ISSN: 2319-8753

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 10, October 2014

Implementation of Character – Mode Device Driver to Read the Processors GDT

Nithya Easwaran

Faculty, Sri Vani International School, Bangalore, Karnataka, India

ABSTRACT: There are so many devices coming in the market all which need device drivers. Writing device driver is OS specific. In Linux, device driver will be added dynamically. All the current information including added modules will be updated in /proc file system in Linux. Every process will be having its own process address space which can be viewed by the administrator. The central concept of virtual memory is the insertion of a level of indirection between the memory address space seen by a program and the address space of the actual memory in the system. The addresses are seen by a program as virtual addresses; they are also sometimes called logical addresses, and simply addresses when the context makes it clear that they pertain to the address space of a particular program

KEYWORDS: device driver, linux, administrator, virtual addresses

I. INTRODUCTION

Introduction to Device Driver

A device driver or software driver is a computer program allowing higher level computer programs to interact with a device. A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware is connected.

When a calling program invokes routine in the driver, the driver issues command to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware dependent and operating system specific.

Making hardware work is tedious. To write to a hard risk, for example, requires that you write magic numbers in magic places, wait for the hard drive to say that it is ready to receive data, and then feed it the data it wants, very carefully.

To write to a floppy disk is even harder, and requires that the program supervise the floppy disk drive almost constantly while it is running. Instead of putting code in each application you write to control each device, you share the code between applications. To make sure that code is not compromised, you protect it from users and normal programs that use it. If you do it right, you will be able to add and remove devices from your system without changing your applications at all.

All devices controlled by the same device driver are given the same major number, and of those with the same **major numbers,** different devices are distinguished by different **minor numbers**.

A device driver simplifies programming by acting as a translator between a device and the applications or operating systems that use it. The higher level code can be written independently of whatever specific hardware device it may control.

The x86 ISA provides two distinct methods to support protection between programs: segmentation and paging. An x86based processor translates each virtual address into a linear address using segmentation, and then optionally translates each linear address into a physical address using paging. If paging is not used, the linear address is used

DOI: 10.15680/IJIRSET.2014.0310056

ISSN: 2319-8753

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 10, October 2014

directly as a physical address. A segment is a contiguous portion of an address space, such as the 32 bit space of physical addresses. Segments are described in one of two arrays called descriptor tables. The Global Descriptor Table, or GDT, can be used by any program, while each program can also have its own Local Descriptor Table, or LDT. A pointer (a physical address) to the GDT is held in the 48bit GDT register (GDTR) along with a 16 bit limit (size 1 in bytes).

When you write device drivers, it's important to make the distinction between "user space" and "kernel space".

- Kernel space: Linux (which is a kernel) manages the machine's hardware in a simple and efficient manner, offering the user a simple and uniform programming interface.
- Bridge or interface between the end user programmer and the hardware. Any subroutines or functions forming part of the kernel (modules and device drivers, for example) are considered to be part of Kernel space.
- User Space. End user programs, like the UNIX shell or other GUI based applications (presenter for example), are part of the user space.Obviously, these applications need to interact with the system's hardware. However, they don't do so directly, but through the kernel supported functions.

Distributed Shared Memory in Kernel Mode – Thobias S.Trevisan Victor Santos Costa Lauro Whately Claudio.L.Amorim [1]

In this Paper, we introduce MOMEMTO (MORE MEMORY THAN OTHERS) a new set of kernel mechanisms that allow users to have full control of the distributed shared memory on a cluster of personal computers. MOMEMTO has been implemented in the Linux 2.4 kernel and preliminary performance results show that MOMEMTO has low memory management and communication overheads and that it can indeed perform very well for large memory configurations. First, we observe that many parallel applications. Ranging from NAS –benchmarks [9] to Database Management systems, often need not only Processor but also memory scalability. For instance, assume a 16-node cluster where each node has 512 MB, running either a large class-A NAS benchmark or large size database that requires 8 GB of memory.

