PROJECT REPORT

Virtualizing a 32 bit Kernel using a purely Hardware-assisted Virtual Machine Monitor

Submitted in partial fulfilment of
the requirements for the award of the degree of

Bachelor of Technology
in
Computer Science and Engineering

Submitted by

Dileep P G
CS04B023
Krishanu Sikdar
CS04B033
Shahzad Abdul Samad
CS04B020
Sreejith B S
CS04B038
Vinaya Surya
CS04B068
Under the guidance of
Mr. Saidalavi Kalady

Department of Computer Science & Engineering
National Institute of Technology Calicut
Kerala - 673601

2008

National Institute of Technology Calicut
Department of Computer Science & Engineering

Certified that this Project Report entitled

Virtualizing a 32 bit Kernel using a purely Hardware-assisted Virtual Machine Monitor

is a bonafide record of the work carried out by

Dileep P G
CS04B023
Krishanu Sikdar
CS04B033
Shahzad Abdul Samad
CS04B020
Sreejith B S
CS04B038
Vinaya Surya
CS04B068

in partial fulfilment of
the requirements for the award of the degree of
Bachelor of Technology
in
Computer Science and Engineering

under our guidance

Mr. Saidalavi Kalady
Lecturer
Dept.of Computer Science & Engineering

Dr.M.P.Sebastian
Professor and Head
Dept.of Computer Science & Engineering

Abstract

Virtualization is a technique for efficient utilization of hardware resources. A Virtual Machine Monitor(VMM) is a piece of software that implements virtualization. It enables concurrent execution of multiple operating systems on a single machine. The initial phase of the project involved a comprehensive study of the recent x86 virtualization extensions. We followed this up with the design and implementation of a simple kernel with basic functionalities. Finally, we worked on extending a purely Hardware-assisted VMM to boot this kernel within its Virtual Machine.

Chapter 1

Introduction

Virtualization is a solution for many enterprise problems such as under utilization of servers, security threats, high maintenance costs and high power consumption. A Virtual Machine Monitor(VMM) essentially controls the concurrent execution of one or more operating systems on a single physical machine. It is generally a thin software layer that exports Virtual Machine(VM) abstractions. Each abstraction looks enough like bare hardware that any software written for the hardware will run on the virtual machine without change. The VMM has the task of providing each Operating System the appearance of full control over a complete computer system (memory, CPU, and all peripheral devices). The nature of VM and VMM abstractions allow various functionalities to be added to the system. For example, VMM can allow users to isolate untrusted applications by confining its operation to a particular VM. Since its operation is confined to that environment the threat is only to that particular VM.

There exist different techniques for virtualization such as Full virtualization, Para Virtualization and the more recent hardware-assisted virtualization. As part of the project, we present a study of the recent architectural support for x86 virtualization. This is followed by the design and implementation of a simple kernel from scratch as well as the extension of a purely hardware-assisted prototype VMM to run the kernel atop.

1.1 Motivation

Until the release of recent extensions from major hardware vendors, x86 the world's most popular architecture remained quite hostile towards virtualization. In its native form, the x86 architecture doesn't meet the Popek & Goldberg's formal requirements for Virtualization[5]. In a virtualized environment, those instructions that require control to be handed over to the VMM are called sensitive instructions. Privileged instructions within a VM can be trapped to handover control to the VMM. If all the sensitive instructions are privileged then the processor is said to be virtualizable. There are 17 instructions in x86 which violate this basic requirement.

Moreover, the x86 architecture has privilege levels called Rings. The operating system normally runs at ring level 0. So to control a guest OS within a Virtual Machine, the VMM must run at a higher privilege level than the OS. So it is only possible to run at the highest privilege level 0. Even in that case the Operating System must execute at level 1 or higher. But most proprietary operating systems are designed to run at level 0 or they will fail to operate. One solution to this problem can be modifying the kernel of guest OS. But for most commercial operating systems it will violate their Licensing terms and conditions. So in such cases implementing a virtualized environment requires expensive changes in operating systems and it might cause software incompatibility issues.

Different software techniques have been in use for x86 vitualization. These include Full Virtualization which completely emulates the underlying hardware and Paravirtualization which necessitates modifications to the Operating System kernel. Both these techniques involve complicated software workarounds to overcome hardware limitations and come with an associated overhead. The recent x86 hardware virtualization extensions from Intel and AMD are the answer to many of the aforesaid problems. Hardware Virtualization, though at a nascent stage at present, represents the future of virtualization. This is the inspiration behind this project. It involved an in-depth study of the recent x86 virtualization extensions. We followed it up with hands-on work on a purely hardware-assisted virtual machine monitor utilizing the aforementioned hardware support.

1.2 Problem Definition

The objective of this project is to complete the following

1. A study of the recent x86 architectural extensions in support of virtualization.

2. Development of a 32 bit kernel from scratch with the following features.
   - Multiboot compatibility
   - Basic Segmentation setup
   - Paging setup
   - naïve memory allocation
   - Interrupt handling
   - Timer, Keyboard and Monitor support

3. Extending a 64-bit Virtual Machine Monitor to run the 32-bit kernel atop. We have worked on the following features of the VMM.

- Initializing AMD SVM
- creating a VM
- booting a guest OS image within the VM

1.3 Literature Survey

Formal requirements for 3rd generation virtualizable architectures are stated in [5].It proposes a model to check whether a given architecture is virtualizable or not. Ability of x86 architecture to support a Secure Virtual Machine Monitor is analyzed in [6]. As per their analysis x86 architecture fails in providing security and cannot be used for a fully secure Virtual Machine Implementation.[7] discusses about current technology and future trends in the implementation of Virtual Machine Monitor. We decided to use Hardware Virtualization for our VMM implementation.

The initial phase involved the study of AMD SVM architecture. The AMD software developer manuals give a good idea of the new hardware virtualization support from AMD. [2] gives a detailed account of the AMD SVM extensions from a VMM developer's point of view.

The next phase dealt with kernel development. [10] describes about the internals of the kernel. We also gained inputs to extend it to a VMM. [11] discusses about advanced Operating Systems topics. [12] states various fine tuning methods in OS implementation which improves features and increases development pace. We used various forums in [11] to get insights from the experts.

