Ktrace: A Framework for Tracing the Linux Kernel

A Thesis Submitted
in the Partial Fulfillment of the
Requirement for the Degree
Bachelor-Master of Technology
(Dual Degree)

by

Navdeep Bhulli
Y2157230

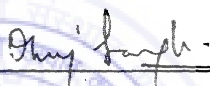
under the guidance of

Prof. Dheeraj Sanghi
and
Prof. Rajat Moona

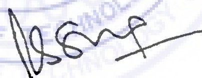
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur
August 2007

CERTIFICATE

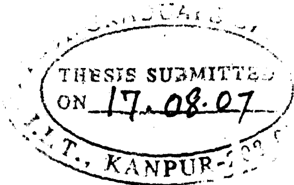
It is certified that the work contained in the thesis titled *"Ktrace : A Framework for Tracing the Linux Kernel"*, by ***Mr. Navdeep Bhulli*** has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.



Prof. Dheeraj Sanghi
Dept. of Computer Science and Engineering
Indian Institute of Technology,
Kanpur.



Prof. Rajat Moona
Dept. of Computer Science and Engineering
Indian Institute of Technology,
Kanpur.



August 2007

Abstract

Many tools exist to analyze the Linux Operating System but very few of them can be used to analyze the Linux kernel. The existing tools are not fully capable of providing a picture of the kernel level activity of the system to a user - some tools are inclined towards a particular aspect of the kernel while others provide a more generic information without going into details.

We propose and implement a centralized tracing framework for the Linux kernel. The framework is based on the simple idea of giving a snapshot of the Linux kernel to the user at the occurrence of certain pre-defined kernel events. The snapshot is provided in terms of a set of kernel level variables. The key feature of the framework is that it is easily extensible - the set of pre-defined events as well as the kernel level variables can be enhanced without redesigning or modifying the existing code.

The tracing framework provides the functionality to trace the entire system or a particular process. The framework also takes care of the security issue - a user can only trace its own processes. Only the *superuser* has the privilege to trace any process.

Contents

1 Introduction **1**

1.1 Related Work . . . 2

1.2 Organization of the Thesis . . . 2

2 Previous Work **3**

2.1 The */proc* Filesystem . . . 3

2.2 Strace . . . 4

2.2.1 How does Strace work . . . 4

2.3 DTrace . . . 5

3 Design of Ktrace **7**

3.1 Overview . . . 7

3.2 User Module . . . 7

3.2.1 List of Events . . . 8

3.2.2 List of Variables . . . 9

3.3 Kernel Module . . . 11

3.4 The Trace Buffer . . . 11

3.4.1 Synchronization Issues . . . 11

3.4.2 Trace Buffer Policy . . . 12

3.4.3 Data Organization . . . 13

3.5 Global Event and Variable Masks . . . 13

3.5.1 Data Logging . . . 13

3.5.2 Data Reading . . . 14

3.6 Tracing a Process . . . 14

3.7 User Access to Tracing . . . 14

4 Implementation of Ktrace **15**

4.1 Kernel Module . . . 15

4.1.1 The Trace Device Driver . . . 16

4.2 User Module . . . 17

4.3 The Trace Buffer . . . 17

4.4 Data Logging . . . 18

4.4.1 Extensibility of Ktrace 18
4.5 Data Reading 19
4.5.1 Extensibility of Ktrace 19
4.6 Data Filtering 20
4.6.1 Data Consistency 20
4.6.2 State of Variables 21
4.6.3 Filtering Mechanism 22
4.6.3.1 Operations on the Sorted List 23
4.7 Global Event and Variable Masks 23
4.7.1 Reader Data Structure 24
4.8 Integrating trace_dump with the Kernel 24

5 Results and Discussion **26**
5.1 Introduction 26
5.2 Comparison with */proc* 26
5.3 Comparison with Strace 27
5.4 Tracing Applications 29
5.4.1 Top Utility 29
5.4.1.1 Analysis of Trace Data 29
5.4.1.2 Improved Implementation 32
5.4.2 Xmag Utility 33
5.4.2.1 Trace Data Analysis 34
5.4.2.2 Alternative Approach 36
5.4.2.3 Other System Calls 36
5.5 Memory Related Examples 37

6 Conclusion and Future Work **39**
6.1 Conclusion 39
6.2 Future Work 39

A Encoding Rules for Buffer Data **44**
A.1 Introduction 44
A.1.1 Encoding of the Tag Field 44
A.1.2 Encoding of the Length Field 45

B Inserting the Trace Module **46**

List of Figures

3.1 Overview of Ktrace 7
3.2 Trace Buffer Policy View Point 12

4.1 Interaction between User level and Kernel level modules 17
4.2 Trace Buffer snapshot at some time 20
4.3 A Snapshot of Trace Buffer at some point of Time 22

5.1 Memory Usage of the System with Time 27
5.2 System Call Tracing for all Applications by Ktrace 28
5.3 Initial System calls made by Top 30
5.4 Repeated System calls made by Top 31
5.5 CPU Usage Comparison between two Implementations 34
5.6 Repeated System calls made by Xmag 35

A.1 Tag Encoding - Case 1 44
A.2 Tag Encoding - Case 2 44
A.3 Length Encoding - Case 1 45
A.4 Length Encoding - Case 2 45

List of Tables

3.1 Table of Events 8
3.2 Table of Variables 10

5.1 Summary of System Calls made by Top 32
5.2 Summary of System Calls made by Xmag 36

Chapter 1

Introduction

The ability to observe the system and to understand what is going on inside the system is very useful for all the users. It is possible that different users may be interested in different aspects of the system. For example, a system administrator may be interested in finding out which system resource is acting as the bottleneck while a programmer may want to find out where the application is spending most of its time. On the other hand, a learner would like to know what is going on in the system just to understand it.

The kernel is assumed to be a black box of the operating system which drives the whole system. But what actually happens in the kernel remains a mystery to most people. This is not desirable since the kernel performs operations such as scheduling, disk I/O, memory management, system call invocation etc. which handle the system resources. It would be better if somehow the factors that rule the performance of the kernel are exposed to the users of the system. If a mechanism is available that can give a complete picture of the kernel activity, it can be used to identify the resource usage in the system and handle the performance bottlenecks.

The usually available statistics like CPU usage, memory usage etc. can identify which resource is not being used judiciously. But just an overall system picture is not very useful. There is no way the entire system activity can be reconstructed by using these tools.

We attempt to build a framework which provides a detailed view of the kernel activities of the system. The framework is known as *Ktrace* and is based on the notion of events and variables. "Events" are certain occurrences in the operating system which the user might be interested in. Whenever any event occurs, the current "system state" is recorded along with the current time stamp. The system state is defined in terms of a set of "variables". The values of these variables provide an insight into the kernel. For example, the "number of free pages" can be recorded at each memory fault event. The variables in our work may correspond to variables in the Linux kernel code or may be only virtual variables, the values of which are computed each time they are to be recorded. The "event" always corresponds to some point of execution inside the kernel.

Some salient features of Ktrace are the following.

- A centralized extensible framework : Ktrace acts as a central place for tracing and is easily extensible. The basic set up has been developed with a certain set of events and variables to provide the user with the tracing data. If at some point of time, a need arises for adding a new event or a variable, it can be easily accomplished without redesigning and without major modifications in the existing code.

- Security : A user can only trace his/her own processes. There is no way a user can access the data of the processes owned by others. The *superuser* can trace all processes.

- System or Process : The entire system can be put under tracing or a particular process can be traced at a given time.

1.1 Related Work

Strace [1] is a tool that provides a complete picture of the system calls made and signals received by a process. This provides a view of the interaction between user mode and kernel mode. However this approach is only limited to system calls and signal behavior of the system. Information about other kernel activities would also be very useful.

The Linux kernel debugger [2] provides a direct way to analyze the Linux kernel. But the problem is that one needs to add explicit breakpoints to observe the kernel memory and data structures at any point of execution.

DTrace [3] is a comprehensive tracing framework for the Solaris Operating System and it provides very good insight into the workings of Solaris kernel. But it can not be directly ported to Linux because of the differences between Linux and Solaris kernels.

All of the above frameworks do not have the ability to reconstruct the kernel level activity and provide the required information to the user. Yet, DTrace provided us with many ideas for designing and implementing Ktrace.

1.2 Organization of the Thesis

The rest of the thesis is organized as follows.

In chapter 2, we discuss various existing works that provide the functionality to analyze the Linux kernel. In chapter 3, we describe various design related issues encountered while developing Ktrace. In chapter 4, we talk about the implementation details of Ktrace. In chapter 5, we show how Ktrace can be applied to analyze the Linux kernel. In chapter 6, we summarize the work done and also list the possible ways in which this work can be extended.

Chapter 2

Previous Work

2.1 The */proc* Filesystem

The */proc* filesystem [4] is a virtual filesystem which acts as an interface to the kernel data structures. It is one of the most basic ways of making kernel data visible to the users. Most of the user space based profilers like *Top* [5], *Vmstat* [6] are based on the data provided by */proc*.

/proc is not a true filesystem in the sense that it does not exist on the disk. In reality, it is just a set of kernel memory data structures which provide important information about the state of the system as a whole as well as about the individual processes. The layout of the filesystem is built around two categories.

- Process Specific Info : */proc* contains a directory for each running process on the system. Some of the files in each process directory are - *cmdline*, *fd*, *mem*, *stat*, *status* etc.

- System Specific Info : */proc* contains a set of files which specify the overall system state at this point of time. Some of the important files in this category are the following.

  - Devices : It contains a listing of all the devices of the system along with their major numbers.
  - Diskstats : It contains statistics related to I/O for each disk device.
  - Meminfo : It contains the total and free memory of the system.
  - Mounts : It contains the list of currently mounted file systems.
  - Net : This is a directory and it contains a set of files specifying network statistics.

