DOS Virtualized in the Linux Kernel

Robert T. Johnson, III

Abstract

Due to the heavy dominance of Microsoft Windows® in the desktop market, some members of the software industry believe that new operating systems must be able to run Windows® applications to compete in the marketplace. However, running applications not native to the operating system generally causes their performance to degrade significantly. When the application and the operating system were written to run on the same machine architecture, all of the instructions in the application can still be directly executed under the new operating system. Some will only need to be interpreted differently to provide the required functionality. This paper investigates the feasibility and potential to speed up the performance of such applications by including the support needed to run them directly in the kernel. In order to avoid the impact to the kernel when these applications are not running, the needed support was built as a loadable kernel module.

1 Introduction

New operating systems face significant challenges in gaining consumer acceptance in the desktop marketplace. One of the first realizations that must be made is that the majority of this market consists of non-technical users who are unlikely to either understand or have the desire to understand various technical details about why the new operating system is "better" than a competitor's. This means that such details are extremely unlikely to sway a large amount of users towards the operating system by themselves.

The incentive for a consumer to continue using their existing operating system or only upgrade to one that is backwards compatible is also very strong due to the importance of application software. Even though attempts are made to make software easy to use, most consumers still spend large amounts of time learning how to be productive with new software. While open source software is available, it is not the dominant business model used by most corporations. This means that consumers also spend large amounts of money purchasing new software. The cost of application software easily exceeds the cost of the operating system, and potentially even exceeds the cost of the entire computing system. If a new operating system is incompatible with the application software currently used by the consumer, the operating system will render the old software useless. This loss of investment in both time and money creates a very strong incentive for a consumer to only migrate to an operating system that is backwards compatible if their current one is ever replaced.

Finally, that is only one of the challenges facing a vendor attempting to introduce a new operating system into the marketplace. Even if large numbers of consumers can be persuaded that their loss of investment is worth the price of migration to the operating system, they will want new applications to be available to replace their existing ones before doing so. Due to the impracticalities of a vendor providing replacements for all possible applications, it is crucial that the vendor has support from other developers to do so. However, developers will have little incentive to develop these applications since the operating system has a low installed base and there are few opportunities to profit from the development. This creates a difficult to solve "chicken or the egg" type of problem where consumers want software to be available before migration that will not be available until after they migrate. Such a high barrier to entry significantly reduces possible competition in the desktop market. Further exacerbating the problem is the current dominance of Microsoft Windows®, which holds the vast majority of the market. Strong competition in the marketplace fuels the drive for competitors to provide the highest quality products for the lowest cost possible. This is clearly seen in the continuing "CPU wars" between AMD® and Intel®. While purely speculative, it is unlikely that Intel® would currently be shipping 3.2GHz processors for commodity prices without the strong competition from AMD®.

This has given rise to speculation that for an operating system to compete with Microsoft Windows® in the desktop market, it must be capable of running its applications. While not an easy task due to problems like undocumented API's and the general black box nature of the system, it has been attempted with some success. This paper investigates the possibility of increasing the performance of such attempts to run applications written for an operating system very dissimilar from the host operating system when both were written for the same machine architecture. The virtualized services are implemented as part of the kernel as a loadable module, both to minimize the overhead of unnecessary context switches and the impact to the kernel when the applications dependent on these virtualized services are not being run. A loadable module was written for the Linux operating system that allows some DOS applications to be run under it. The target applications were ones that only used DOS services and did not attempt to execute privileged instructions or access the hardware directly.

In the next section, related work will be discussed. Section 3 will discuss background related to DOS application programming to acquaint the reader with any needed terminology and Section 4 will discuss the implementation details of the module and the changes made to the stock kernel. Section 5 will discuss the testing methodology and the results. Finally, Section 6 will summarize my conclusions.

2 Related Work

DOSEMU is a DOS emulator capable of running DOS programs under Linux [1]. It requires some version of DOS in order to perform the emulation and run the programs. The project appears to have been abandoned, probably due to the impact of the Wine project. However, apparently it was first due to this project that the system call vm86 was incorporated into the kernel [2]. This call was later renamed vm86old and a new vm86 was created. The only apparent difference is that debugging support was included in the new version. The vm86 call was used in the implementation of the module and will be discussed further in a related section.