The last phase was on Virtual Machine Monitor development which needed a thorough understanding of AMD SVM system programming. [3] describes the system programming features including the VMM specific ones of AMD SVM architecture. [4] describes the general purpose registers and data structures of AMD SVM architecture in detail.[3] and [4] were extensively referred during our hands-on work on the Hardware assisted Virtual Machine Monitor.

Chapter 2

System Study

As the initial part of our work, we present an overview of the hardware support for Virtualization. We focus on the AMD SVM extensions.

2.1 Hardware Support for Virtualization

Until recently, the VMMs had to use complicated software techniques to overcome the inherent architectural limitations to implement virtualization. Now with the advent of processor level support for virtualization from major x86 hardware vendors, the VMM and multiple guest Operating Systems atop have the right environment they need to run without any emulation or OS modification. Hardware virtualization support also vastly simplifies the design of a VMM. The transfer of control between VMM and guest OS is now done largely on hardware which does away with most of the software overhead. Also, key state information for the CPU and guest OS can now be stored in protected memory that only the VMM has access to.

The respective technologies from Intel and AMD in this domain are Intel VT codenamed Vanderpool[7] and AMD SVM codenamed Pacifica[2]. The two technologies are similar but incompatible in the sense that a VMM designed to work on one cannot automatically run on the other. Our study focusses on AMD SVM.

2.2 AMD Secure Virtual Machine architecture

The AMD SVM architecture provides a set of hardware extensions designed to enable economical and efficient implementation of virtual machine systems. The term host refers to the execution context of the VMM, and guest, that of an OS running atop. World switch refers to the operation of switching between the host and guest. The AMD virtual machine architecture is designed to provide Mechanisms for fast world switch between guest and host. Prominent features include the ability to intercept selected instructions or events in the guest, external access protection for memory, assists for interrupt handling and virtual interrupt support, a guest/host tagged TLB, Nested Paging to reduce virtualization overhead. These are described in detail in the following section.

Instruction Set Additions

AMD SVM introduces several new instructions and modifies several existing instructions to facilitate the implementation of VMM systems on the x86 architecture or more specifically the AMD64 architecture. The following are the virtualization specific additions to the instruction set.

1. VMRUN - Start execution of a guest
2. VMLOAD - Save subsets of processor state
3. VMSAVE - restore subsets of processor state
4. VMMCALL - Allow guests to explicitly communicate with the VMM
5. STGI - set the global interrupt flag
6. CLGI - clear the global interrupt flag
7. SKINIT - Secure init and control transfer with attestation
8. INVLPGA - Invalidate TLB entries in a specified ASID

Guest Mode

This new processor mode is entered through the VMRUN instruction. When in guest mode, the behavior of some x86 instructions changes to facilitate virtualization.

Virtual Machine Control Block (VMCB)

There is a VMCB for each running guest OS. The VMCB is divided into two areas.

1. **Control Area:** contains various control bits including the intercept vector with settings that determine what actions cause #VMEXIT (transfer of control from the guest to host). Rich set of intercepts allow the host to customize each guest's privileges.
2. **State Area:** All CPU state for each guest is saved in this area. Information about the intercepted event is put into the VMCB on #VMEXIT

VMRUN

- Host state is saved to memory
- Guest state loaded from VMCB
- Guest runs

#VMEXIT

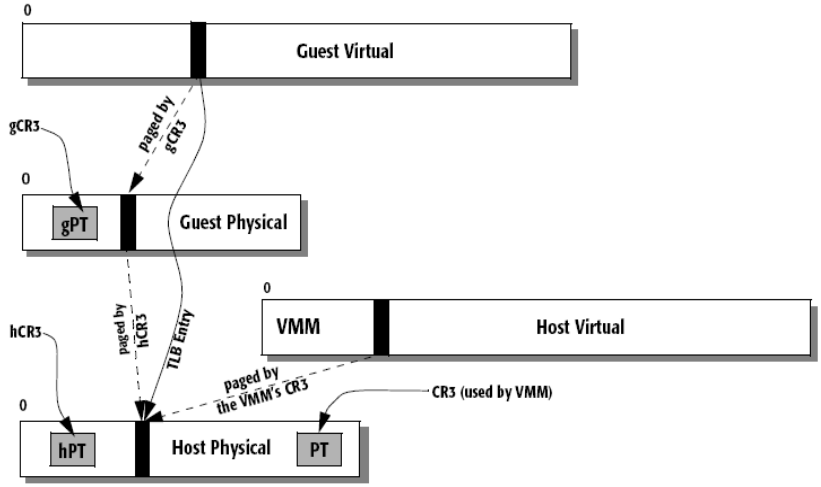
- Guest state is saved back to VMCB
- Host state loaded from memory

- Host State Save Area is pointed to by Model Specific Register(MSR) VM_HSAVE_PA
- VMCB is pointed to by register RAX

Nested Paging

The SVM Nested Paging facility provides for two levels of address translation in hardware, thus eliminating the need for the VMM to maintain the so called shadow page tables in software.

With nested paging enabled, the processor applies two levels of address translation. A guest page table (gPT) mapping guest virtual addresses to guest physical addresses located in guest physical space. Each guest also has a host page table (hPT) mapping host virtual addresses to host physical addresses located in host physical space. Both host and guest levels have their own copy of the CR3 register, referred to as hCR3 and gCR3, respectively.



After translating a guest virtual address using the guest page tables, the resulting (guest physical) address is treated as a host virtual address and is further translated, using the host page tables, into a host physical address. The resulting translation from guest virtual to host physical address is cached in the TLB and used on subsequent guest accesses.

Nested paging is enabled by the VMRUN instruction if the NP_ENA bit in the VMCB is set to 1; nested paging is disabled by #VMEXIT.

Thus there are 3 different registers – hCR3, gCR3 and CR3. The value of hCR3

can be different from the CR3 in effect while the VMM is running; this gives the VMM maximum flexibility on how to further remap guests' physical address spaces, and where to optionally map guest physical pages in the VMM's address space. The optional host paging mechanism allows a VMM to page out guest pages and to use copy-on-write techniques (i.e. sharing of redundant physical pages) between guests. We are not planning to implement host paging in our HVMM at this point.