While */proc* provides an extensive list of data related to each aspect of the system, it only provides the information related to the final state of the system at any given time.

It does not have the capability to provide the information on the events that happened en route to the system's present state.

2.2 Strace

Strace [1] is a tool that provides a mechanism to trace system calls and signals. It executes the specified command (taken as an option at run time) and prints the system calls made by the corresponding process and the signals received by the process. For system calls, the output consists of the name of each system call, its arguments and the return value. In case of signals, the signal name is printed.

A subset of system calls or signals can be specified if one is not interested in tracing all the system calls or signals. An already running process can also be traced by specifying its *pid* as the command line option. A detailed description of Strace can be found in the online manual page [1].

2.2.1 How does Strace work

Strace basically makes use of the *ptrace* system call. The ptrace [7] system call provides a mechanism by which a process (tracing process) may observe and control the execution of another process (traced process). It is primarily used to implement breakpoint debugging and system call tracing.

In the simplest case, the traced process can be a child process of the tracing process. The traced process is stopped whenever a signal is delivered to it and the tracing process is notified.

The *ptrace* system call takes as an argument a "request" field which determines the action to be performed. The following are the possible values for this "request" field.

- PTRACE_TRACEME : This is used by the traced process to enable tracing. The common use of PTRACE_TRACEME option is when in a program, *fork* is called. The child process (created by *fork*) enables the tracing prior to executing *execve* system call, thereby enabling the tracing process to trace the execution of the concerned program.

- PTRACE_ATTACH : If the traced process is already running, then another process owned by the same user can issue the *ptrace* system call with the PTRACE_ATTACH option. In this case, the process with specified "pid" (which is the second argument of *ptrace*) is "traced" by the calling process.

- PTRACE_SYSCALL : If the tracing process uses the PTRACE_SYSCALL option, the traced process is also stopped at the system call entry or exit and the tracing process is notified. The tracing process can examine the traced process' registers and obtain the system call number, its arguments and the return value.

- PTRACE_CONT : When called by the tracing process, this option continues the execution of the traced process that had been stopped because of trace event.

PTRACE_ATTACH, PTRACE_SYSCALL and PTRACE_CONT are used by the tracing process. In essence, tracing a process requires the traced process to make a call to *ptrace* with PTRACE_TRACEME or the tracing process to make a call to *ptrace* with PTRACE_ATTACH. Subsequently, the tracing process can make calls to *ptrace* with PTRACE_SYSCALL and PTRACE_CONT.

This mechanism is not easily extensible. The *ptrace* system call does not have any option to intercept any other events and record them. Also, this approach tinkers with the natural scheduling of the processes - the traced process is suspended till the tracing process issues a PTRACE_CONT. So it might not be very useful if one wants to study the behavior of the scheduler.

2.3 DTrace

DTrace [3] is a comprehensive dynamic tracing framework for the Solaris Operating System. It provides the capability to execute user specified actions at the occurrence of certain kernel level events. *DTrace* has two major components.

- Kernel Instrumentation : This part is responsible for tracing the kernel code. The Solaris kernel code is modified to execute certain specified actions on occurrence of certain events.
- D language : DTrace provides a scripting language which provides a way for a user to express what he wants to be traced.

A comprehensive overview of the architecture of *DTrace* is provided by Cantrill and others [8]. A sample D script looks something like the following.

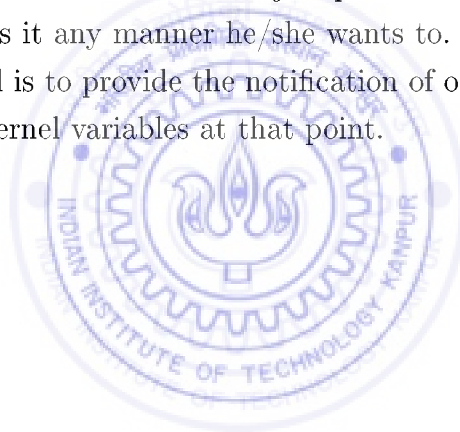
```
dtrace:::BEGIN
{
    scall = 0; sread = 0; swrit = 0;
}

syscall:::entry    { scall++; }
sysinfo:::sysread  { sread++; }
sysinfo:::syswrite { swrit++; }

dtrace:::END
{
    printf("%d %d %d\n", scall, sread, swrit);
}
```

A script consists of certain number of clauses - each clause made up of a relevant event and the actions to be taken upon the occurrence of that particular event. Except for two special events - BEGIN (start of trace) and END (end of trace), all other events correspond to some point of execution within the kernel. Solaris Dynamic Tracing Guide [9] provides details about writing DTrace scripts.

The kernel has been instrumented at each place corresponding to all the events supported by *DTrace*. Whenever a relevant event occurs, the control is transferred to the *DTrace* module to execute the actions specified by the user. The actions included in the script are converted into some intermediate form.

A compiler is needed to convert a script into an intermediate form and check for the errors in the script. Also, since any user can write the tracing script and the actions are to be executed from within the kernel, the script needs to be validated. It needs to be checked for conditions like illegal memory access etc. requiring significant amount of processing.

Most of the actions allowed to be written in the D language are related to actual formatting of the data. From the performance viewpoint, there is no need for such actions to be executed in the kernel. The kernel can just provide the user with "raw" tracing data and the user can process it any manner he/she wants to. The basic functionality that is required from the kernel is to provide the notification of occurrence of certain events and the values of different kernel variables at that point.



Chapter 3

Design of Ktrace

3.1 Overview

The aim of Ktrace is to provide the user with the "system state" (values of variables) on the occurrence of specific events. There are two major components of Ktrace - a user level module and a kernel level module. The kernel level module is the actual tracing component. The user level module is more like a front-end of Ktrace. It is responsible for passing various options to the kernel module and to process the trace data provided by the kernel module in a human-readable format.

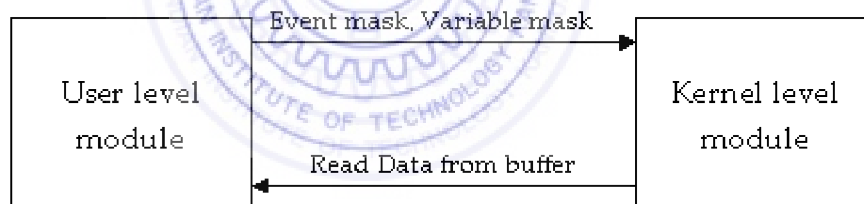


Figure 3.1: Overview of Ktrace

3.2 User Module

The user module acts as a bridge between the user and the kernel module. The user module provides the user interface. It interacts with the kernel module using a device driver. It must pass the following information to the kernel module (through *ioctl* system call).

- **Event Mask** : It is a mask for the list of events that the user wants to be traced. This is a 32 bit integer that supports one bit for each event. Up to 32 events can be specified to the kernel.
- **Variable Mask** : It is a mask for the list of variables that the user wants to be traced. This is also a 32 bit integer, supporting up to 32 variables, one bit per variable.
- **Pid** : It is the pid of an already running process which the user wants to be traced.

The event and variable masks can be computed by ORing the masks for individual events or variables as listed in Tables 3.1 and 3.2. The events and variables listed in the table below have been prepared after examining the execution flow of the Linux operating system.

3.2.1 List of Events

The complete list of events (along with masks for each one of them) supported by Ktrace is shown in Table 3.1.

Table 3.1: Table of Events

Event Name	**Mask**
Syscall Entry	0x1
Syscall Exit	0x2
Page Alloc	0x4
Page Free	0x8
Schedule In	0x10
Schedule Out	0x20
Lock Acquire	0x40
Lock Release	0x80
Signal Dispatch	0x100
Signal Handle	0x200
Socket Open	0x400
Send Packet	0x800
Receive Packet	0x1000
Mount Filesystem	0x2000
Unmount Filesystem	0x4000

A brief description of the events is as follows.

- Syscall Entry : A system call is about to start execution in the kernel mode. The Linux kernel contains a centralized mechanism to invoke system calls (using interrupts through *int $0x80* instruction) and this central point corresponds to the system call entry event.

- Syscall Exit : A system call is about to return to the user mode. The return from a system call also takes place through a central mechanism and it corresponds to the system call exit event.
- Page Alloc : A page of memory is assigned by the Linux kernel. This event corresponds to the beginning of the *__alloc_pages* function [10] of the kernel.
- Page Free : A page of memory is freed by the Linux kernel. This event corresponds to the beginning of the *free_pages* function [11] of the kernel.
- Schedule In : The scheduler is invoked by the Linux kernel. The actual point is the beginning of the *context_switch* function [12] in the Linux kernel.
- Schedule Out : The scheduler's job is finished. The actual point is the end of the *context_switch* function [13] in the Linux kernel.
- Lock Acquire : A lock is acquired by the current process.
- Lock Release : A lock is released by the current process.
- Signal Dispatch : A signal is delivered to the current process.
- Signal Handle : A pending signal of the current process is handled.
- Socket Open : A socket is created using the *socket* system call.
- Send Packet : A TCP Packet is sent by the system.
- Receive Packet : A TCP Packet is received by the system.
- Mount Filesystem : A filesystem is mounted using the *mount* system call.
- Unmount Filesystem : A filesystem is dismounted using the *umount* system call.

3.2.2 List of Variables

The complete list of variables (along with masks) supported by Ktrace is shown in Table 3.2.

A brief description of the variables is as follows.

- CPU Registers : It corresponds to the process specific set of CPU registers. We update them only at the time of system call entry and system call exit. So they always store the last issued system call on the system along with its arguments and the return value.
- PID : It is the combination of the *pid* and *uid* of the current process.
- PPID : It refers to the *pid* of the parent of the current process.

