Mid Semester Evaluation Report

# Crash Analysis Tool

## PROFILING APPLICATIONS IN MULTITASKING ENVIRONMENT

Faizan Ullah
NAMAL CLLEGE MIANWALI | 15026424

Supervisor
DR. ANJUM NAVEED | DR. ADNAN IQBAL

# Abstract

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 point out the unhandled stray locks that might cause the exceptions stated above.

# Table of Contents

# Chapter One

# 1 Introduction

This chapter defines the utility, rationale for choosing it and objectives to attain from it; along with the structure of this dissertation.

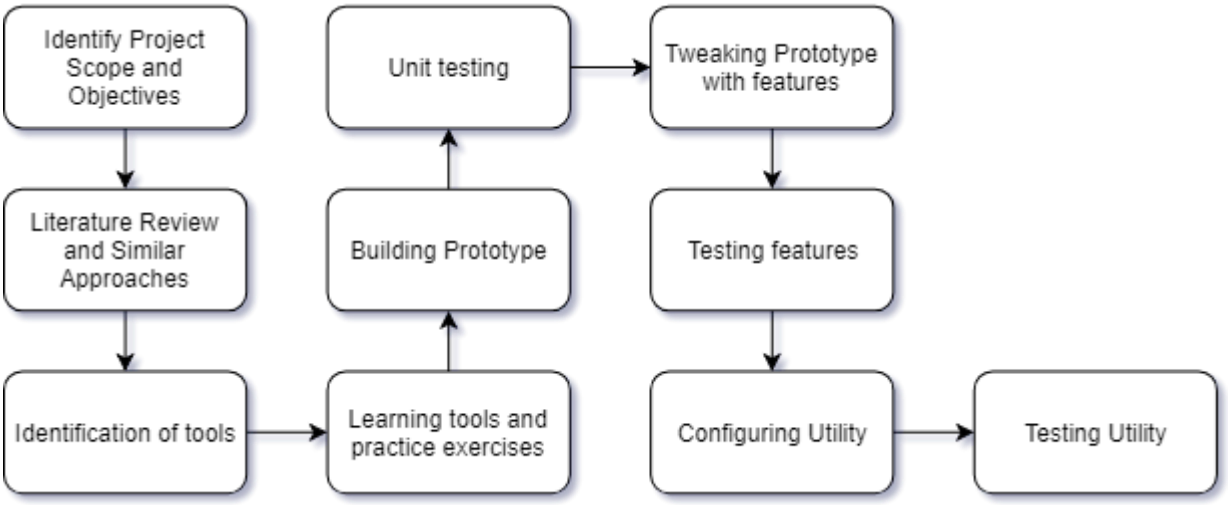
## 1.1 Motivation

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 as in many cases the utility might not even start or needs a complete reboot or sometimes the lock is permanent which needs extensive manual debugging before the user is able to use it again. MYSQL Database is one example of it that use file lock to acquire a socket which has beside of some plus points more adverse effects as in case of system failure the DBMS won't get the chance to leave the resource and in future the file is already there, and system consider that socket already being locked and not allowed to acquire.

This is one example and there are countless to make oneself familiar with the complication an incautious behavior of a programmer arises. Keeping in mind the problems and the efforts end user might be needing to put in, I took this challenge to devise a tool that'll make rigorous disruption in the locking mechanism of a software and show the developer exact spot where the problem is arising. This is a challenging task and very few people try to work at this level. I wanted to go deep into the kernel, deep enough to cherish the things many developers take for granted.

## 1.2 Methodology

Following Methodological step will be followed during this development journey of this utility




# 1.3 Project Plan

## 1.3.1 Background Understanding

Before getting to these steps we are required to get ourself acquainted with some related software like strace, ltrace, lsof, and some system calls like ptrace, execvp, fork, wait and techniques of Inter process communication (IPC) like signals and system locks etc. These prequels will help us get our main part done with ease.

## 1.3.2 Building Prototype Skeleton

Preparing a Prototype skeleton in which we'll be tweaking and modifying the following services. Our Prototype will be tracing children, grandchildren and detecting their system calls.

## 1.3.3 Adding Main Features

### *1.3.3.1 Lock Detection*

This step requires us to build individual detection system for different types of locks. Every type of lock mutex, semaphore, spinlock, seqlock etc. requires a sperate identification method.

### *1.3.3.2 Printing Backtrace*

Printing the Back Trace of a specific call. This may include steps/functions that lead to this call. The stack can be a great help in this case.

### *1.3.3.3 Detection of locked resources?*

At this step we need to add this main functionality of generate a log having all the description of the resource that has been locked.

#### *1.3.3.4 Detection of waiting process?*

To generate a log having all the description of the process that has been locked.

#### *1.3.3.5 What type of locks?*

To generate a log having all the description of the types of locks on the process and resource.

### 1.3.4 Testing some well-known software

In the end for validation purposes, some well-known application and some pre-written fake application must be tested in order to get the best results from the utility.

## 1.4 Document Structure

This following chapters will contain the discussion of the literature related to this project in journals, research papers etc. which will be discussed in the chapter named as literature review. Moreover, a fine requirement analysis will be put down to make implementation steps clear and final. After requirement analysis comes implementation, a chapter related to prototype description will define the functionalities of implemented protype. All of this will be concluded in the end and in addition to, it we'll also define the futuristic steps to be added in the software.