Tagged TLB

In the SVM usage model, the VMM is mapped in a different address space than the guests each of which in turn have their own address spaces. To reduce the cost of world switches, the TLB is tagged with an address space identifier (ASID) distinguishing host-space entries and different guest-space entries from each other. The ASID Tag in the TLB specifies to which virtual machine, each memory page is assigned. This allows more efficient switching between virtual machines as it completely eliminates the need for TLB flushes each time a different virtual machine is scheduled.

Chapter 3

Design

The project involved two distinct phases of design.

1. Design of a simple x86 based OS kernel with the following functionalities.

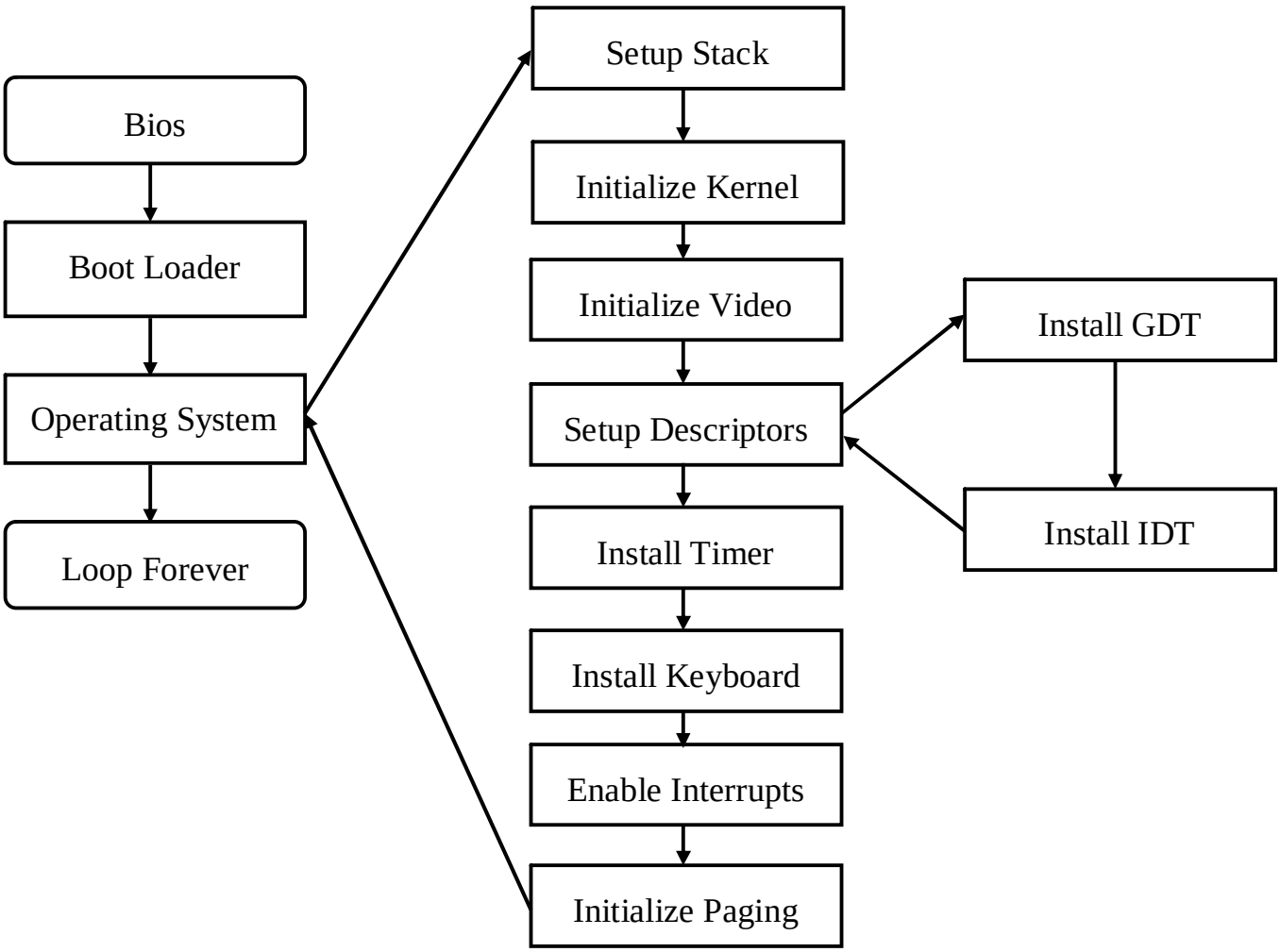
   - Basic Segmentation setup
   - Paging setup
   - naive memory allocation
   - Interrupt handling
   - Support for the following devices
     1. Monitor
     2. Keyboard
     3. Hardware Timer

2. Design of a purely Hardware-assisted Virtual Machine Monitor to run the OS kernel atop with the following added features. Some of the proposed features are still to be implemented.

   - Initializing AMD SVM
   - creating VMs
   - booting a guest OS image within each VM
   - scheduling the running OSs
   - User interface to start, stop and switch screen control to a VM.