Table 3.2: Table of Variables

Variable Name	Mask
CPU Registers	0x1
PID	0x2
PPID	0x4
PGID	0x8
Free Memory	0x10
Total Memory	0x20
Page Faults	0x40
Page Ins	0x80
Page Outs	0x100
Swap Ins	0x200
Swap Outs	0x400
Outgoing Process	0x800
Incoming Process	0x1000
Signal	0x2000

- PGID : It stands for Process Group Id. This represents the process group to which the current process belongs.
- Free Memory : It denotes the amount of free memory of the entire system.
- Total Memory : It denotes the total memory of the system.
- Page Faults : This variable corresponds to the total number of page faults that have happened so far in the system.
- Page Ins : This variable stores the number of disk reads that have happened so far in units of pages.
- Page Outs : This variable stores the number of disk writes that have happened so far in units of pages.
- Swap Ins : This variable stores the number of swap ins that have happened so far in units of pages.
- Swap Outs : This variable stores the number of swap outs that have happened so far in units of pages.
- Outgoing Process : It corresponds to the *pid* of the last running process on the system. This variable only makes sense at the time of "Schedule In" event.
- Incoming Process : It corresponds to the *pid* of the process that is next going to run on the system. This variable only makes sense at the time of "Schedule Out" event.

- Signal : This variable denotes the last signal that has been delivered to the current process.

3.3 Kernel Module

The kernel module is responsible for implementing tracing. It obtains the tracing options from the user module through a device driver interface and can set up Ktrace accordingly. At the occurrence of an event to be traced, the kernel module has to store the values of all the required variables (as defined by the variable mask).

For the purpose of storing the values, the kernel module needs access to a memory buffer. This memory buffer keeps the "system state" on the occurrence of events. But this tracing data has to be transferred to the user module because it is the user module which processes this data for the user.

The tracing is therefore implemented with a two way interaction between the user module and the kernel module.

- The user module passes on the tracing options to the kernel module. This happens only once for each user module.
- The data logged by the kernel module is continuously transferred to each user module. This happens repeatedly during the time tracing is going on.

3.4 The Trace Buffer

The kernel module has access to a buffer to log the tracing data. This buffer is modified by two operations.

- **Data Logging** : The data is written to the trace buffer upon occurrence of an event specified by the event mask. There is a data logging function which is called upon occurrence of the trace events.
- **Data Reading** : The data is transferred from the trace buffer to the user module. There is a data reading function which is called when the user module issues the *read* system call for the *trace* device.

These operations are implemented in the kernel module.

3.4.1 Synchronization Issues

The kernel module is the sole writer to the trace buffer but there can be multiple readers, each reader corresponding to one user module. This appears to be a multiple reader-writer problem which requires proper synchronization using locks etc. We consider only

single processor systems as synchronization issues become more complicated in case of multi-processor systems.

There are two functions at work here - a reading function which acts on behalf of the readers and a logging function which does the writer's job. These functions are implemented in the kernel trace module and hence both are kernel functions. As we know, the Linux kernel is a non-preemptible kernel i.e. a user process executing inside the kernel can not be forced to give up the CPU until its work is done. This means that a process is not going to be scheduled out while either the reading or writing to the trace buffer is taking place. By this simple design, synchronization has been ensured.

3.4.2 Trace Buffer Policy

The trace buffer is maintained as a circular buffer with a "write pointer" indicating the next place to write the data. There are several "read pointers", one for each reader. The kernel module keeps on writing into the buffer in a circular manner. Such mechanism may overwrite the previous stored data (which becomes obsolete). The reason for this policy is that different readers will move at different rates and not writing when the buffer is full means we are limiting all the readers to the speed of the slowest reader which is not fair to all other readers.

An alternative view of this scenario is shown in Figure 3.2. The trace buffer can be considered to be of infinite size but there is a window of fixed size which corresponds to the current active region. This window will contain the data to be read by the user modules. All the data logged before this window is obsolete and can be thought of as ancient forgotten history (as it is no longer available in the trace buffer) and space ahead can be thought of as belonging to the future.

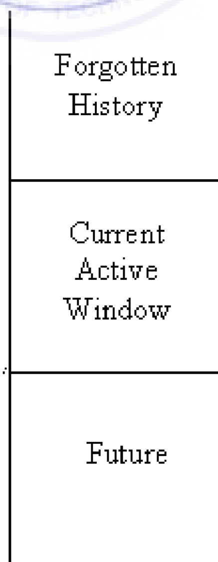


Figure 3.2: Trace Buffer Policy View Point

The kernel module writes at the end of the current active window leading into the future space. The reader can only read data from the current active window (as only that data is relevant). If some reader lags behind, its "read" pointer is brought into the current active window at the time of its next read.

3.4.3 Data Organization

Since the trace buffer data has to be transferred between two entities (between the user and kernel modules), a protocol needs to be established which will determine the encoding and decoding of the data. The data inside the trace buffer is organized in the form of TLV triplets i.e. **T**ag, **L**ength and **V**alue. Each event and variable is assigned a unique numerical *tag*. The *length* field stores the space taken by the value of the corresponding tag. The *value* field stores the data - the actual value in case of variables and the time stamp in case of events.

The mapping from data to a TLV triplet is determined by the **B**asic **E**ncoding **R**ules used in ASN.1 [14]. ASN.1 stands for **A**bstract **S**yntax **N**otation One - a standard for representing, encoding, transmitting and decoding data. ASN.1 provides few encoding rules and we use the Basic Encoding Rule Set to encode and decode our data. The detailed mapping according to the rules is described in Appendix A.

3.5 Global Event and Variable Masks

There can be multiple user level trace modules running at the same time - each user module corresponding to a separate trace invocation. Each user level module might be possibly interested in different sets of events and variables, which it can convey to the kernel module.

Naturally the question now arises - what events and variables the kernel module is going to log into the trace buffer at the occurrence of an event. So the concept of a "global event mask" and a "global variable mask" comes into the picture. The "global event mask" is the union of all the event masks specified by all the currently active user level modules (same holds for the global variable mask).

3.5.1 Data Logging

The kernel module is only concerned with the global event and variable masks at the time of logging the data. Only those variables are logged which are present in the global variable mask and logging is done only if the event is part of the global event mask.

3.5.2 Data Reading

The concept of global event and variable masks also affects the data reading procedure (which transfers the tracing data from the trace buffer to each individual user module). The trace buffer contains data corresponding to the global masks. However a user module expects data according to its own specifications of event and variable masks. The reading function therefore is designed to transfer only the relevant data to each user module.

3.6 Tracing a Process

The user might be interested in tracing a particular process only rather than the entire system. There are two possibilities.

One way is that the user may want to trace a process which is already running on the system. In this case, the user passes the *pid* of the concerned process to the user module which passes it onto the kernel module. The issue with this approach is that since the process is already running, the events leading up to the starting of the trace can not be traced.

The second possibility is that the user can also pass the path of the executable which it wants to be traced. In this case, the user level module executes the executable (as a child process), obtains its *pid* and passes it onto the kernel module.

The trace buffer may contain data corresponding to other processes (because of other user module preferences). The reading function again has to take care to provide the user module with only the relevant data (of the concerned process).

3.7 User Access to Tracing

Security is one of the important issues that always needs to be addressed in case of tracing the system. Care has to be taken that any user can only see the tracing results of its own processes. In no way should the data of processes owned by other users be made available to the third party. Only the *superuser* has the right to trace all the processes.

In our case also, the trace buffer may contain data corresponding to other users and the reading function therefore arranges for the required data only.

Chapter 4

Implementation of Ktrace

Ktrace has been implemented on the Linux Operating System. Linux kernel version 2.6.18 has been used as the base for implementation. In this chapter, we look at the details of implementation.

4.1 Kernel Module

The kernel module needs to satisfy the following requirements (as discussed in chapter 3).

- A kernel level trace buffer to log the data.
- A mechanism to obtain the tracing options from the user level modules.
- A mechanism to filter the trace buffer and pass relevant data to the user level modules.

We implement these mechanisms using a device driver implementation. All requirements are easily satisfied by a device driver as described below.

- Being a kernel level module, a device driver can allocate a trace buffer by using the *kmalloc* kernel function.
- A memory based device driver supports the *ioctl* system call which allows user level programs to pass certain options to the kernel device driver.
- A memory based device driver also supports the *read* system call by which kernel data can be transferred to the user level programs after appropriate filtering as described in chapter 3.

A pseudo device name has been created as */dev/trace* and the kernel module acts as a driver for this device.

4.1.1 The Trace Device Driver

The *trace* device driver is implemented as a loadable device driver. Therefore the device driver has to implement the following additional functions.

- **trace_init** : This function is called by the kernel when the device driver is first loaded into the system (by using the *insmod* command [15]). This function mainly deals with two things - memory is assigned to the trace buffer associated with the driver and the device driver is registered with the kernel by using the major number and minor number of the device. According to the convention with the newer kernels, the device requests some major number from the kernel. The kernel assigns the major number depending upon the availability and the module then gets inserted into the kernel. The device name entry (*/dev/trace*) should then be created by reading the major number of the corresponding device driver from */proc/modules*.

- **trace_exit** : This function is called when the device driver is finally unloaded from the kernel (by using the *rmmod* command [16]). The main work done here includes freeing up trace buffer and any other memory areas allocated, deregistering the driver from the kernel etc.

Apart from the above mandatory functions, the device driver implements the following functions.

- **open** : This function is called when a process issues the *open* system call to open the */dev/trace* device.

- **close** : This function is called when the *close* system call is used with the file descriptor of the */dev/trace* as argument.

- **read** : This function is called when the *read* system call is issued to read from */dev/trace* device (i.e. from the trace buffer).

- **write** : This function is called when the *write* system call is used to write to the */dev/trace* device. This function is not implemented by the trace device driver as the *trace* device is a read-only device that provides the trace data.

- **ioctl** : This function is called when a process uses *ioctl* system call to configure certain options of the */dev/trace* device.

