If the module doesn’t allow the worker process to spawn, then we are not able to launch instances.

##### Test details

Placement-API needs 30 to 65 necessary futex calls and like Keystone, it also works as uWSGI in the background and needs one master and two workers processes. After almost 3 necessary calls which master process needs if others were killed by jprobefutexwakekill module then the worker process will be unable to run, and we see the following error when trying to launch an instance.

Unable to retrieve availability zones

unable to retrieve flavors

If launches instance when placement-API get stopped due to not allowing any futex calls, we get the following error

No valid host was found.

#### Neutron

Neutron is an OpenStack project to provide "networking as a service" between interface devices (e.g., vNICs) managed by other OpenStack services (e.g., Nova).

##### neutron-server

No Futex call recorded so jprobefutexwakekill module is ineffective here.

##### neutron-openvsw

No Futex call recorded so jprobefutexwakekill module is ineffective here

##### neutron-dhcp-ag

No Futex call recorded so jprobefutexwakekill module is ineffective here

##### neutron-l3-agen

No Futex call recorded so jprobefutexwakekill module is ineffective here

##### neutron-rootwra

No Futex call recorded so jprobefutexwakekill module is ineffective here

##### neutron-metadat

No Futex call recorded so jprobefutexwakekill module is ineffective here

## Discussion

The software tested behaved as expected when ran in a traced environment. All manual testing and tool testing gave the same results. In the case of our tool, the application’s crash analysis can be generated and the routine that leads to that crash could be explored because of the stack trace functionality. This test results could be proven very useful for the developers. As they could add exception handling into their software and remove malfunctioning behavior. As we can see in the above results the developer could easily pinpoint the issue without the need of exploring and recompiling.

C**hapter Seven**

# Epilogue

This chapter is being written to summaries the above chapters and their effects and results. Moreover, it’ll include goals achieve by this project, future work that can be formed by this dissertation. This part of the document also includes the lesson learned during the development journey of this project.

## Conclusion

This is a complete functional crash analysis tool for multi-tasking application. Its practical aspects can be observed in the previous chapters. Till date, the application solves this kind of problem by hiring a diagnostician whose job is to find and debug the issue in a minimum amount of time which takes about months. This tool could significantly affect the production system as potential locking problems could easily be observed before official shipping. The goals defined in the beginning has been fully functionally implemented and working just perfectly. Major experimenting has been done on well-known software in order to check its results. This project has taught me how to identify problems and fabricate their solution with little or no help as the internet is not very interested in this domain.

# Bibliography

Best, S. (2005) ‘Linux: Debugging and Performance Tuning - Tips and Techniques’, *Instrumentation*, p. 149247.

Downey, A. B. (2016) ‘The Little Book of Semaphores (2-nd edition v2.2.1)’.

Göetz, B. and Professional, A. W. (2006) ‘Java Concurrency In Practice’, *Building*, 39(11), p. 384. doi: 10.1093/geront/gns022.

Kaufman, J. (1981) ‘Blocking in a Shared Resource Environment’, *IEEE Transactions on Communications*, 29(10), pp. 1474–1481. doi: 10.1109/TCOM.1981.1094894.

Li, T. *et al.* (2005) ‘Pulse : A Dynamic Deadlock Detection Mechanism Using Speculative Execution’, *Atc*, pp. 31–44. doi: 10.1016/j.ejvs.2010.09.017.

Myers, G. J. (2004) *The art of software testing, Second Edition*, *… 1991., Proceedings of the IEEE 1991 National*. doi: 10.1002/stvr.322.

Padala, P. (2002) ‘Playing with ptrace, Part II’, *Linux Journal*, 104, pp. 86–91. Available at: http://www.linuxjournal.com/article/6210.

Strace.io (2019) *strace*, *strace.io*. Available at: https://strace.io/ (Accessed: 14 January 2019).

Sun Microsystems (2019) *Synchronization*, *Sun Microsystems*. Available at: https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html (Accessed: 14 January 2019).