3.1 OS Design

OS Control Flow



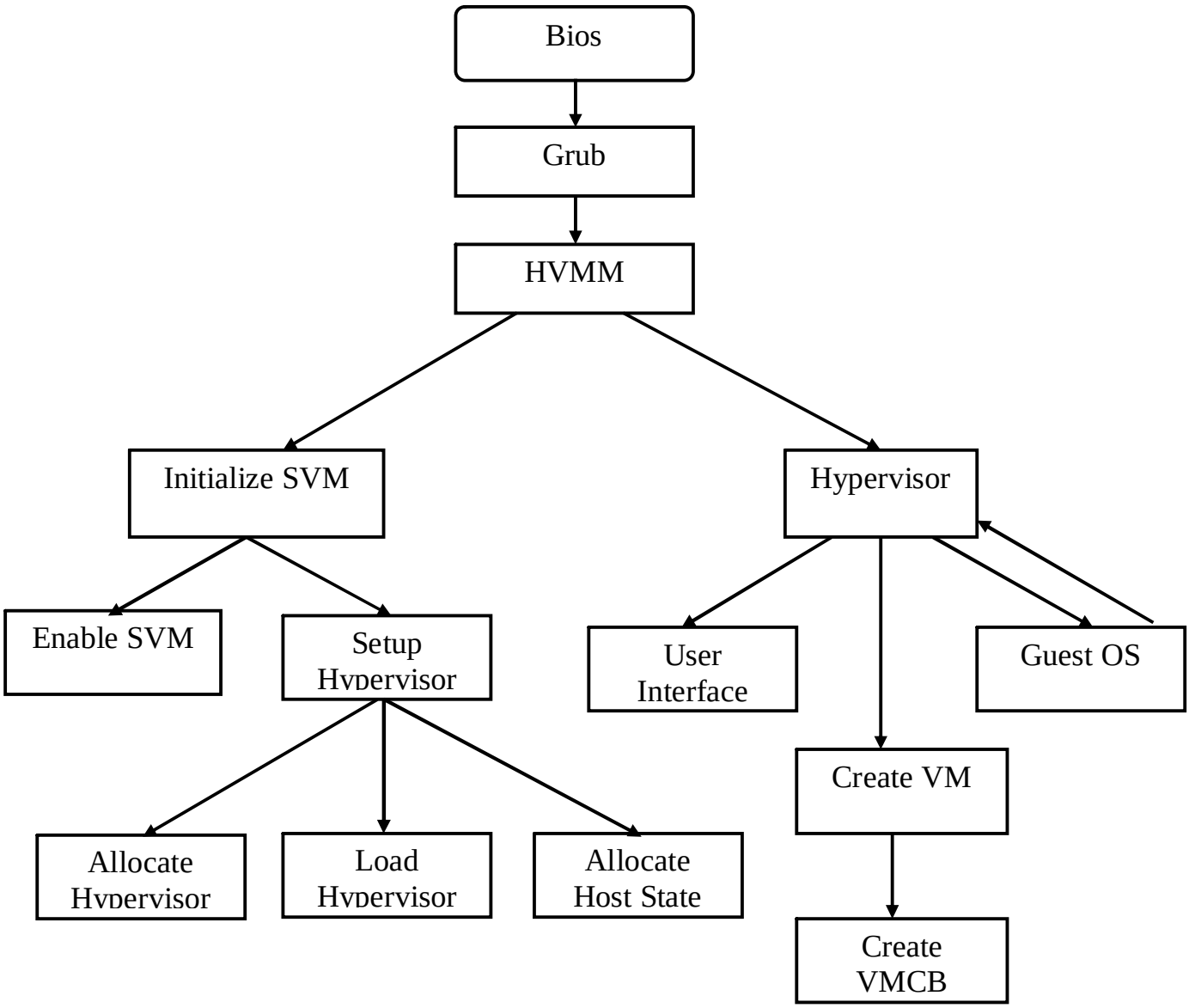
OS Pseudo Code

```
OS ( ) {
     SETUP_STACK ( );
     KERNEL_MAIN ( );
}

KERNEL_MAIN ( ) {
     INIT_VIDEO ( );
     SETUP_DESCRIPTOR_TABLES ( );
     INSTALL_TIMER ( );
     INSTALL_KEYBOARD ( );
     ENABLE_INTERRUPTS ( );
     INITIALIZE_PAGING ( );
     DEMO ( );
}

INIT_VIDEO ( ) {
     CLEAR_SCREEN ( );
}

SETUP_DESCRIPTOR_TABLES (  ) {
     INSTALL_GDT ( );
     INSTALL_IDT ( );
}

INSTALL_GDT (  ) {
     //Initialize the GDT limit and base
     //Setup three segment descriptors null, code and data
     //Load the GDTR register with a pointer to our GDT
}

INSTALL_IDT (  ) {
     // Initialize the IDT limit and base
     SETUP_PIC ( );
     // Set Interrupt Gates to handle standard exceptions and
hardware interrupts
}

INSTALL_TIMER ( ) {
     INSTALL_INTERRUPT_HANDLER (TIMER_HANDLER);
}

INSTALL_KEYBOARD ( ) {
     INSTALL_INTERRUPT_HANDLER (KEYBOARD _HANDLER);
}

INSTALL_INTERRUPT_HANDLER (HANDLER_ROUTINE) {
     // Map a particular IRQ to its Handler routine
}
```

```
INITIALIZE_PAGING ( ) {
     INITIALIZE_FRAME_BITMAP (  );
     SETUP_KERNEL_PAGE_DIRECTORY (  );
     INSTALL_INTERRUPT_HANDLER (PAGEFAULT_HANDLER);
     SWITCH_PAGE_DIRECTORY (KERNEL_DIR);
}

INITIALIZE_FRAME_BITMAP (  ) {
     //Create a sequence of bits to keep track of used and unused
frames in Physical Memory
}

SETUP_KERNEL_PAGE_DIRECTORY (  ) {
     //Create a Kernel Page Directory structure
     KERNEL_IDENTITY_MAP (  );
}

SWITCH_PAGE_DIRECTORY (DIR) {
     //  Load  the  system  paging  registers  CR3  &  CR0  with
appropriate values
}

KERNEL_IDENTITY_MAP (  ) {
     // Allocate Frames (in lower Physical Memory) for each Kernel
page in a way that the virtual and physical addresses remain the
same
}
```

