Securing Operating Systems Through Utility Virtual Machines

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

by

Robert Denz

Thayer School of Engineering

Dartmouth College

Hanover, New Hampshire

June 2017

Examining Committee:

Chairman________________________
Stephen Taylor, Ph.D.

Member________________________
George Cybenko, Ph.D.

Member________________________
Eugene Santos, Jr. Ph.D.

Member________________________
Ryan Durante, Ph.D.

____
F. Jon Kull
Dean of Graduate Studies

Abstract

The advent of hypervisors revolutionized the computing industry in terms of *malware prevention and detection*, *secure virtual machine managers*, and *cloud resilience*. However, this has resulted in a disjointed response to handling known threats rather than preventing unknown *zero-day* threats. This thesis introduces a new paradigm to cloud computing – *utility virtual machines* – that directly leverages virtualization hardware for protection and eliminates often accepted roles of the operating system kernel. This represents a break from prevailing practices and serves to establish a hardware root of trust for system operation.

Acknowledgments

I would like to thank my adviser Dr. Stephen Taylor for his valued support while in graduate school. He provided me with the opportunity to delve deep into the fields of virtualization and symmetric multiprocessing. Giving me the time and resources to develop a far deeper understanding of computing systems and an appreciation for the academic process.

As Rome wasn't built in a day or by a single individual, I would like to thank my research group Scott Brookes, Martin Osterloh, Stephen Kuhn, Morgon Kanter, Jason Dahlstrom, and Colin Nichols, who without them, much of the development would still be on going. Through our combined efforts we have built a truly modern system, which has provided a wealth of novel research topics. Those outside of the research group, Jacob Russell, who as a coworker, friend, and roommate has always provided a keen perspective on the work our group does and Karen Thurston who has been invaluable in helping me balance the logistical details of a Ph.D.

Specifically, I would like to thank my parents Michael and Mary Denz who have supported me throughout academia, career, and academia again. You have been role models throughout my life and always provided me the wisdom needed to make the right choice. Also deserving of thanks is my sister, Brittany Denz, an integral part of my family support system.

My fiancée Emily, who has provided all means of support to me over the years, from home cooked meals, to fun family getaways, as well as allowing me to turn our dining room into my makeshift office to finalize the remainder of my thesis. You have gone above and beyond in your support of me of me on this journey.

Lastly, every one who has listened to me spend days complaining about a random bug in my code.

This research is supported by the Defense Advanced Research Projects Agency as part of the MRC program under contract FA8750-11-2-0257. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

Table of Contents

List of Tables:

List of Figures:

List of Acronyms

Address Resolution Protocol	ARP
Address Space Layout Randomization	ASLR
Advanced Configuration and Power Interface	ACPI
Advanced Programmable Interrupt Controller	APIC
Application Processor	AP
Application Programming Interface	API
Basic Input/Output System	BIOS
Basic Virtualization	VT-x
Bootstrap Processor	BSP
Control Register	CR
Cylinders, Heads, and Sectors	CHS
Destination Format Register	DFR
Direct Kernel Structure Manipulation	DKSM
Dynamic Host Configuration Protocol	DHCP
Extended Feature Enable Register	EFER
Extended Page Tables	EPT
GNU Assembly	GAS
Implementation of the Preboot eXecution Environment	iPXE
Input Output Memory Management Unit Virtualization	VT-d
Input/Output Advanced Programmable Interrupt Controller	I/O APIC
Inter-Processor Interrupts	IPI
Interrupt Control Register High	ICRH

Interrupt Control Register Low	ICRL
Interrupt Virtualization	APIC-v
Lightweight IP	LWIP
Logical Destination Format Register	LDR
Master Boot Record	MBR
Memory Management Unit	MMU
Memory Mapped Input/Output	MMIO
Model Specific Register	MSR
Multi-Processor	MP
Multiple APIC Descriptor Table	MADT
Network File Sharing Daemon	NFSD
Network File System	NFS
Network Utility Virtual Machine Helper Daemon	NUVMHD
Network Virtualization	VT-c
No-Execute	NX
Page Address Extension	PAE
Page Directory Table	PDT
Page Map Level 4 Table	PML4T
Page Table	PT
Page-Directory-Pointer Table	PDPT
Port Input/Output	PIO
Programmable Interrupt Timer	PIT
Programmable Interrupt Controller	PIC

Return Oriented Programming	ROP
Remote System Descriptor Pointer	RSDP
Remote System Descriptor Table	RSDT
SIMD Extension	SSE
Startup Inter-Processor Interrupt	SIPI
Symmetric Multiprocessing	SMP
Time Stamp Counter	TSC
Transition Lookaside Buffer	TLB
Trivial File Transfer Protocol	TFTP
Utility Virtual Machine	UVM
Video Graphics Array	VGA
Virtual APIC	VAPIC
Virtual Machine	VM
Virtual Machine Monitor	VMM
Virtual Switch	vSwitch