Wine is a regular user mode program capable of running some DOS and Windows® applications under Linux [3]. This is done by creating a process known as the Wine Server that provides the functionality needed by the Windows® or DOS processes it spawns. This project has been in development for years with developers constantly trying to improve it and chase after the continually expanding Win32 API. It provides good support for running some key Windows® applications. However, the performance of many applications is likely to be slower than their native counterparts. This is due at least in part to the fact that the program runs entirely as a user mode process in Linux and potentially pays penalties due to extra context switch overhead.

The latest, stable version of the Linux kernel, 2.4.22 as of this writing, contains support for running applications written for other variants of Unix [4]. This is done by associating a personality to each running process. Depending on the personality of the process, various aspects of the kernel are handled differently, such as system calls and signals. The structures linux_binfmt and exec_domain are used to accomplish this. Since personality and both structures are used in the module implementation, they will also be discussed in more detail in a related section.

3 DOS Assembly Programming

DOS programs were originally written to run in what is known as real-address mode in the Intel IA-32® (x86) architecture. For all intents and purposes, this mode models the original 8086 processor. Memory addresses are computed by adding the appropriate segment and offset registers together. The segment register is left shifted by four bits before being added to the offset register. Therefore the processor has 20 bit addresses and a total of 1 megabyte of memory can be accessed. All addresses are physical addresses and there are no protection mechanisms for any address. Due to the segment registers, memory is partitioned into 64-kilobyte segments. Later programs used DOS extenders to put the processor into protected mode so they could access more memory. Protected mode includes support for virtual memory, memory protection and a 32-bit flat address space. However, this module only supports the original real mode programs.

There are seven memory models available to the DOS assembly programmer. Except for the flat memory model, which is only available in protected mode, the models are used for real mode programs and differ on the total amount of memory segments allowed for code and data. The small memory model allocates one code and one data segment for the program, and was the model used for the DOS programs run by the module during testing.

Four of the general-purpose registers are named ax, bx, cx, and dx. They are 16-bit registers and the eight most significant or least significant bits can be used as an 8-bit register by dropping the x and appending either an h or l respectively (ex. ah). When the 386 processor was introduced, the 16-bit registers were extended to 32-bits. The extended registers have an e in front of their 16-bit names (ex. eax).

DOS services are invoked by executing the int 0x21 instruction. This instruction triggers the external interrupt 32 in the processor so the appropriate interrupt handler will be invoked. The interrupt is called external since programs can trigger it. Over 50 DOS services are available through this interrupt. The desired service is identified by the number in the ah register and other parameters specific to the service are passed in various other registers.

The program segment prefix consists of 256 bytes that are stored directly before the position in memory that the program is loaded into. This prefix contains the number of characters stored in the command tail, up to the maximum of 127, at offset 128 in the prefix. The characters in the tail follow at offset 129, and contain exactly what is typed on the command line after the program name, up to the maximum allowed.

4 Implementation

Almost the entire functionality needed was written inside the module. The only changes required to the stock kernel code involved adding a total of eight lines to two files. The various details involved in the implementation will be discussed in turn in the appropriate subsection.

4.1 16-bit versus 32-bit code

There is at least one important difference between the instructions generated for 16-bit real mode and 32-bit protected mode programs. This difference is the default operand size used by the processor in each mode. The following example illustrates this difference with a simple instruction that moves an immediate value into a register.

	16-bit real mode	32-bit protected mode
mov ax, 0x1111	0xB81111	0x66B81111
mov eax, 0x11111111	0x66B811111111	0xB811111111

The operand size override prefix is reversed for the two instructions due to the different default operand sizes of the two modes. This means that 16-bit real mode code cannot be executed directly by a 32-bit protected mode operating system such as Linux. However, to allow such operating systems to run such programs the Intel IA-32® architecture includes a quasi-mode called virtual-8086 mode that is used in conjunction with protected mode. Setting the VM flag in the eflags register will put the processor in this mode. While in this mode, the processor is still running in protected mode, but the 16-bit real mode programs will run correctly.

Linux includes support for an application to put itself into virtual-8086 mode with the vm86 and vm86old system calls. The vm86 call was used so that the DOS process would first make the transition from protected mode into virtual-8086 mode before running the original program. The kernel code in vm86.c that supports the mode assumes that a parameter will be present on the kernel stack upon return from the mode. This assumption is violated if the process was put directly into virtual-8086 mode by the module. Therefore this method avoids rewriting significant parts of kernel code with only a small performance penalty. The vm86 call is invoked with parameters signaling that the mode should be entered and the initial state of the registers upon entry.