3.2 VMM Design

VMM Control Flow



VMM Pseudo Code

```
High-Level VMM Pseudo-Code {

     INITIALIZE_SVM();
     HYPERVISOR();
}

LOAD_VMM_UI() {
     //display text based interface listing guest OS choices
     //wait for user input
}

HYPERVISOR() {
     while (1) {
          If(vmm_switch) {
               LAUNCH_VMM_UI();
               vmcb = GET_SELECTED_VMCB();
               if(vmcb  = NULL) {
                    vmcb = SETUP_VMCB();
                    ADD_ACTIVE(vmcb);
               }
               LAUNCH_VM_UI();
          }
          else {
               vmcb = GET_NEXT_VMCB();  // from scheduler
          }
          rax = &VMCB;
          VMLOAD(rax);
          While(running_vm) {
               VMRUN(rax);
               Switch(exit_code) {
                    //handle interrupt in each case
                    If(timer_expire  OR  vmm_switch  or  power_off)
               break;
               }
          }
          If(power_off) REMOVE_ACTIVE(rax);
          else VMSAVE(rax);
     }
}

INITIALIZE_SVM() {
     ENABLE_SVM();
     SETUP_HYPERVISOR();
}

ENABLE_SVM() {
     EFER.SVME = 1;
}
```

```
ALLOCATE_VMCB() {
     //allocate a region of physically contiguous, page-aligned,
non-pageable memory
}

SETUP_VMCB() {
     ALLOCATE_VMCB();
     // CLGI instruction is executed to disable global interrupts
     // initialize the control area of the VMCB with a set of
intercept conditions that will cause execution to transfer out of
the guest and back to the hypervisor;
     // initialize the guest area of the VMCB with the address
where guest execution should begin
}

SETUP_HYPERVISOR() {
     ALLOCATE_HYPERVISOR_CODE();
     LOAD_HYPERVISOR_CODE();
     ALLOCATE_HOST_STATE_AREA();
}

ALLOCATE_HYPERVISOR_CODE() {
     //allocate a non-paged area in kernel memory
}

LOAD_HYPERVISOR_CODE() {
     //copy the hypervisor code to memory
}

ALLOCATE_HOST_STATE_AREA() {
     //allocate a non-paged contiguous physical memory space for a
host save area
     //store the physical address to this area in the VM_HSAVE_PA
register
}
```

Virtualized OS Control Flow

BIOS -> BOOTLOADER -> VMM -> VM

The OS runs from within the VM. Once it gains control, it continues until certain specific predefined operations cause transfer of control back to the VMM.

Chapter 4

Implementation

In the course of the project, we had to go through two distinct phases of implementation as well corresponding to the two design phases.

1. development of a 32 bit OS with basic functionalities as described in the previous chapter
2. Extension of a 64 bit AMD-V based HVMM to boot our OS within a virtual machine

4.1 OS Implementation Details

Development Environment

The major portion of the OS kernel is written in 'C' language. The early startup code needs to be in assembly and is done using the NASM (Netwide Assembler) assembly language.

OS	:	32-bit Windows XP
'C' Compiler	:	DJGPP
Assembler	:	nasm
Linker	:	GNU ld
Image Creator	:	WinImage
Emulator	:	Bochs
BootLoader	:	GRUB

The object files produced by nasm and DJGPP are linked together to form a single binary file 'kernel.bin' by the linker 'ld' using a custom linker script. A bootable floppy disk image containing GRUB as well as kernel.bin is created using WinImage and the kernel boots within the bochs emulator.

Startup Assembly Code

The initial part of the assembly code deals with making the os Multiboot compliant. Multiboot is a standard to which GRUB expects a kernel to comply. It is a way for the bootloader to

1. Know exactly what environment the kernel wants/needs when it boots.

2. Allow the kernel to query the environment it is in.

To make your kernel multiboot compatible, there needs to be a multiboot header structure somewhere in the first 4 KB of the kernel. The Multiboot header structure is defined in nasm as follows.

```
dd   MBOOT_HEADER_MAGIC
dd   MBOOT_HEADER_FLAGS
dd   MBOOT_CHECKSUM
dd   mboot
dd   code
dd   bss
dd   end
dd   start
```

The header flags are defined as follows.

MBOOT_HEADER_MAGIC

A magic number. This identifies the kernel as multiboot-compatible.

MBOOT_HEADER_FLAGS

A field of flags. We ask for GRUB to page-align all kernel sections (MBOOT_PAGE_ALIGN) and also to give us some memory information (MBOOT_MEM_INFO).

MBOOT_CHECKSUM

This field is defined such that when the magic number, the flags and this are added together, the total must be zero. It is for error checking.

mboot

The address of the structure that we are currently writing. GRUB uses this to tell if we are expecting to be relocated.

code, bss, end, start

These symbols are all defined by the linker. We use them to tell GRUB where the different sections of our kernel can be located.

On bootup, GRUB will load a pointer to another information structure into the EBX register. This can be used to query the environment GRUB set up for us.

Immediately on bootup, the asm snippet tells the CPU to push the contents of EBX containing a pointer to the multiboot information structure onto the stack, disable interrupts (CLI), pass control to 'C' land where the main kernel code resides.

Interfacing 'C' and Assembly code

The function call conventions used by DJGPP on x86 goes as follows

- All parameters to a function are passed on the stack.
- The parameters are pushed *right-to-left*.
- The return value of a function is returned in EAX.

The kernel Kickoff in 'C'

```
int kickoff(struct multiboot *mboot_ptr) {
     initVideo();
     installDescriptorTables();
     installTimer();
     installKeyboard();
     enableInterrupts();
     initializePaging();
     loopForEver();
     return 0xDEADBEAF;
}
```

Initializing Video

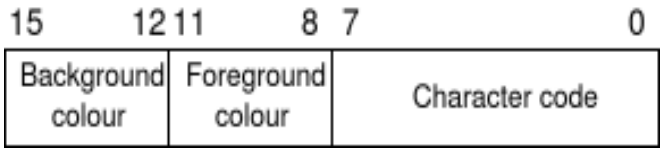
The kernel gets booted by GRUB in text mode. It has available to it a framebuffer, an area of memory that controls a screen of characters (not pixels) 80 wide by 25 high. This will be the mode of operation of our kernel as far as disply is concerned.

The area of memory known as the framebuffer is accessible just like normal RAM, at address *0xB8000*. It is not actually normal RAM. It is part of the VGA controller's dedicated video memory that has been memory-mapped via hardware into our linear address space.

The framebuffer is just an array of 16-bit words, each 16-bit value representing the display of one character. The offset from the start of the framebuffer of the word that specifies a character at position *x*, *y* is :

(y * 80 + x) * 2

In ASCII (Unicode is not supported in text mode), 8 bits are used to represent a character. That gives us 8 more bits which are unused. The VGA hardware uses these to designate foreground and background colors (4 bits each). The splitting of this 16-bit value is shown below.