# Chapter Two

# 2 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.

## 2.1 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 here in Harvard referencing style.

## 2.2 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 use 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 insure the executing of 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 processes etc. We are specifically focusing on multitasking applications. Multitasking; as name suggest, execution of multiple task 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 locking 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 se how this problem occurs and cause us to analyze that crash.

## 2.3 Problem 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. 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 turn 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 inspite of the resource is available it will see that the resource is not usable and is being holding by some other system utility.(Kaufman, 1981)

### 2.3.1 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.

## 2.4 Useful Tools

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 there owner that are locking it. Both of these tools can help us identifying the problems related to 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 sees all the system calls

to the kernel, all the inter process communication, and can be use to temper with them. It is mostly usable when the source code, of the application being tested, isn't available. You can have a through 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.

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.

## 2.5 API's

To cure the problem beforehand, we are going to see how these tool work in the 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 where as 'p' in its name specifies 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)

## 2.6 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)

Chapter four

# 3 Requirement Analysis

Shall we begin to understand the need of hour. It's what is being required as we have taken the responsibility to cure the problem beforehand. In this chapter we'll discuss the requirement that our utility must be capable of.

## 3.1 Utility Description

A command line tool that will take a path of the software, that is going to be tested, as an argument and use the api above described as the tool to auto-debug it. It will check whether it is prone to acquire unnecessary resources and starve it-self or other process. What is the resource and what lock was not making it free for others? If the application gets crashed, then where was the point it got crashed? What was the Backtrace? It will be capable of generating a log of all the features describe above.

## 3.2 Functionality

Features needed for the software to work for the users.

### 3.2.1 Stray Lock Check

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 unmanage and declared it as stray.

### 3.2.2 Starving Process

What are the starving processes that are in the need of that resources?

### 3.2.3 Lock Information

After getting the knowledge of the stray lock it'll be capable of defining the type of lock. Whether it was mutex, spin, sequential, semaphore etc.

### 3.2.4 Crash Path

If the application is crashing, as because of the unavailability of the resource then we'll provide the functionality to provide the Backtrace that lead to the crash.

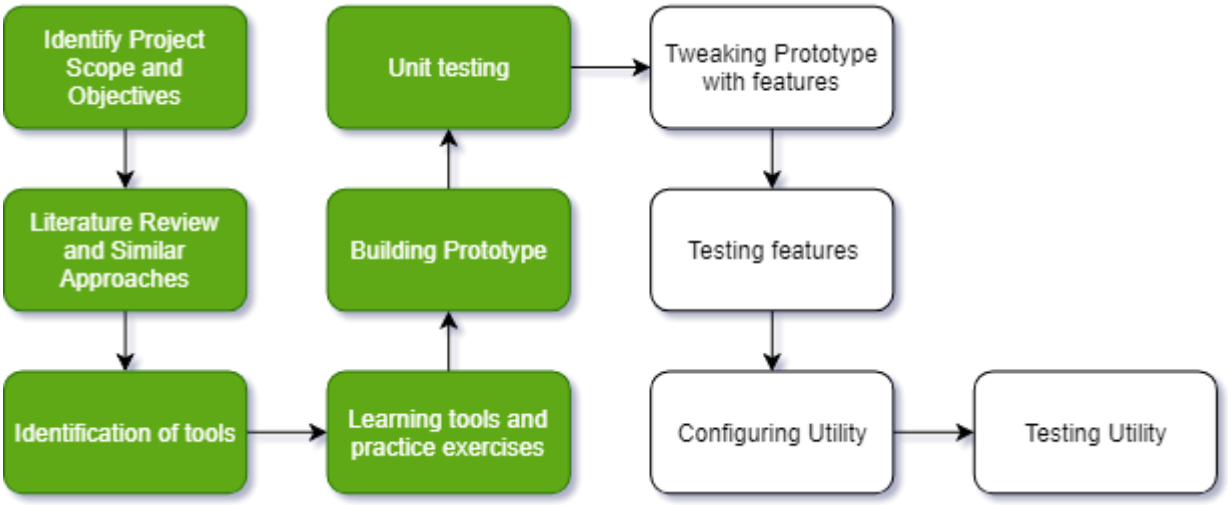
### 3.2.5 Resource Information

If you know the lock then you should know the resource that was being locked in order to tweak in the code to never make that happen again.

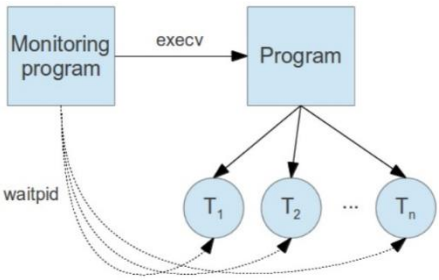
# Chapter Four

# 4 Prototype Description

This is where we'll discuss the point where our software has reached and what capabilities it has uptill now. Let's start by revisiting the methodological chart we are inclined to follow.




Our utility has developed all the base structure that it was required before adding main features. It is tracing another process with the capability of tracing its forked/cloned children.




It starts a process by taking its name as an argument or can attach with already running process also from the command line just only by taking its id followed by '-p' as an identifier. We are capable of tracing the system calls from the processed and threads we are tracing.

# Chapter Five

# 5 Conclusion

An application which can be very useful for the programmers and developers in finding their undebuggable bugs. We have developed the skeleton of our Frankenstein with which we will be tweaking in future in order to turn it into a giant. Our motivation is a natural problem, and for it solution we have gained considerable knowledge to devise a solution capable enough to solve the problem. Our utility needs its main features to be fully functional and in the next iteration our software will be in a working phase.

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).