These functions are internally called by the operating system when a process makes the corresponding system call. A detailed description of device driver implementation can be found in *Linux Device Drivers* [17].

4.2 User Module

The user module has to accomplish the following things.

- It passes various tracing options to the kernel module - event mask, variable mask and *pid* of the relevant process to be traced. This is accomplished using the *ioctl()* system call provided by the *trace* device.

- The second aim of the user module is to obtain the data logged in the trace buffer. The *read* system call is used for this purpose.

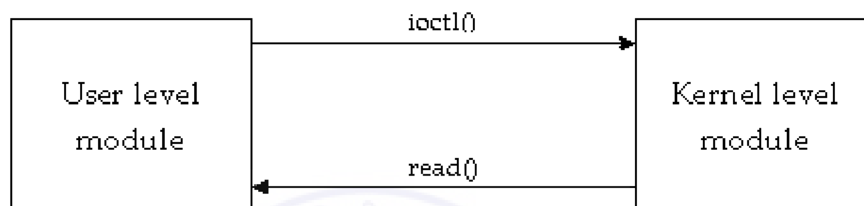


Figure 4.1: Interaction between User level and Kernel level modules

The user module first uses the *open()* system call to obtain a file descriptor for the *trace* device. It then calls *ioctl()* to pass options regarding event and variable masks and *pid* of the traced process to the kernel module. The kernel module then starts logging the data into its trace buffer. When the user module wants to exit, it can use *close()* system call.

4.3 The Trace Buffer

The kernel module allocates a fixed size trace buffer by using the *kmalloc* kernel function. Whenever a trace event is encountered, the following information is logged in the trace buffer.

- A pseudo variable to indicate the type of event encountered.

- The current time stamp.

- All variables that have changed since the previous recording.

The trace buffer is of limited size and can potentially overflow. There is a need to optimize the size needed for data being written into the trace buffer. We reduced the size needed for the buffer by recording the values of only those variables into the trace buffer which have changed since the last recording.

4.4 Data Logging

The kernel module implements a function which logs the data into the trace buffer. This function is called when a relevant event takes place inside the kernel. It takes the event type as a parameter. The definition of this function is the following.

```
void trace_dump(long event}
{
    if(event is in global_event_mask)
        store variables whose values have changed
}
```

The Linux kernel source has been modified to call this function whenever any traceable event takes place (i.e. an event that is supported by the tracing system). Calling this function is not straightforward as it involves some issues. These issues will be discussed later in section 4.8.

4.4.1 Extensibility of Ktrace

This section discusses how the logging of data leads to an easy extensibility of Ktrace. Each "variable" has the following fields.

- A Pointer to the current value (*ptr*) of the variable inside the kernel. This value is continuously updated inside the kernel as the execution proceeds.
- The last logged value of the variable (*last_value*).

The implementation of *trace_dump* function uses these information as follows.

```
void trace_dump(long event}
{
    if(event is in global_event_mask)
        for i = 1 to n
         if (variable[i] is in global_variable_mask &&
            (*ptr[i] != last_value[i]))
                last_value[i] = *ptr[i];
                record last_value[i] in trace buffer;
}
```

The pointers to the actual values of the variables are set up at the time of the initialization of the program. For the variables which exist directly in the kernel such as *totalram* (that stores the total memory of the system), setting up the pointer is straightforward as in *ptr[i] = &totalram*. In other cases, a new variable is introduced and the pointer is set up accordingly.

If need arises to introduce a new variable, it can be easily accomplished by following two steps.

- First, the pointer of the new variable needs to be set up.
- Secondly, the upper limit of the *for* loop in the code needs to be incremented by one.

4.5 Data Reading

The user module issues the *read()* system call for reading the data. This read is redirected by the Linux kernel to the *read()* function implemented by the *trace* device driver.

The kernel module obtains the data available in the trace buffer, filters it for the appropriate values and passes it over to the user module. A "read" pointer is maintained for each reader.

```
ssize_t read(.....)
{
    str = get filtered data from buffer;

    copy_to_user(str);
    return length(str);
}
```

There can be multiple active readers at the same time. Therefore, a list of "reader" structures is maintained. The structure of the list is discussed in section 4.7. A user module can start (using open) or exit (using close) at any time and hence traversal, insertion and deletion operations are performed on this list. However, we expect only a limited number of user modules running a trace at the same time. We use a linked list data structure for this purpose.

4.5.1 Extensibility of Ktrace

The reading function is the second reason for the easy extensibility of Ktrace. The *read* function simply transfers the data available in the trace buffer to the user level module. Since the data corresponding to the new variables is available in the trace buffer (due to *trace_dump*), it is transferred to the user module. Here also, nothing extra needs to be done for incorporating new variables.

4.6 Data Filtering

Data is logged in the trace buffer in a compact form - only those variables are recorded whose values have changed since the last recording.

The procedure to read implements filtering and other aspects as discussed earlier and described here. The data inside the trace buffer needs to filtered at the time of *read* and the criteria for filtering is based on the following.

- **Event**: The data logged at the time of writing corresponds to global event mask while a specific process is interested in only a subset of these events as specified in its event mask. Hence the data needs to be filtered depending on the event mask of the reader.
- **Pid**: The user module might be interested in only the data of a particular process. Therefore, the data needs to be filtered according to the specified *pid*.
- **Uid**: Only the data of user owned processes are provided to a particular user which involves filtering depending on the *uid* of the reader process.

4.6.1 Data Consistency

The filtering of data in the trace buffer may lead to inconsistency of data. The issue is explained with an example. Assume that there are two readers and both of them are interested in the same two variables - "v1" and "v2". But the first reader is only interested in event "e1" and the second is interested in event "e2". At some stage, the trace buffer might look like as shown in Figure 4.2.

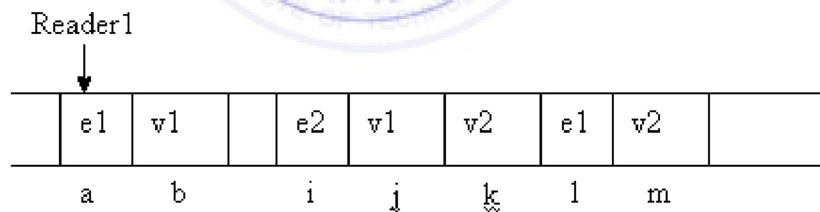


Figure 4.2: Trace Buffer snapshot at some time

Let us examine the first reader (reader1) when the "read pointer" is at location "a" (the first occurrence of e1). The value of v1 is transferred to reader1 along with e1. After some time, the read pointer reaches the location marked "i". Since reader1 is not interested in event e2, the values of v1 and v2 are not transferred to reader1. When the read pointer reaches location marked "l" (the second occurrence of e1), value of v2 is transferred to reader1. From reader1's perspective, the situation looks like the following after data transfer.

Current Event	State of Variables
e1	v1
e1	v2

The reader1 will interpret that the value of v1 remains the same even after the second occurrence of e1, which is clearly not true. This leads to a data inconsistency between the trace buffer and the reader.

4.6.2 State of Variables

To solve the problem of data inconsistency, a "state of variables" needs to be maintained for each active reader. For every user module, the kernel module stores the values of all the variables that the reader is interested in (i.e. included in variable mask). It maintains a list with the following two fields.

- The *last value* of each variable that has been seen by the corresponding user module while traversing the trace buffer.
- A *flag* indicating whether the stored value is the same as the value that has been transferred to the user module.

The list is updated as the trace buffer is traversed at the time of reading by the corresponding user module.

Let us go back to the scenario shown in Figure 4.2. There are two readers - the kernel module will maintain two separate lists of variables. We focus on reader1 again. After the first occurrence of event e1, the list corresponding to reader1 (let us say list1) will store the value of v1 and this data will also be transferred to reader1. The flag at this stage will be set to false for v1 because the transferred value is the same as the last stored value. When the "read pointer" reaches e2, no data will be transferred to reader1 but the list1 will still be updated. After e2 has been traversed, list1 will store the changed values of v1 and v2 and the corresponding flags will be set to true. List1 has been updated but no data has been transferred to reader1 because it is not interested in e2. When e1 is encountered again, value of v2 is updated in list1 and transferred (its flag switches to a false value). But at this stage, flag of v1 is also true and therefore the value of v1 is also transferred to reader1.

Thus, by maintaining a "state of variables" along with a "read pointer" per user module, we achieve data consistency. The trace buffer data needs further processing before being transferred to the user module because it also contains data corresponding to different processes and different users.

4.6.3 Filtering Mechanism

The case of filtering data on the basis of *pid* is discussed in this section. The other filtering is similar with minor differences.

The trace buffer contains several occurrences of the *pid* variable (whenever *pid's* value changes, it is logged into the buffer if any event is enabled to be traced as per the global event mask). A separate list is maintained corresponding to each occurrence of *pid* variable in the trace buffer. Each entry of the list stores the *index* of occurrence of the *pid* variable and its actual *value*. This list is sorted with respect to the *index* field.