4.2 Module Initialization And Removal

When the module is first inserted into the kernel various initialization steps will be taken. First, pointers are saved to needed system calls from the current system call table. These will be used by the DOS services that are currently implemented. The calls must be retrieved from the table since they are made directly available to modules in the current kernel. However, it would be better if the system calls were directly available to avoid any potential problem with system calls that have been replaced by other modules, and remains as a possible enhancement to the current implementation. After saving the calls, the DOS binary format and execution domain are registered with the kernel. When the module is removed, these will be unregistered.

The registered binary format is a linux_binfmt structure that contains a pointer to the function that will load the binary into memory and a pointer to the module where the function resides in. Other members of the structure are unused. The registered execution domain is an exec_domain structure that is named DOS, supports only a single personality, contains a pointer to the function that implements the system calls of the domain and a pointer to the module where the function resides in. Again, other members of the structure are unused.

4.3 DOS File Format

A page, which is also called a block, is defined to be 512 bytes and a paragraph is defined to be 16 bytes. All of the entries in the table are 16-bit quantities except for the two initial signature bytes as well as the code and data of the program.

File Offset	Description
0	First signature byte representing the ASCII value for M
1	Second signature byte representing the ASCII value for Z
2	Actual number of bytes in the last page of the file, or zero if it is a full page
4	Total number of pages in the file
6	Total number of relocation entries in the file, zero entries are possible
8	Total number of paragraphs used for the header
10	Minimum number of paragraphs allocated in addition to the code size
12	Maximum number of paragraphs allocated in addition to the code size
14	Stack segment value to be added to the segment the program was loaded at
16	The initial stack pointer value
18	The 16-bit file checksum, which was generally just set to zero, but if set properly, the sum of all 16-bit quantities in the file will be zero
20	The initial instruction pointer value
22	Code segment value to be added to the segment the program was loaded at
24	Offset of the first relocation entry in the file
26	Overlay Number, zero means this file contains the main program
?	Offset of the first relocation entry in the program, stored at offset 24
? + 2	Segment of the first relocation entry in the program, relative to the code segment

…	Remaining relocation entries
??	The code and data of the program, this offset is calculated by multiplying the value at offset 8 by the size of a paragraph

The minimum and maximum amounts of memory are most likely irrelevant since the program is running in a virtual address space. Due to lack of information, it is unknown what a non-zero overlay number means. This is possibly some form of DOS library or how segments for the larger memory models were stored on disk. Relocation entries are segment addresses that must be adjusted if the code segment is not loaded into memory at the address used during assembly. This was generally address zero, but could be controlled by the programmer.

Certain values are also defined to be present besides the ones specified above upon program entry. Either zero or the number of characters in the command tail is stored in ax. The 32-bit combined value representing the available memory for the program is stored in bx and cx, with bx representing the most significant bits. The dx register holds the value zero. Finally, both the extended segment and data segment registers hold the segment address of the program segment prefix.

4.4 Loading The Binary Into Memory

Loading the program into memory is accomplished by the load_dos_binary function registered with the kernel inside the module. When a program is invoked by a user, the kernel reads in the first 128 bytes of the program binary and stores it in a linux_binprm structure that contains this information as well as other information needed by program loaders, such as the command line arguments the program was invoked with. This structure along with another structure that represents the machine registers is passed to the loader. The kernel simply calls each registered loader in some order until one of them recognizes the executable or the list is exhausted.

When the DOS loader is called, both the signature bytes and the file length are examined to determine if the binary is the appropriate type. If either of these does not match what is expected, the loader returns that the binary was unrecognized. This allows the kernel to continue calling the other loaders. Otherwise, all the relevant information in the header is recorded so that the initial execution environment may be set up correctly. Since the program segment prefix will start at virtual address zero, both the stack segment and code segment values are updated to reflect this. This involves right shifting the starting virtual address of the program (256) four bits and adding the result to the recorded values. Generation of all segment portions of a virtual address involve right shifting by four bits, while generation of all virtual addresses involve left shifting the segment value by four bits and adding the offset value to the result. Future discussion of address generation will not mention this explicitly, but it is implied.

Due to the way Linux spawns new processes, traces of the previous process that was forked must still be flushed. These traces include such things as the virtual memory mappings associated with the forked process. The new DOS personality of the process is also set, which will be important later. Now a new virtual address space must be set up. This involves associating appropriate virtual addresses for the code segment, data segment and stack segments of the process. Since DOS programs are not segmented like regular Linux processes, these segments are set to overlap. An anonymous memory mapping is then applied to the address space, so that the overlap of the segments will not generate segmentation faults upon access. After some bookkeeping is performed, the process has slightly over a full megabyte of virtual address space associated with it. Some virtual memory is associated with the process outside of its addressable range which will be used to initially put the process into virtual-8086 mode. Since Linux demand pages virtual memory into physical memory, unused memory in the virtual address space does not involve any extra overhead.