4 bits for a color code gives us 15 possible colors we can display. The VGA controller also has some ports on the main I/O bus, which we can use to send it specific instructions. It has a control register at 0x3D4 and a data register at 0x3D5 which we use to instruct the controller to update its cursor position.

```
void initVideo();
/* clears the screen and initializes a pointer to the start of the
VGA frame buffer at location 0xB8000 */

void moveCursor();
/* Updates the hardware cursor by setting the high and low cursor
bytes */

void scroll();
/* Scrolls the text on the screen up by one line by moving the
existing text chunk up by one line and overwriting the last line
fully with 80 blank spaces. Finally We position the cursor at the
beginning of the last line */

void cls();
/* clears the screen by overwriting with blank spaces */

void putChar(u8 c);
/* Writes a single character 'c' on to the screen. */

void putNum(u32 num);
/* Writes the integer interpretation of a single machine word
'num' on to the screen. */

void putHex(u32 hex);
/* Writes the Hexadecimal interpretation of a single machine word
'hex' on to the screen */

void putStr(const u8 *str);
/* Writes an ASCII-Z-String pointed to by 'str' on to the screen
by outputting characters one by one until NULL or a '\0' is
encountered */
```

Initializing Descriptor Tables

The GDT and the IDT are arrays of flags and bit values describing the operation of either the segmentation system (GDT), or the Interrupt Vector Table (IDT).

The Global Descriptor Table Setup

The GDT is nothing but a list of segment descriptors each of which describes a memory segment. In the x86, we have 6 segmentation registers. Each holds an offset into the GDT. They are CS (Code Segment), DS (Data Segment), ES (Extra Segment), FS, GS, SS (Stack Segment). The code segment *must* reference a descriptor which is set as a 'code segment'. There is a flag for this in the access byte. The rest should all reference a descriptor which is set as a 'data segment'.

GRUB sets a GDT up. The problem here is that we don't know where that GDT is, or what is in it. So we could accidentally overwrite it resulting in a triple-fault and reset. This explains why we set up our own GDT.

```
struct gdt_entry {
     u16 limit_low;
     u16 base_low;
     u8 base_mid;
     u8 access;
     u8 granularity;
     u8 base_high;
} __attribute__((packed));
/* A GDT entry  in 'C' */

struct gdt_ptr {
     u16 limit;
     u32 base;
} __attribute__((packed));
/* To tell the processor where to find our GDT, we have to give it
the address of this special pointer structure. The base is the
address of the first entry in our GDT, the limit being one less
than the size of the table (the last valid address in the table).
*/

void initializeGDT (int num, u32 base, u32 limit, u8 access, u8
gran)
/*   Sets the different values of a GDT entry structure */

void installGDT();
/*  Sets the base and limit of the GDT Structure and then
initializes the three basic segment descriptors namely the  NULL,
CODE and DATA */
```

The portion of the code that loads the GDTR register with our GDT pointer is written in assembly and uses the nasm lgdt instruction.

The Interrupt Descriptor Table Setup

Interrupts affect the normal flow of program execution. Interrupts can be fired externally, via IRQs, or internally, via the 'int n' instruction. The *Interrupt Descriptor Table* tells the processor where to find handlers for each interrupt. It is very similar to the GDT. It is just an array of entries, each one corresponding to an interrupt number. There are 256 possible interrupt numbers, so 256 Interrupt Vectors must be defined. If an interrupt occurs and there is no entry for it, the processor will panic and reset.

```
struct idt_entry {
     u16 base_low;
     u16 sel;
     u8 zero;
     u8 flags;
     u16 base_high;
} __attribute__((packed));
/* An IDT entry  */

struct idt_ptr {
     u16 limit;
     u32 base;
} __attribute__((packed));
/*  A  struct  describing  a  pointer  to  an  array  of  interrupt
handlers. This is in a format suitable for giving to 'lidt' */

void installIDT();
/*   Sets the value of an IDT entry */

void idtSetGate(u8 num, u32 base, u16 sel, u8 flags);
/*   Sets the value of an IDT entry */
```

As with the GDT, the code to load the IDTR is written in asm using the nasm lidt instruction

Faults and exceptions

The processor sometimes needs to signal your kernel in case something unexpected happens, such as a divide-by-zero, or a page fault. To do this, the first 32 interrupts are used. It is therefore very important that all of these are mapped and non-NULL .Otherwise the CPU will triple-fault and reset.

The initial Interrupt handling routine code is written in assembler. The processor on the occurrence of an interrupt or an exception transfers control to this

code after pushing relevant error codes on to the stack. From here, the control is routed to the appropriate handler routine in 'C' with the error codes as well as the essential processor state passed as an argument.

Interrupt Requests and the PIC setup

Interrupts are a very useful mechanism of communication at the hardware level. When the device is ready it will cause a CPU interrupt, causing our interrupt handler to be run.

All devices that are interrupt-capable have a line connecting them to the PIC (programmable interrupt controller). In all modern PCs, there are 2 PICs, the master and the slave, serving a total of 15 interruptible devices (one line is used to signal the slave PIC). PIC as the name suggests can be programmed to alter the interrupt number it delivers to each IRQ line. This is referred to as *remapping the PIC* and is actually extremely useful. When the computer boots, the default interrupt mappings are:

- IRQ 0..7 - INT 0x8..0xF
- IRQ 8..15 - INT 0x70..0x77

This causes us somewhat of a problem. The master's IRQ mappings (0x8-0xF) conflict with the interrupt numbers used by the CPU to signal exceptions and faults. The normal thing to do is to remap the PICs so that IRQs 0..15 correspond to ISRs 32..47 (31 being the last CPU-used ISR). The communication with PICs is via the I/O bus. Each has a command port and a data port:

- Master - command: 0x20, data: 0x21
- Slave - command: 0xA0, data: 0xA1

Remapping involves a full reinitializing of the PIC.

```
void remapIRQs() {
    outportb(0x20, 0x11);
     ....
     ....
    outportb(0xA1, 0x0);
}
// Remap the irq table.
```

We set IDT gates for handling different Faults and Exceptions. We also set IDT gates for numbers 32-47, for our IRQ handlers numbered 0-15.

```
void setInterruptGates();
/* Map the different Interrupt and Exception handler routine to be
executed by suitably initializing the IDT */
```

Timer Setup

The programmable interval timer is a chip connected to IRQ0. It can interrupt the CPU at a user-defined rate (between 18.2Hz and 1.1931 MHz). The PIT is the primary method used for implementing a system clock.