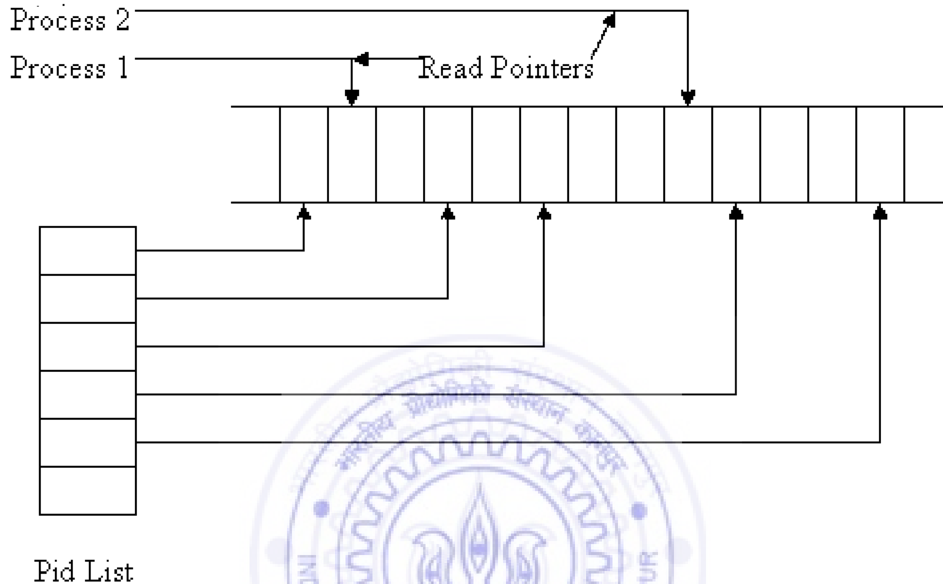


Figure 4.3: A Snapshot of Trace Buffer at some point of Time

Each time a user module calls *read*, the data is read from the current position of its read pointer. This data can only be transferred to the user module if the process which logged this data is the same as the process which the user module is interested in tracing. We already know in which process the user module is interested (from the *trace_pid* field of the "reader" structure of the current user module). But we need a way to find out the "owner" process of the data where the read pointer is currently located. This can be easily deciphered from the trace buffer - the last logged value of the *pid* variable in the trace buffer is the actual owner of the data. One way to do this is to move backwards through the trace buffer and locate the last occurrence of the *pid* variable.

A much quicker way is to maintain a list of occurrences of *pids* in the trace buffer and look for the previous *pid* by going through this list rather than going through the entire trace buffer. So, we find the pid under whose scope the data comes by using the "pid list" and the read pointer for that user module. For example in Figure 4.3, the read pointer of the first process comes under the scope of the first entry in the "pid list" while

the read pointer of the second process corresponds to the third entry of the "pid list". At this point of reading, the data will be returned to the first user module only if first entry of the "pid list" is the same as the *pid* it is interested in tracing.

Similarly for event and *uid* filtering, separate lists are maintained for the occurrence of events and the uid variables in the trace buffer and the corresponding data is filtered out.

4.6.3.1 Operations on the Sorted List

The sorted lists for filtering have to support the following operations.

- Insert : Whenever an event, *pid* or *uid* is encountered in the trace buffer for the first time, they are inserted into the corresponding lists. Care has to be taken that each such insertion takes place only once because these variables are encountered multiple times, once for each reading process.

- Look Up : The sorted list needs to provide the index under whose scope the currently read data comes. This function takes as argument the current read pointer and returns the corresponding index.

- Delete : The entries also have to be deleted from the lists. The deletions occur when the read pointer of each reading process goes beyond the scope of that entry.

The above operations enable the filtering of data and hence only the relevant data is returned to the user.

4.7 Global Event and Variable Masks

As discussed in section 3.5, the kernel module is only concerned with global event and variable masks at the time of data logging.

As mentioned earlier, when an event is encountered by the kernel, it is checked for relevancy, that is whether the event is included in the global event mask or not. When the event is relevant, all variables included in the global variable mask are recorded in the trace buffer if they have changed since the previous recording. This is implemented by *trace_ dump* function as shown below.

```
void trace_dump(long event}
{
    if(event & global_event_mask != 0)
        for i = 0 to numVariables
            if(variable[i].mask & global_variable_mask != 0)
                store variable[i] //if value changed
}
```

On the reader side, the kernel trace device driver provides data from the trace buffer to the reader processes. The trace buffer contains data corresponding to all the user level modules. Therefore at the time of reading, the data needs to be filtered according to the specifications of each user level module. A list of "reader" structures is maintained by the kernel module for this purpose.

4.7.1 Reader Data Structure

Each "reader" contains the following fields.

- **Pid**: The pid of the reader process. This field is set up at the time of opening the trace device.
- **Event Mask**: The mask of the events this particular reader is interested in. This field is set up by *ioctl()* call.
- **Variable Mask**: The mask of the variables this reader is interested in. This field is also set up by *ioctl()* call.
- **Trace Pid**: This field is also setup by the user module using the *ioctl* system call. If the user wants to trace a particular process, this field refers to the pid of that process.
- **Read Pointer**: The read pointer of this reader process. This field is updated regularly in *read()* system call.

During the read procedure, the kernel module looks up the reader structure corresponding to the current process (based on the *pid* field) and uses the event mask, variable mask and *pid* to return the appropriate data to the user.

The actual mechanism of filtering the data is described in section 4.6.

The user level module can start tracing (using *open* and *ioctl*) or exit (using *close*) at any time. Therefore, the global event and variable masks are updated at every invocation of *ioctl* and *close*. At the time of *ioctl*, one can just do a bit-wise OR to update the global masks. At the time of *close* by a reader, the entire list of readers is traversed to recalculate the global masks.

4.8 Integrating trace_dump with the Kernel

We implement the tracing functionality using a loadable module. In addition, the kernel is modified to make a call to *trace_dump* function at all the events of interest.

The problem is that the *trace_dump* function is not available till the tracing module is loaded to the kernel. Therefore, the *trace_dump* function can not be directly called as the module will not be loaded at all times the system is running. The *trace_dump*

function is only meaningful as long as the *trace* device driver is loaded into the kernel. We make use of a function pointer to solve this problem.

A function pointer is declared and initialized to a dummy trace function and is set to the *trace_dump* function when the module is loaded. It is again set to the *dummy_trace* function when the module is unloaded. The *dummy_trace* is just a dummy function having same prototype as *trace_dump* but it does not do anything.

Chapter 5

Results and Discussion

5.1 Introduction

In this chapter, we discuss how Ktrace can be applied to Linux applications to learn about their kernel activity. First, Ktrace is compared with existing work - */proc* and Strace. Then we give examples of certain applications which we traced using Ktrace and we were able to come up with solutions that can improve their performance. These applications include *Top*, a Linux command line utility which provides a dynamic real time view of a running system, and Xmag, a program which allows one to magnify portions of an X screen.

5.2 Comparison with */proc*

Most of the user space based profilers like *Top*, *Vmstat* etc. rely on the data provided by the kernel in */proc* directory. These user space profilers process the relevant data available in */proc* and display it in a particular format.

/proc provides information like - memory usage, network statistics, disk statistics, process specific information etc. Let's consider an application that uses */proc* to provide memory usage of the system.

/proc provides the memory usage of the system in a file named */proc/meminfo*. Any user space based profiler can be implemented by reading this file at fixed intervals of time.

The memory usage of the system can be observed by Ktrace by invoking the trace with the following masks.

- **Event Mask** : Two events are required - page allocation and page freeing.
- **Variable Mask** : Two variables are required - free memory of the system and total memory of the system.

The following graph was obtained by one such invocation of the tracer.

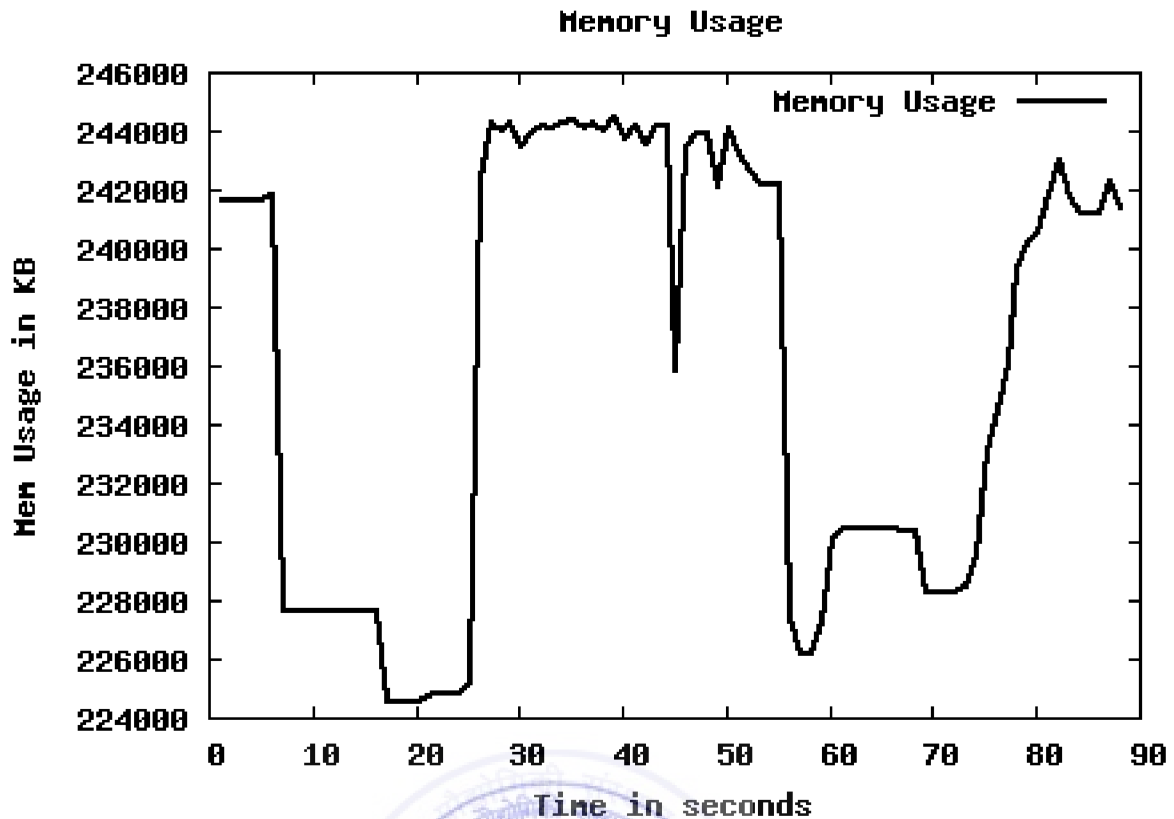


Figure 5.1: Memory Usage of the System with Time

Ktrace enables us to observe the memory usage every time memory is allocated or freed by the system. On the other hand, information provided by */proc* is only refreshed after regular intervals of time and it can not tell us about the events that were responsible for change in memory usage.