Now that the address space has been set, the instructions and data stored in the binary must be associated with the appropriate part of the address space. Once this is done, the relocation entries must be patched to update the segment references located at the positions pointed to by the entries so that the references are relative to the true starting segment of the program entry point. The extra code needed to perform the switch to virtual-8086 mode is also written into the memory of the process. This code consists of the machine instructions to perform the vm86 system call and to perform an exit system call so the process will terminate cleanly if it ever returns to protected mode for some reason. The structure representing the initial state of the registers upon entry into virtual-8086 mode is also initialized and written into memory at this point.

The next step involves setting the appropriate binary format for the process and setting up the command line argument pages in memory. Once this is done, the command tail can be copied to the appropriate portion of the program segment prefix in the virtual address space. Finally, the registers for the process are set up so that the first instructions that will be executed involve making the vm86 system call. The DOS process is now completely loaded into memory and is ready to start its execution.

4.5 Stock Kernel Changes

As mentioned previously, the required changes to the core kernel were minimal. A new enumeration was added for the DOS personality in personality.h. The other change involved adding an if-else statement to the do_int function in vm86.c. This allows the system call function in the DOS execution domain to be called when a process with the DOS personality triggers an interrupt in virtual-8086 mode.

4.6 DOS Services

Once external interrupt 32 has been invoked and reflected to the appropriate handler, the requested DOS service must still be performed. The following DOS services listed below are currently implemented in the module. The notation representing an address with a segment and an offset (Segment: Offset) is used.

Service	Description
Print String to Screen	ah contains 9, address of dollar terminated string in ds:dx
Terminate Program	ah contains 76, al contains return value
Close File	ah contains 62, bx contains a valid file handle
Open File for Reading	ah contains 61, al contains 0, address of null terminated filename in ds:dx, ax contains file handle on success
Read from File	ah contains 63, bx contains the file handle, cx contains the number of bytes to read, ds:dx contains the address to store the data into, ax returns the number of bytes actually read on success

Implementing the services involves translating the input parameters to versions expected by the true system calls and translating the output of the call back into a form expected by the DOS program if necessary. The system calls used are write for printing to stdout, exit for program termination, and open, read, and close for file I/O.

Printing a string to the screen involves generating the virtual address and determining the length of the string before writing to stdout. If the string is not terminated properly a segmentation fault may be generated if a dollar sign is not found before extending past the virtual address space of the process. Terminating the program only involves using the value in al as the parameter to the call.

Closing, opening and reading a file all involve the possibility of an error occurring. When an error is encountered in all cases, the carry flag will be set. Conversely, on success the carry flag will be clear. On a file close or read error the ax register is set to six, which means an invalid file handle was in bx. On a file open error the ax register is set to two, which means the file was not found. These are not the only possible error codes, but are the only ones returned by the module.

The file handle is passed directly from bx to close a file. Opening a file to read from is only slightly more complicated. It involves generating the virtual address for the filename (which must currently be in a format compatible with the Unix file format since it is not parsed), and also passing the appropriate flag and access mode as parameters. The file descriptor returned is then stored in bx. Finally, reading from a file involves generating the virtual address that will hold the data and then passing it and the other parameters directly to the call. The number of bytes actually read that is returned from the call is then stored in ax.

5 Results

The performance of the module relative to both Windows® and Wine was investigated. A cat-like DOS program was written that takes a filename from the command line and prints it to the screen. Four input file sizes were used of roughly 1, 4, 8, and 16 kilobytes in size. In order to minimize accounting differences between systems and keep the results comparable across systems, the RDTSC instruction was used to read the time stamp counter of the processor immediately before and immediately after the "system" system call was used to execute the DOS program by each system under test. The difference in the time stamp counter and the processor speed was then used to calculate the execution time of the DOS program. The execution times were averaged over 1,000 runs to minimize potential noise present during an individual run. Also, the time taken to execute a system call containing an invalid executable name was also averaged over 1,000 runs and removed from the results. This was used to minimize the impact of the system call overhead and the startup time required for the Wine program. All three systems were measured in their respective windowing environments, and then the performance of both the module and Windows® was measured in full-screen console text mode. Wine was unable to be used in the second group of tests due to its inability to operate in console mode. All tests were performed on the same physical machine, a 1GHz Pentium 3 laptop with a 32MB video card. Windows 2000® and Red Hat® 9 were used for the respective operating systems. The Wine version used was version wine-20031118-1rh9winehq.i386.rpm. Due to the similarity of the results for the various file sizes, only the measurements for the 8KB file will be presented.