We setup the PIT to fire interrupts at regular intervals of our choice. This essentially involves programming the PIT to operate at the required frequency and telling our interrupt mechanism to handle IRQ0 with our naive timerHandler() routine in 'C'.

The PIT has 4 registers in I/O space - 0x40-0x42 are the data ports for channels 0-2 respectively, and 0x43 is the command port.

```
void installTimer(u32 interval);
/* Register our timerHandler routine with our Interrupt Mechanism.
Optionally send send to the PIT a value to divide it's input clock
by, to get our required frequency divisor. The divisor must be
small enough to fit into 16-bits. This involves sending a command
byte followed by the frequency divisor */

void timerHandler(struct regs *r)
/* Perform the necessary operations when a Timer interrupt fires
*/
```

Keyboard Setup

The keyboard assigns a number to each key on the keyboard called the scancode. We must use a lookup table (an array of values) and use the scancode as the index into this table. The lookup table is called a keymap, and will be used to translate scancodes into ASCII values.

The keyboard is controlled by a special microcontroller chip on the motherboard . The keyboard controller has two registers the Data register at 0x60 and the Control register at the 0x64 . Anything that the keyboard wants to send the computer is stored into the Data register. The keyboard will raise IRQ1 whenever it has data for us to read.

The keyboard's data register exists at 0x60. When the IRQ happens, we call this

handler which reads from port 0x60. This data that we read is the keyboard's scancode. Following this we take appropriate action say, displaying the corresponding ASCII character on the screen.

```
void installKeyboard()
/* Register our timerHandler routine with our Interrupt Mechanism
*/

void keyboardHandler(struct regs *r)

/* Read from the keyboards data buffer, interpret the scancode and
perform the necessary operations when a Keyboard interrupt fires
*/
```

Paging Setup

Paging is a memory management technique which serves a twofold purpose - memory protection, and virtual memory. Paging works by splitting the virtual address space into blocks called *pages*, which are usually 4KB in size. Pages can then be mapped on to *frames* - equally sized blocks of physical memory.

Paging Data Structures

The CPU needs to know about a *page directory*, which is a 4KB large table, each entry of which points to a *page table*. The page table is, again, 4KB large and each entry is a *page table entry* which gives the physical frame address as well as certain status bits associated with the page. This way, the entire 4GB address space can be represented page by page.

Page structure in 'C'

```
typedef struct page {
    u32 present    : 1;
    u32 rw         : 1;
    u32 user       : 1;
    u32 accessed   : 1;
    u32 dirty      : 1;
    u32 unused     : 7;
    u32 frame      : 20;
} page;

typedef struct page_table {
    page pages[1024];
```

```
} page_table;

typedef struct page_directory {
     page_table *tables[1024];
     u32 table_phys_addrs[1024];
} page_directory;
```

Page Table is just an array of 1024 pages and Page directory holds 1024 pointers to Page Tables. In addition, the physical address of each of its Page Tables. This is necessary when we get our kernel heap allocated and the directory may be in a different location in physical memory.

Naïve Memory Allocator

As we're allocating quite early on in the kernel bootup, we assume that nothing that is allocated will ever need to be freed. We just have a pointer to the start of free memory which we go on incrementing on each allocation

```
u32 kalloc(u32 size, int align, u32 *phys);
/* allocate and return a pointer to a new chunk of memory
satisfying the request */
```

Frame allocation

If we want to map a page to a frame, we need some way of finding a free frame. Of course, we could just maintain a massive array of 1's and 0's, but that would be extremely wasteful - we don't need 32-bits just to hold 2 values, we can do that with 1 bit. So if we use a bitmap, we will be saving a lot of space.

```
void initializeFrameBitmap(u32 mem_size);
/* initialize the bits for the entire size of main memory */

u32 findFreeFrame();
/* find the first free frame. ie. The first set bit in the bitmap
*/

void setFrame(u32 frame_no);
/* set a bit in the frames bitset */

void clearFrame(u32 frame_no);
/* clear a bit in the frames bitset */

void allocPageFrame(page *p, int is_kernel, int is_writable);
/* Function to allocate a frame. */

void freePageFrame(page *p)
```

```
/* Function to deallocate a frame. */
```

Enabling Paging

This involves

1. Copying the location of the page directory into the CR3 register. This must, of course, be the **physical** address.
2. Setting the *PG* bit in the CR0 register. This can be done by OR-ing with 0x80000000.

```
void initializePaging();
/*
create the frames bitset, and set everything to zero. Then allocate space for a page directory. After that, allocate frames such that any page access will map to the frame with the same linear address, called identity-mapping. Finally register an interrupt handler for page faults does this by moving the address of the physical address of that directory into the CR3 register. After that it first gets the contents of CR0, then OR-s the PG bit (0x80000000), then rewrites it. This enables paging and flushes the page-directory cache as well. */
```

Page faults

When a process does something that the memory-management unit cannot handle, a page fault interrupt is thrown. Situations causing this include reading from or writing to an area of memory that is not mapped (page entry/table's 'present' flag is not set), user mode process trying to access a kernel-only page etc.

The page fault interrupt is number 14, and this throws an error code. This error code details are used by our minimal pageFaultHandler routine written in 'C' and already registered with our Interrupt mechanism. The handler is a naïve one which just displays the details of the fault onto the screen.

```
void switchPageDirectory(page_directory *pdp);
```

/ It takes a page directory, and switches to it. It does this by moving the address of the physical address of that directory into the CR3 register. After that it first gets the contents of CR0, then OR-s the PG bit (0x80000000), then rewrites it. This enables paging and flushes the page-directory cache as well. */*

4.2 HVMM Implementation Details

Development Environment

The bulk of the HVMM is written in 'C' language. Here also, the startup code needs to be in assembly and is done using the GNU Assembler.

OS	:	64-bit openSUSE 10.3
'C' Compiler	:	GCC
Assembler	:	GNU AS
Linker	:	GNU ld
Image Tool	:	dd
Emulator	:	AMD simnow 4.4.2
Bootloader	:	GRUB
Other	:	GNU Make utility

Here the object files from AS and GCC are linked together to form a single x86-64 ELF binary file 'hvmm' by 'ld' using a separate custom linker script. A bootable hard disk image containing GRUB as well as 'hvmm' is created using dd and the hvmm boots within the AMD simnow simulator.