5.3 Comparison with Strace

Strace gives a list of all the system calls made by the process along with the arguments and the return value for each system call. The functionality provided by Strace can be achieved by Ktrace as it supports both the system call entry and exit events.

Ktrace needs to be invoked with following masks.

- **Event Mask** : Two events are required - system call entry and system call exit.
- **Variable Mask** : Only one variable is required - CPU_Registers. It stores the CPU registers of the process at the time of system call entry and system call exit. The CPU registers provide the arguments of the system call (at the time of entry) as well as the return value (at the time of exit).

The tracer can be invoked with the relevant event and variable masks to obtain the trace data. In addition, by tracing a particular process (by giving *pid* as argument), the

behavior of Strace can be emulated. Figure 5.2 shows a sample output from Ktrace. The trace data has been processed to make it look like the output of Strace.

```
open (134544812 0 438)  =  6
fstat64 (6 3237449404)  =  0
mmap2 (0 4096 3 34)  =  3103576064
read (6 3103576064 1024)  =  569
socketcall (1 3237449568)  =  7
ioctl (7 35093 3237449676)  =  0
ioctl (7 35111 3237449676)  =  0
socketcall (1 3237449476)  =  8
ioctl (8 35123 3237449564)  =  0
close (8)  =  0
socketcall (1 3237449216)  =  8
ioctl (8 35142 3237449556)  =  0
close (8)  =  0
close (7)  =  0
socketcall (1 3237449568)  =  7
ioctl (7 35093 3237449676)  =  157
write (2 3237437632 64)  =  64
ioctl (7 35111 3237449676)  =  0
socketcall (1 3237449476)  =  8
ioctl (8 35123 3237449564)  =  0
```

Figure 5.2: System Call Tracing for all Applications by Ktrace

Each line in the sample output describes one system call. The name of the system call is followed by arguments (at the time of execution) and then the return value is listed after the "=" sign. The arguments and the return value have very large numbers at times when they provide addresses.

But as discussed before, Strace functionality is limited to system call tracing only while Ktrace can trace many more events and variables at the same time.

5.4 Tracing Applications

The system call mechanism provides the interface for the programs to request services from the kernel. Each system call invocation requires a certain amount of overhead as it involves a transition from user mode to the kernel mode and back again. Also, system calls are the most common events occurring inside the Linux kernel. So our major experiments are concerned with the tracing of system calls.

In this section, we discuss results obtained by tracing system calls made by certain applications using Ktrace.

5.4.1 Top Utility

Top [5] provides a dynamic view of the system - displaying a summary of entire system state followed by a list of all the current processes. *Top* updates this information after a fixed interval of time which can be specified as an option at the command line. All this information is made available by the kernel in the */proc* directory and *Top* reads it from there. */proc* has certain files which specify overall system "state". Besides this, */proc* has one directory for each process and the process specific information in contained within these directories.

We decided to examine the system calls made by *Top*. The tracer was invoked with event and variable masks to trace the system calls (as described in section 5.3). Since we want to trace *Top* exclusively, the user module was invoked with a third parameter : *process* = *top* (the name of the executable which we want to trace). Some snapshots of the trace data are shown in the Figures 5.3 and 5.4. Figure 5.3 shows the first few system calls made by Top while Figure 5.4 shows the system calls made by Top after execution has been going on for some time. As explained in section 5.3, Figures 5.3 and 5.4 show the system calls made by *Top* along with their arguments and return values. After *Top* has been executing for some time, it shows a pattern in the system calls it issues. Figure 5.4 tries to capture the pattern by showing the system calls that *Top* issues repeatedly.

A summary of the trace data is shown in Table 5.1.

5.4.1.1 Analysis of Trace Data

As one can see in Figure 5.4, a *read* system call is preceded by an *open* system call and followed by a *close* system call. Table 5.1 also shows the fact that *open*, *read* and *close* system calls are much larger in number as compared to the others. This makes sense as *Top* has to go through every process specific directory in the */proc* directory to collect all the relevant information. So, *Top* takes the straightforward way of opening the specific files, reading the data and closing them as soon as the work is done. This would have been fine if *Top* had to do this only once. But being a dynamic picture of the system, *Top* has to refresh the information after a fixed interval and hence it has to repeat the

```
execve (134539383 3233405892 3233406132)  =  0

brk (0)  =  134578176

mmap (3236743680 4983912 4980660 4096)  =  3103412224

access (4962852 4)  =  254

open (4971342 0 0)  =  3

fstat64 (3 3236742652)  =  0

mmap (3236742620 255)  =  3103334400

close (3)  =  0

open (3103374868 0 0)  =  3
```

Figure 5.3: Initial System calls made by Top

open-read-close cycle again and again.

This shows where *Top* spends most of its time. Now, how can we improve this? During every refresh cycle, reading every process specific directory is mandatory because the data might have changed and the previously read data might have become obsolete.

During every refresh cycle, *Top* makes repeated calls to *open* and *close* system calls. This can be a major overhead if the process is present across multiple refresh cycles because then we are making unnecessary *open* and *close* system calls. The default refresh cycle of *Top* is 3s which is lower than the lifetime of many processes. This performance issue has been reported by others as well [18] and we have been able to identify this with Ktrace. We implement the solution provided by *Gregg* [18] to overcome this problem.

In this solution, each process specific directory and its children files are opened only once as long as the process is alive. The corresponding file descriptors are stored for further *read* system calls. When the process exits, the *close* system call is issued to free the file descriptors. This approach requires clever managing of file descriptors as opposed to the current approach where one can manage with a single file descriptor.

Another issue with this ideal approach is that every time the files have to be read from the start, so if the file is opened only once - then each call to *read* will increment the file offset (from where the data will be read next time). Then the *lseek* system call needs to be used to bring the file pointer to the beginning of the file again. This implies the use of another system call, the very issue we were trying to avoid. Fortunately, there is a work around to this problem also. The kernel provides a *pread* system call which takes the file offset (from where to read the data) as an argument. So, one can always use *pread* (with offset 0) rather than *read*.

Ktrace has helped us in understanding what does the process do. By looking at the trace data, we also have been able to come up with a solution that has the potential to

```
open (6275584 0 0)  =  4
read (4 6282016 1023)  =  133
close (4)  =  0
open (6275584 0 0)  =  4
read (4 6282016 1023)  =  14
close (4)  =  0
stat64 (134588996 3236741980)  =  0
open (6275584 0 0)  =  4
read (4 6282016 1023)  =  133
close (4)  =  0
open (6275584 0 0)  =  4
read (4 6282016 1023)  =  14
close (4)  =  0
stat64 (134588996 3236741980)  =  0
open (6275584 0 0)  =  4
read (4 6282016 1023)  =  132
close (4)  =  0
open (6275584 0 0)  =  4
read (4 6282016 1023)  =  14
close (4)  =  0
```

Figure 5.4: Repeated System calls made by Top

Table 5.1: Summary of System Calls made by Top

System Call	No of Invocations
open	3699
fstat64	42
close	3693
read	4113
stat64	1817
ioctl	43
rt_sigaction	682
gettimeofday	19
fcntl64	835
getdents64	38
newselect	18
alarm	1010
write	43

improve the performance. We know that system calls involve a significant overhead due to change from user mode to kernel mode and back again. But we wanted to get some estimate on how much performance gain can be obtained in this case. So we went ahead and implemented the alternative approach to evaluate the performance gain.

5.4.1.2 Improved Implementation

Top as a whole utility is not a very small application as far as source code is concerned and is not very easy to rewrite. We are more concerned with measuring the overhead of system calls rather than other aspects of *Top*.

We first wrote a small program which replicates the system call behavior of *Top* i.e. it processes the */proc* directory and opens the *status* file in each of the process specific directories. This program does not attempt to display any meaningful data to the user. It merely iterates through the */proc*, opening the *status* file of each process, reading it and then closing it. The pseudo-code of the program is shown below.

```
proc = opendir("/proc");

while(1)
{
    entry = readdir(proc);

    while(entry != NULL)
    {
        fd = open status file within entry if its a process directory
        read(fd, str);
```

```
            close(fd);

            entry = readdir(proc);
        }
        sleep(1);
    }
```

Next, we implemented a program which tries to avoid repeated opening and closing of files as long as the process is alive. A list of open files is maintained along with their file descriptors. Every time, */proc* is scanned, the list is searched to see if the file is already open. If it is, then the existing file descriptors can be reused. Otherwise, the file is opened and its file descriptor is stored in the list. The list is also cleaned to close the file descriptors of the processes which have exited. The pseudo-code of the program is shown below.

```
    proc = opendir("/proc");

    while(1)
    {
        entry = readdir(proc);

        while(entry != NULL)
        {
            this_fd = check if fd is present in list of open files;
            this_fd = if already not open, open status file and store
            fd;
            pread(this_fd, str, ..., 0);
        }
        clean(list of open files);//if process finished, close fd
        sleep(1);
    }
```

We ran both the programs to compare their CPU usages using the *Top* utility. The results are shown in Figure 5.5. As is evident in the graph, the improved implementation consumes considerably less amount of CPU.

5.4.2 Xmag Utility

The Xmag utility [19] allows one to magnify portions of an X screen. The region that is to be magnified can be selected at runtime and it is displayed in a separate graphical window.