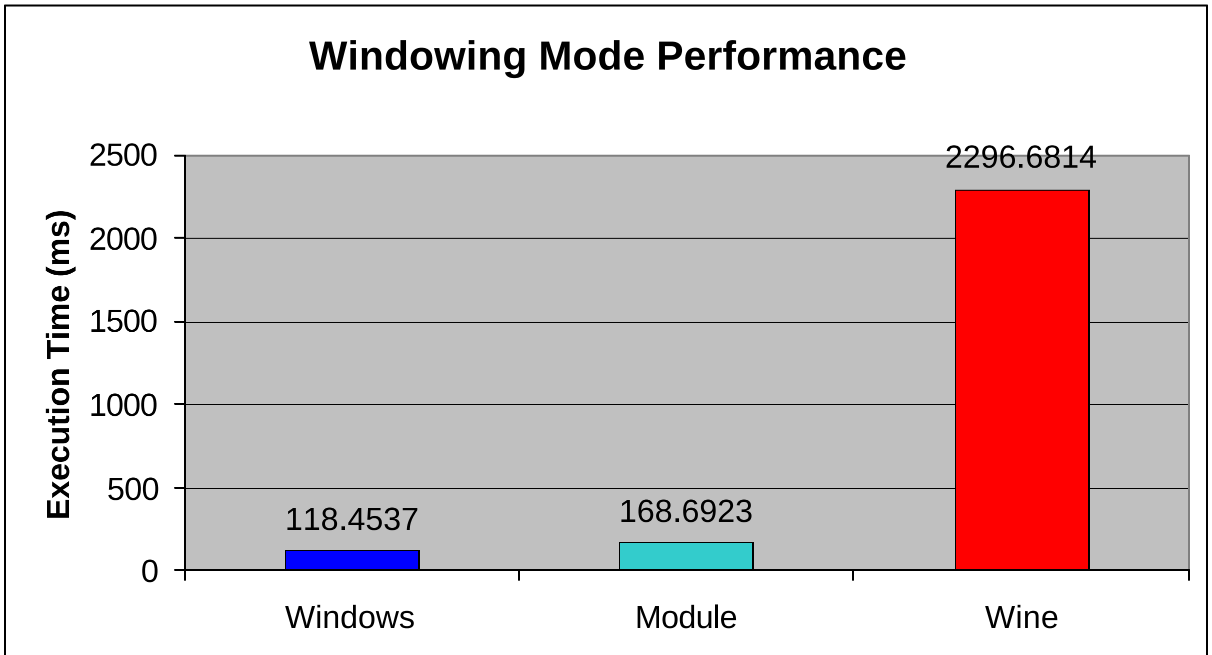

Figure 1

As can be seen from the above graph, Windows® provided the best performance of the three systems, running the cat program in roughly 2/3 the time it took the module to run the same program. Not unexpectedly, Wine came in a distant last, exhibiting very poor performance for running the program compared to the other two methods. This is even after the attempt was made to remove the initialization time for Wine from the results. The sluggish performance is most likely due to the extra context switch overhead required by implementing the functionality needed by the cat program as a regular user process. During the testing, it was noted almost all of the execution time of the module was spent writing to the screen. Speculating that a less efficient windowing system was hampering the performance of the module, the tests were run again in a full-screen console text mode. Wine could not be included in the next tests since a requirement for running it is the X Windowing system.