AMD Sim now simulator provides a virtual environment which emulates the AMD SVM architecture. A virtual disk is created using the disktool utility. This emulates a real physical hard disk. The virtual disk created is mounted on a 64 bit Linux machine. The binary files of the kernel, VMM along with boot loader files and settings are copied to the disk. Then the disk is loaded in AMD Simnow and the kernel is run from within the VMM.

The VMM is designed to run on an AMD64 machine with long mode and SVM extensions enabled. The initial phase in VMM development involved porting and reusing the basic startup code as well as some of the basic functionalities including Multiboot compliance, naïve memory allocation, as well as video display setup. The next task was adding VMM specific system initialization.

The the next step involves analyzing the Multiboot information structure passed to us by GRUB. The kernel which is going to be virtualized is made available to us as an additional GRUB Module. The multiboot structure includes details on these modules. These have been loaded by GRUB into memory along with the VMM.

```
if ( ! ( mbi->flags & MBI_MEMMAP ) ) {
     fatal_failure ( "Bootloader provided no memory information\n"
);
}
if ( ! has_guest_image ( mbi ) ) {
     fatal_failure ( "No guest operating system is specified.
Check bootloader configuration.\n");
}
/* We make sure that the guest kernel image has been loaded into
memory */
```

Now, We use the already written simple memory allocation routine to reserve the first 1 GB of physical memory and we map our VMM there. Before we do that we have to copy our guest image to somewhere beyond the VMM reserved area to prevent it from being overwritten.

```
copy_guest_image ( const struct multiboot_info *mbi, const struct
e820_map *e820, struct pmem_layout *pml )
/* copy the guest image to higher memory */

naive_allocator_init ( const struct e820_map *e820, struct
pmem_layout *pml )
/* reserve the lower 1 GB of physical memory for the VMM */
```

This is followed by the core AMD SVM enabling code.

```
u32 eax, edx;
rdmsr ( MSR_EFER, eax, edx );
eax |= EFER_SVME;
wrmsr ( MSR_EFER, eax, edx );
/* Before any SVM instruction can be used, EFER.SVME (bit 12 of the EFER MSR
register) must be set to 1.  */
```

Now, we need to allocate an area of memory to save the processor state information when the guest kernel takes control of the CPU.

```
host_save_area = alloc_host_save_area ( );
 .   .   .   .
wrmsr ( MSR_VM_HSAVE_PA, phys_hsa_lo, phys_hsa_hi );
/* phys_hsa_lo, phys_hsa_hi represent the lower and higher order
bits respectively of the physcal address of the host save area. */
```

Once the basic setup is complete, we proceed to the creation of a Virtual Machine to run the guest on.

```
vm_create ( struct vm *vm, unsigned long guest_image_start,
unsigned long guest_image_size, unsigned long vm_pmem_size ) {
     vmcb = alloc_vmcb ( );
     vm->vmcb = vmcb;

     set_control_area ( );
     set_state_save_area ( );
     // setup necessary bits in the VMCB control area as well as
status area
     vmcb->fields.intr_masking = 1;
     // to force hardware interrupts such as a key press to
transfer control to the VMM
     vmcb->rip = load_elf_image ( guest_image_start,
guest_image_size  );
     //load the executable ELF binary of the kernel and point the
RIP field in the VMCB to it
}
```

The final step is booting up the guest kernel within the Virtual Machine.

```
vm_boot( struct vm *vm) {
     while ( 1 ) {
          start_guest( vm );
          handle_vmexit( vm );
```

```
        }
}
```

With this we successfully ran the 32 bit x86 based guest operating system which we implemented as part of the project within the purely hardware-assisted AMD64 based Virtual machine Monitor.

Currently, work is being put into setting up 64 bit interrupt handling mechanism in the VMM. Additional features including a User Interface as well as scheduling of more than one OS require VMM level interrupt handling.

Chapter 5

Conclusion

As part of the project, we have been able to accomplish the following

- From the study phase, we gained a thorough understanding of the x86 as well as AMD64 architecture, particularly focusing on the SVM virtualization extensions.
- With the implementation phase, we developed a simple 32 bit kernel with basic functionalities. Following this we successfully worked on extending a purely Hardware-assisted 64 bit Virtual Machine Monitor to run the kernel from within its Virtual Machine.

Bibliography

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A.War_eld. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM SOSP*, pages 164.177, October 2003.

[2] AMD, AMD64 Virtualization Codenamed "Pacifica" Technology Secure Virtual Machine Architecture Reference Manual, Publication No.33047 Revision 3.01 May 2005.

[3] AMD, AMD64 Architecture Programmer's Manual Volume 2: System Programming, Publication No.24593 Revision 3.13 July 2007.

[4] AMD, "AMD64 Architecture Programmer's Manual Volume 3:General-Purpose and System Instructions, Publication No.24594 Revision 3.13 July 2007.

[5] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. Commun. ACM 17, 7 (1974), 412–421.

[6] ROBIN, J., AND IRVINE, C. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In Proceedings of the 9th USENIX Security Symposium (2000).

[7] Rich Uhlig, Gil Neigerl Dion Rodgers, Amy L Santoni, Fernando C.M. Martins, Andrew V Anderson ,Steven M Bennett, Alain Kagi, Felix H Leung, Larry Smith, Intel Virtualization Technology, Published by the IEEE Computer Society May 2005.

[8] Keith Adams, Ole Agesen A Comparison of Software and Hardware Techniques for x86 Virtualization. In ACM 2006 October.

[9] Mendel Rosenblum Tal Garfinkel, Virtual Machine Monitors: Current Technology and Future Trends, Published by the IEEE Computer Society May 2005

[10] http://osdever.net/ , An introduction to OS development

[11] http://www.osdev.org/, Advance Operating system development

[12] http://visopsys.org/osdev/index.html , Techniques in OS development

[13] http://kvm.qumranet.com/kvmwiki, The Kernel Vertual Machine

[14] http://www.xensource.com The XEN Virtual Machine Monitor

[15] http://www.virtualbox.org/, Virtual Box, an open source virtualization solution