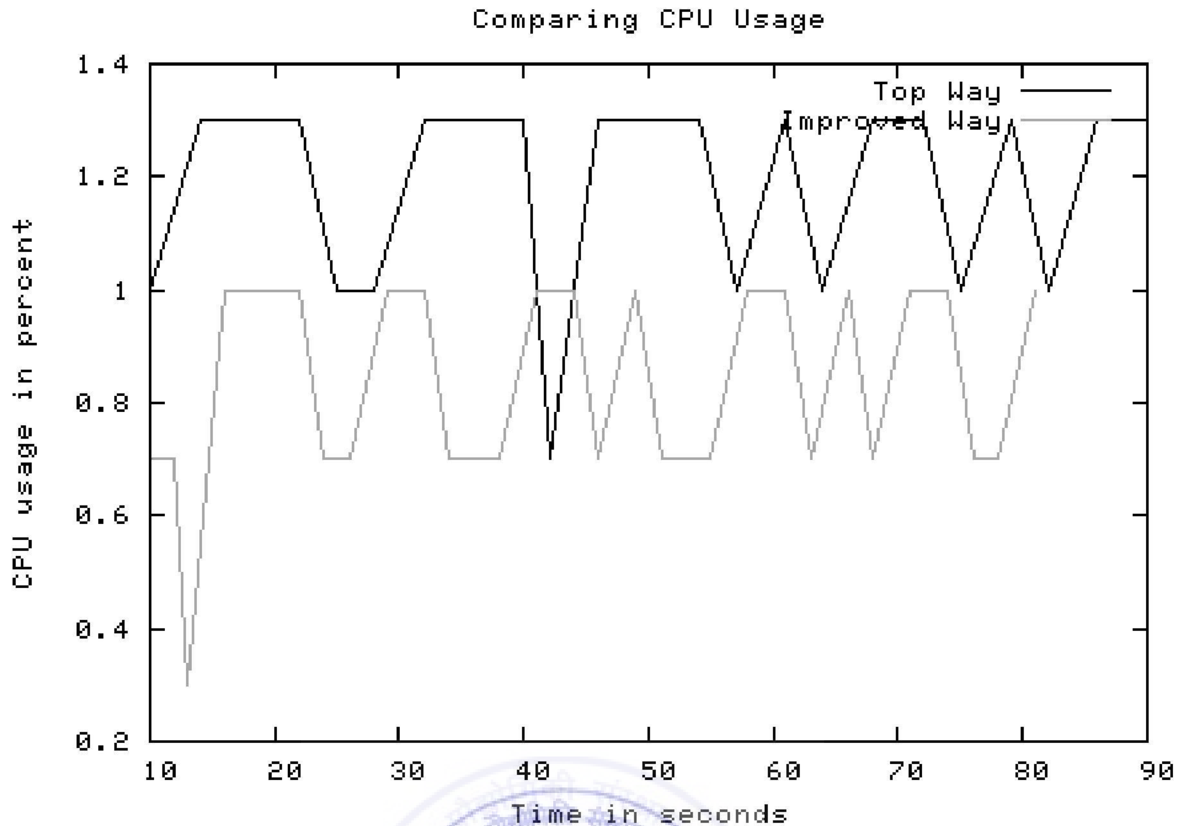


Figure 5.5: CPU Usage Comparison between two Implementations

We decided to do system call tracing on the Xmag program. The data obtained from the tracing is shown in Figure 5.6. Xmag also issues a set of system calls repeatedly after the initialization phase is over.

A summary of the trace data is shown in Table 5.2.

Figure 5.6 shows that the program is repeatedly issuing a certain set of system calls - *select*, *read*, *write*, *ioctl*, *gettimeofday*. Table 5.2 also supports the fact as number of invocations of these system calls are much more than the rest of the system calls. To find the answer to this, we need to do some analysis of the code.

5.4.2.1 Trace Data Analysis

Xmag is a graphical program built using X Toolkit Library and the Athena Widget Set, both of which are in turn based on the Xlib package. The Athena Widget set deals with the graphical items such as buttons, menus etc. The X Toolkit Library is the one which is responsible for the basic underlying display and the handling of the events.

A program specifies the events it is interested in, sets up necessary event handlers and then waits for the events to happen. The X Toolkit Library keeps track of the various events and passes the relevant events to the program. To accomplish this, the program calls a library function - *XtAppMainLoop*.

This function is implemented by the library - the function sits in a loop waiting for

```
newselect (4 3231276788 3231276916 3231277044)  =  1
ioctl (3 21531 3231276632)  =  0
read (3 3231274452 32)  =  32
ioctl (3 21531 3231277336)  =  0
write (3 134564568 52)  =  52
ioctl (3 21531 3231277336)  =  0
gettimeofday (3231277304 0)  =  0
newselect (4 3231276788 3231276916 3231277044)  =  2
ioctl (3 21531 3231276632)  =  0
read (3 3231274452 128)  =  128
ioctl (3 21531 3231276760)  =  0
ioctl (3 21531 3231277336)  =  0
write (3 134564568 4380)  =  4380
ioctl (3 21531 3231277336)  =  0
gettimeofday (3231277304 0)  =  0
newselect (4 3231276788 3231276916 3231277044)  =  1
ioctl (3 21531 3231276632)  =  0
read (3 3231274452 32)  =  32
ioctl (3 21531 3231277336)  =  0
ioctl (3 21531 3231277336)  =  0
```

Figure 5.6: Repeated System calls made by Xmag

Table 5.2: Summary of System Calls made by Xmag

System Call	No of Invocations
read	2683
open	322
fstat64	82
close	82
fcntl64	1
newselect	1594
write	1510
stat64	2
gettimeofday	1405
ioctl	3792

the events to happen and keeps passing them onto the program. This loop also calls *gettimeofday* in every iteration. This seems to be a bit strange. It turns out that the X Toolkit library provides a *TimeOut* feature. A program can also add certain timeouts and ask the library to be notified when the timeout expires. The library implements this feature by repeatedly checking in the loop whether any timeouts have occurred.

This clearly is not a very appropriate way to implement this scenario as system calls are being issued unnecessarily.

5.4.2.2 Alternative Approach

There is a need to notify a process when a certain timeout occurs. The *sleep* system call can be an alternative but the process might not necessarily want to go to sleep. The process might want to process certain events and then do something else when the timeout expires. So *sleep* is also not feasible.

The *alarm* system call sends a SIGALARM signal to the process after a specified number of seconds have elapsed. So a proper alternative can be to install a signal handler (which will inform the process that a timeout has occurred) for the SIGALARM signal. Then a call to *alarm* can be issued for the next timeout. This way, the process will be notified when a time out has occurred and busy waiting can be avoided.

This will get rid of a lot of unnecessary *gettimeofday* calls and can lead to an improvement in performance.

5.4.2.3 Other System Calls

In the above section, we have accounted for *gettimeofday* system calls, but there were a set of other system calls which were being called repeatedly - *select*, *write*, *read* and *ioctl*.

Select is being called from within the same waiting loop. It is used by the X Toolkit Library to wait for events to happen and then pass them on to the graphical program.

This has to be done and we can not think of another alternative.

The remaining system calls - *read. write* and *ioctl* are not called directly by either the Xmag program itself or the X Toolkit Library or the Athena Widget Set library. If one looks at the first argument of each of these system calls, it remains the same - 3 which is the first open file descriptor. The only possibility left is that these system calls are employed by the underlying Xlib package to communicate with the XServer and the first open file descriptor might have to do something with the display.

If that is the case, then this behavior should be common across all the graphical programs implemented using the X Toolkit library. We traced the system calls made by a couple of other such graphical programs - XCalc, Xgc. The data obtained was quite similar to the earlier trace data - the same loop kept repeating again and again. Every time some event occurred like a button was pressed or mouse was rolled over some button, the same system calls were issued.

We however did not do any further analysis on this.

5.5 Memory Related Examples

In the previous section, we focussed exclusively on the system calls and how their tracing can be useful. In this section, we examine certain memory aspects using the tracing system - namely how rapid allocation of memory leads to page faults in the system.

First, we traced a normally running system for number of page faults happening with time. Apart from the normal processes running on a PC, only the tracing was active. The following data was obtained.

Total Page Faults 1297 (over 1minute 26.5seconds)

This gives a rate of of about **15** page faults per second. The tracing was done under the *time* command.

Next time the tracing was started, several additional graphical applications were fired up which included Web Browser, Konqueror, Terminal and KSysGuard. The tracing was stopped as soon as all these applications were up and running. The number of page faults were again counted and results were as follows.

Total Page Faults 72040 (over 39.1seconds)

This gives a rate of of about **1847** page faults per second which is about a 100 times more than that of the normal system. The increase makes sense because X applications are "memory hungry" and hence the sum of the size of virtual memories of the applications will be much more than the total physical memory leading to a large number of page faults.

The idea behind stopping the tracing as soon as the applications were up is that page faults should increase dramatically when the application starts execution because it is during this time that most of the memory allocations will be done. Once the applications are up, memory would have been allocated and rate of change of page faults should return back to normal.

After this, we wrote a small program which repeatedly called *malloc* without any calls to *free*. Such a program is killed by the kernel after some time due to insufficient memory. The tracing was started, the program was executed and the tracing was stopped when the programm was killed. The results were as follows.

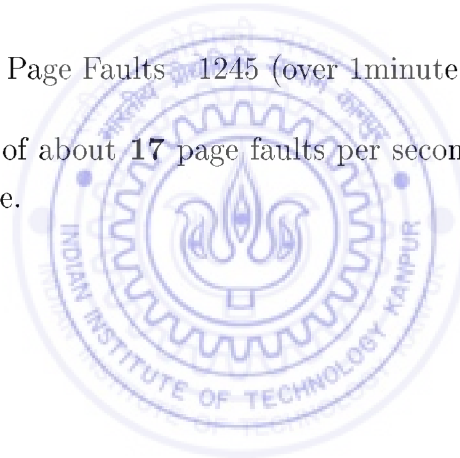
Total Page Faults 180661 (over 1minute 24.7seconds)

This gives a rate of of about **2140** page faults per second upholding the fact that page faults are dramatically affected by memory hungry programs.