Making Linux Interrupts Tasked By Zhao Hogweed, Fang Kechi,Zang Xuebai Research of Technology

This paper illustrates the Linux Interrupt handling mechanism. After analyzing its interrupts mechanism, a new solution was designed to make interrupts handle as a kernel thread. A special interrupt can be handled immediately if only it has a higher priority, otherwise it will return and a service thread will take control of it. A new solution has been aroused for the tasking of interrupts. This solution handled interrupts as a requesting and serving pattern and solved the problem of real time tasks disturbed by interrupts. The real-time performance of embedded Linux was effectively improved.

Application defined Scheduler Implementation in RTLinux [3] By J.Vidal, I. Ripoll,A.Crespo and P.Balbastre.

This paper illustrates that scheduling theory has been one of the most active and fertile areas in real-time. The result of this intensive research was a large number of scheduling algorithms. Most of the times, this policies come to live due to the great variety of applications of real-time and the inherent peculiarities to each one. For example, in multimedia applications CPU reservation was initially implemented using a_xed priority scheduler, but now it has successively been adapted to dynamic priorities, to obtain better CPU utilization and easy reclaiming of the unused CPU time. In control applications, however, it is mandatory to remove or, at least, minimize the delay of control tasks. Fixed priority schedulers do not accomplish this requirement, and several proposals of new scheduling algorithms achieve a better control performance of the system.

Checking Array Bound Violation Using Segmentation Hardware [4] By Lapchung Lam Tzicker Chiueh.

The ability to check memory references against their associated array/buffer bounds helps programmers to detect programming errors involving address overruns early on and thus avoid many difficult us down the line. This paper proposes a novel approach called Cash to the array bound checking problem that exploits the segmentation feature in the virtual memory hardware of the X86 architecture. The Cash approach allocates a separate segment to

DOI: 10.15680/IJIRSET.2014.0310056

ISSN: 2319-8753

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 10, October 2014

each static array or dynamically allocated buffer, and generates the instructions for array references in such a way that the segment limit check X86's virtual memory protection mechanism performs the necessary array bound checking for free. In those cases that hardware bound checking is not possible, it falls back to software bound checking. As a result, Cash does to need to pay per reference software checking overhead in most cases.

II. OBJECTIVES

- To review the literature on Linux character device drivers,pro file system in Linux kernel and related parts of the kernel
- To arrive the design specifications for the character mode device driver for the processors installed proc file system.
- To implement the designed character device driver
- To test the implemented device driver for the normal user

III. METHODOLOGY

- Literature review on Linux character device drivers, proc file system in Linux kernel and related parts of the kernel is carried out by referring the books, journal papers and websites.
- The design specifications for the character mode device driver for the processors installed physical memory is arrived at from the literature review.
- The character mode device driver for the processors installed physical memory is designed using the arrived design specifications.
- The C code is developed for the designed character mode device driver using the kernel module programming concepts
- The developed C code will be implemented on the Pentium processor based system
- The working of the implemented device driver is verified by writing the user program for the normal user.

IV.CONCLUSION

The device drivers and kernel level programming has been a new experience for kernel device programmers. It has given a lot of scope in memory mapping, segmentation. Kernel programmers are subjected to constraints that do not show in user level programming. We need to look at the design constraints and manipulate device registers to get kernel working.

All GDTs are stored in the cpu_gdt_table array, while the addresses and sizes of the GDTs (used when intialising the gdtr registers) are stored in the cpu_gdt_descrarray.

The working of the implemented device driver is verified by writing the user program for the normal user.

REFERENCES

1. Niel Mathew and Richard stone, "Beginning Linux Programming", Wiley Publishing., Inc.,2004
2. Daniel.P.Bovet, Marco Cesati, "Understanding the Linux Kernel", O'Reilly, 2005
3. Alessandro Rubini & Jonathan, "Linux device drivers", O'Reilly, 2001.