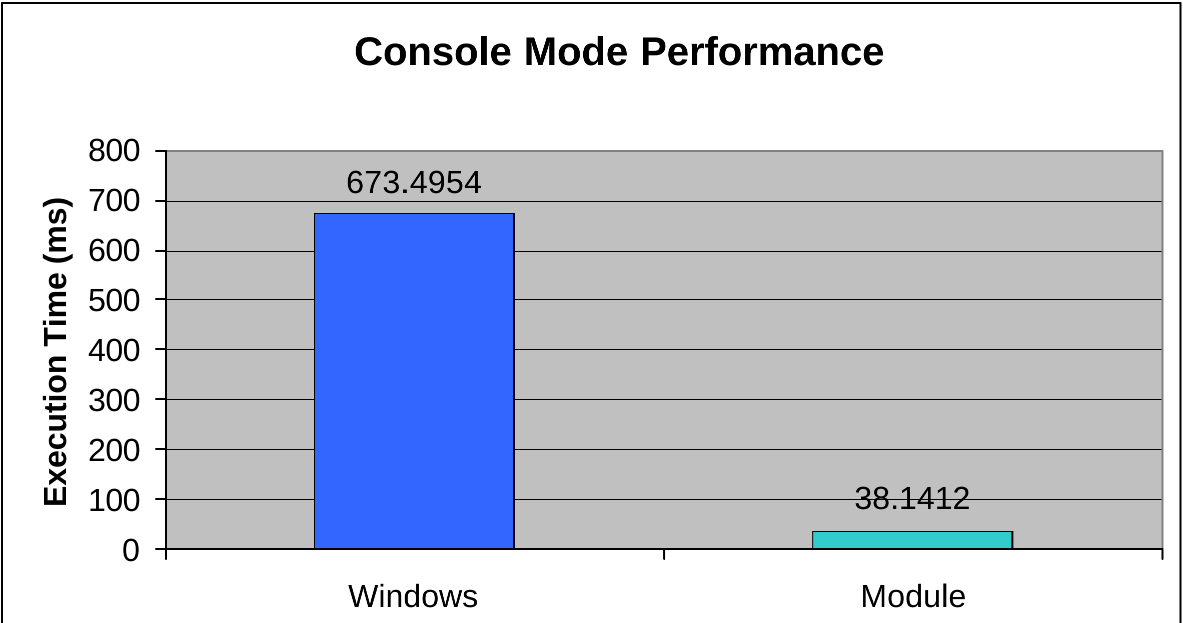

Figure 2

The results turned out to be very surprising. While the performance of the module was very good, turning in the best results of the tests, the Windows® performance lagged significantly behind their regular windowing system numbers. This was on a test that was not supposed to even stress the graphics capabilities of the system. An attempt for a fair test turned into an extremely one-sided contest. The full-screen console text only mode for Windows® appears to be terribly inefficient. It is possible that Windows® does not put the graphics card into a true text-only mode, but rather mimics the mode on top of their regular windowing system. Such a method could impact the performance of screen updates, and perhaps this is the reason for the lackluster performance numbers measured.

6 Conclusions

My work investigated the steps necessary to run applications written for one operating system on a newer, dissimilar host operating system. In an attempt to achieve the best performance possible, the needed functionality was incorporated directly into the kernel. Due to the thoughtful design used to support such ideas, my method achieved very good results. The impact to the stock kernel was minimized while very competitive performance numbers were achieved. This method appears to be a very viable one that can be used to increase the performance of non-native applications. During the course of this work, a great many things were learned about how a modern kernel performs its tasks and about the machine architecture it runs on. Such practical, hands-on experience will undoubtedly be useful in the future.

References

[1] DOSEMU – http://www.dosemu.org

[2] Linux man pages – man vm86

[3] Wine – http://www.winehq.com

[4] Linux Kernel 2.4 Internals – http://www.tldp.org/LDP/lki/index.html

[5] Understanding the Linux Kernel, 2nd Edition – Daniel P. Bovet & Marco Cesati, Published by O'Reilly

[6] Kernel Projects for Linux – Gary Nutt, Published by Addison Wesley

[7] Kernel Source Online – http://lxr.linux.no/source/

[8] Linux VM86 – http://nexus.cs.usfca.edu/~cruse/cs686/vm86blue.cpp

[9] IA-32 Intel® Architecture Software Developer's Manual, Volume 1: Basic Architecture – http://www.intel.com

[10] IA-32 Intel® Architecture Software Developer's Manual, Volume 2: Instruction Set Reference – http://www.intel.com

[11] IA-32 Intel® Architecture Software Developer's Manual, Volume 3: System Programming Guide – http://www.intel.com

[12] Assembly Language for Intel-Based Computers, Third Edition – Kip R. Irvine, Published by Prentice Hall

[13] DOS EXE File Format – http://www.delorie.com/djgpp/doc/exe/

[14] DOS EXE File Format – http://lf.8k.com/MZEXE.HTM

[15] MZ EXE File Format – http://www.itee.uq.edu.au/~cristina/students/david/honoursThesis96/appendix.htm