To see if there were any after effects on the system, we again measured the rate of change of page faults on a normal system (where only the tracing was active). The results showed no such after effects.

Total Page Faults 1245 (over 1minute 11.1seconds)

This gives a rate of of about **17** page faults per second. The system seemed to have returned to normal state.



Chapter 6

Conclusion and Future Work

6.1 Conclusion

We have devised and implemented a mechanism to observe the Linux kernel. The tracing framework implemented as a part of the thesis provides a way to reconstruct the entire system activity during the time tracing is active. By examining the trace data, one can determine what events happened and what were their effects on the system. Ktrace can not only do the work of existing trace facilities on Linux like */proc*, Strace but can do more than that.

Ktrace was successfully applied to observe two applications - Top and Xmag. We were able to understand what the applications were doing at the kernel level and we were also able to come up with certain solutions which would help in improving the performance of the applications.

The thesis has been able to show that understanding the kernel workings can be very useful in optimally designing an application. Various applications have to request the kernel for certain services and it is only through proper examination of the kernel proceedings that one can figure out how to optimize the code.

6.2 Future Work

The system has been developed with a limited number of events and variables. It needs to be augmented with more events and variables to bring the kernel under complete observation. Finding more relevant events and variables will require closer scrutiny of the kernel source code. The existing sets of events and variables cover broader aspects of system calls, memory, signals, network statistics etc. but there is a need to delve into the deeper details. Certain parts of the kernel like interrupts, modules have not been covered at all while other areas like network, I/O etc need to examined for more events and variables.

Secondly, more applications can be traced using Ktrace to examine their kernel level

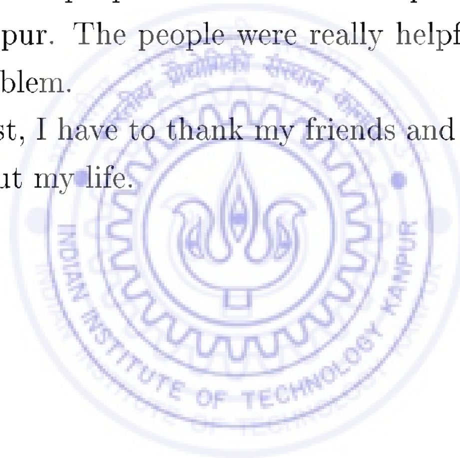
behavior. We have examined some applications in this thesis to show how Ktrace can be used to identify performance issues in an application. Next step can be to trace a significantly complex application like OpenOffice and observe the results. The tracing will require considerable work as the application is quite complex and might be doing multiple things simultaneously.

Acknowledgement

This work would not have been possible without the support of my thesis supervisors - Prof. Dheeraj Sanghi and Prof. Rajat Moona. I am grateful to them for assisting me in numerous ways ranging from proof reading the thesis to discussions on the problems encountered during the work. Without their invaluable help and guidance throughout the thesis, I would have been really lost.

I am also thankful to the people and the staff of Department of Computer Science and Engineering at IIT Kanpur. The people were really helpful and they did whatever they could in case of any problem.

Last but not the least, I have to thank my friends and family who believed in me and supported me throughout my life.



Navdeep Bhulli
August 2007
Indian Institute of Technology, Kanpur

References

[1] "Strace Manual Page," http://www.linuxmanpages.com/man1/strace.1.php.

[2] "Linux Kernel Debugger," http://oss.sgi.com/projects/kdb/.

[3] "DTrace Home Page," http://www.sun.com/bigadmin/content/dtrace/.

[4] "Proc Filesystem Manual Page," http://www.linuxmanpages.com/man5/proc.5.php.

[5] "Top Linux Utility Manual Page," http://www.linuxmanpages.com/man1/top.1.php.

[6] "Vmstat Linux Utility Manual Page," http://www.linuxmanpages.com/man8/vmstat.8.php.

[7] "Ptrace Manual Page," http://www.linuxmanpages.com/man2/ptrace.2.php.

[8] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *USENIX 2004 Annual Technical Conference*, 2004.

[9] "Solaris Dynamic Tracing Guide," http://docs.sun.com/app/docs/doc/817-6223.

[10] "Page Allocation Function in Linux Kernel Source Code," http://lxr.linux.no/source/mm/page_alloc.c?v=2.6.18#L910.

[11] "Page Freeing Function in Linux Kernel Source Code," http://lxr.linux.no/source/mm/page_alloc.c?v=2.6.18#L1116.

[12] "Schedule In Event in Linux Kernel Source Code," http://lxr.linux.no/source/kernel/sched.c?v=2.6.18#L1803.

[13] "Schedule Out Event in Linux Kernel Source Code," http://lxr.linux.no/source/kernel/sched.c?v=2.6.18#L1831.

[14] "Abstract Syntax Notatrion One," http://asn1.elibel.tm.fr/.

[15] "Insmod Linux Program Manual Page," http://www.linuxmanpages.com/man8/insmod.8.php.

[16] "Rmmod Linux Program Manual Page," http://www.linuxmanpages.com/man8/rmmod.8.php.

[17] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers,*. O'Reilly Media Inc., 2005.

[18] "Top Performance Issue," http://www.brendangregg.com/DTrace/prstatvstop.html.

[19] "Xmag Linux Utility Manual Page," http://www.linuxmanpages.com/man1/xmag.1x.php.

Appendix A

Encoding Rules for Buffer Data

A.1 Introduction

As we have discussed in this thesis, the data is logged in the kernel buffer in the form of Tag, Length and Value triplets. The mapping of data into the corresponding triplets is governed by the Basic Encoding Rules of ASN standard.

A.1.1 Encoding of the Tag Field

Each entity is assigned a unique tag. The tag field is encoded in the following way.

If tag <= 30, then the encoding is described in Figure A.1.

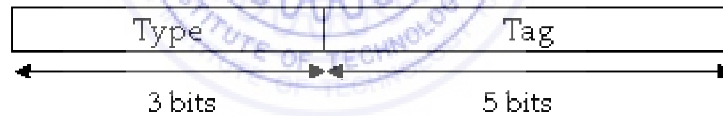


Figure A.1: Tag Encoding - Case 1

ASN provides the abstraction to divide the data into several classes - universal, application, private etc. The 3 bits are used to determine what class the data belongs to. So, effectively only 5 bits are available for tag encoding. But if tag >= 31, then the encoding is described in Figure A.2.

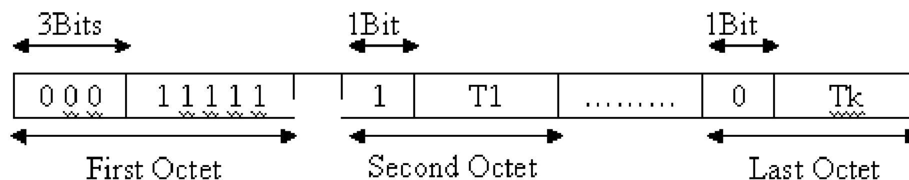


Figure A.2: Tag Encoding - Case 2

and Tag = T1 + T2 + ... + Tk.

A.1.2 Encoding of the Length Field

Let the length of the data be L octets. If $L < 128$, then the encoding is as shown in Figure A.3.

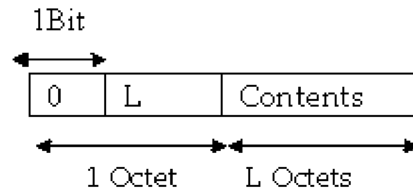


Figure A.3: Length Encoding - Case 1

If $L > 128$, then the corresponding encoding is shown in Figure A.4.

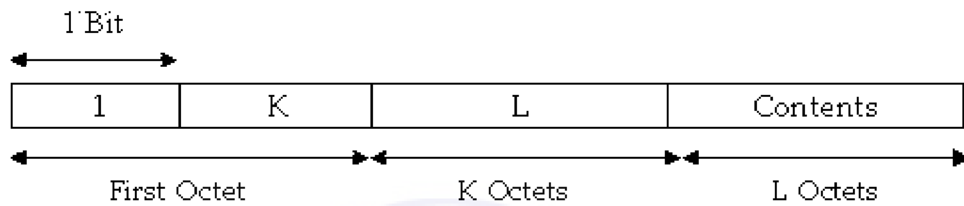
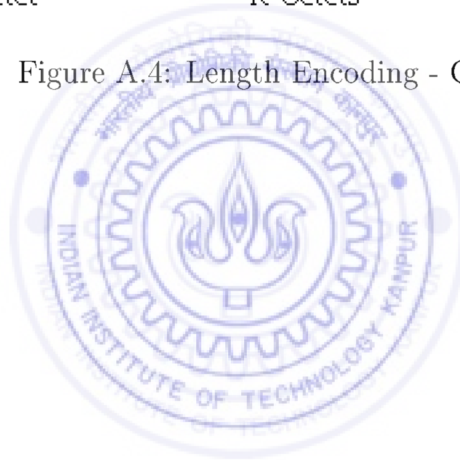


Figure A.4: Length Encoding - Case 2



Appendix B

Inserting the Trace Module

This section describes how to get the trace module up and running. The *insmod* command needs to be issued from the source directory of the trace module.

```
$$ insmod trace.ko
```

The trace module uses a kernel assigned major number (which is the convention in the newer kernels). The */proc/devices* file needs to be examined to find what major number has been assigned to the *trace* module. This can be done as shown below.

```
$$ MAJOR=‘grep trace /proc/devices | awk ’{print $1}’‘
```

Now, the *trace* device can be created with the *mknod* command.

```
$$ mknod /dev/trace c $MAJOR 0
```

The tracing can be started by using the user level module as described in the thesis.