Chapter 1 – Introduction

Problem: *Adversaries tailor their attacks in the presence of a hypervisor, which alone cannot protect against zero-day vulnerabilities in the guest operating system*

Hypothesis: *The ability to conduct zero-day attacks can be reduced through hardware isolation and a minimized code surface with acceptable performance.*

The approach advocated in this thesis to mitigating kernel-level zero-day attacks is to directly utilize hardware supported guest-virtual isolation in a radical new generation of hypervisor designs. These designs -- termed utility virtual machines (UVM) – improve security by eliminating the conventional kernel and replacing it with a collection of specialized virtual machines, which employ hardware protections to enforce isolation between system components such as device drivers and system daemons. This establishes a root of trust in hardware and prevents the compromise of one component from undermining the system as a whole.

There are four central challenges to this technique: Can existing hardware virtualization mechanisms be used to extend conventional inter-process communication and synchronization mechanisms (such as rendezvous and message passing primitives) to the hypervisor layer? Can virtualization support the fragmentation of an operating system into individual utility virtual machines to support the appearance of a cohesive system? Can the resulting security features be implemented in a manner that has a minor impact to performance? Finally, can methods be devised to schedule a multiplicity of tasks across multiple cores?

1. 1 Background & Motivation

Virtualization is enabled through the addition of a new layer to the software stack known as the hypervisor [1] or Virtual Machine Monitor (VMM) [2]. The hypervisor encapsulates the hardware, allowing it to be used by multiple operating system instances concurrently. This flexibility, coupled with the cost and performance advantages of sharing the underlying hardware, has revolutionized the computing industry: large numbers (i.e. hundreds of thousands) of generic hardware platforms, using multi-core blade technology, are now coupled through high-performance networking to produce a generic computing surface. Any subset of this collection can be combined to operate in tandem for a particular application using a multitude of operating systems.

Conceptually, the hypervisor presents a virtual machine abstraction that restricts malicious code embedded in one operating system instance from affecting a different instance [3], by containing it within one virtual machine using hardware protection techniques This is achieved through type-1 or bare-metal virtualization [4] as seen in Figure 1.

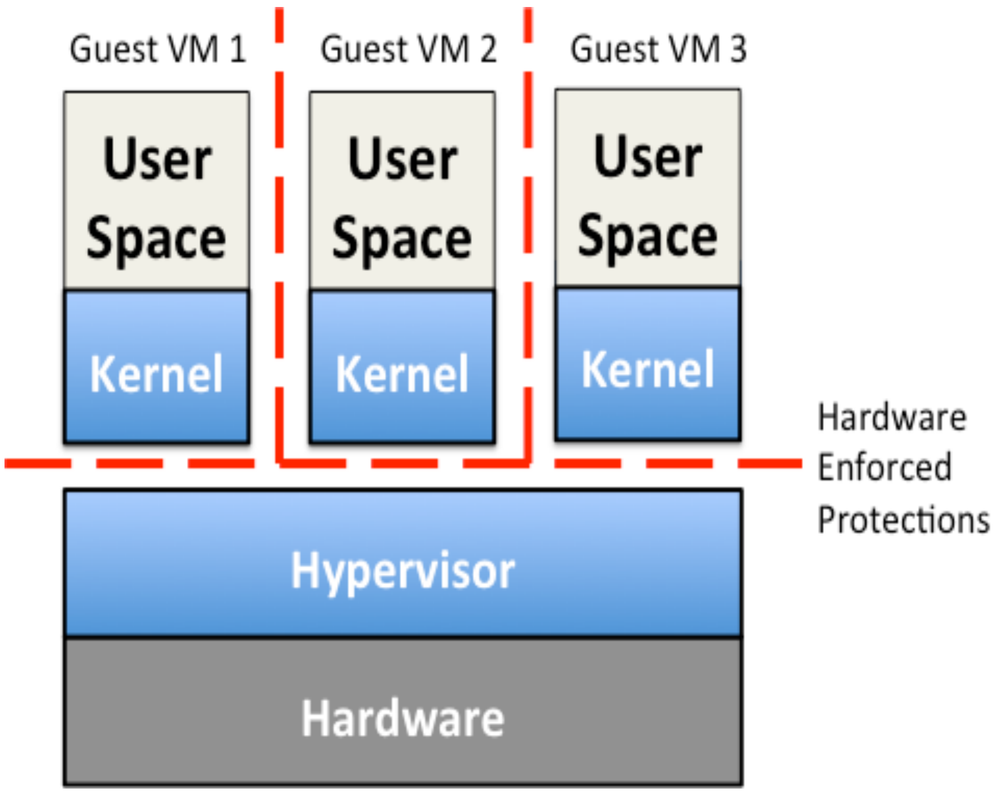


Figure 1 - Bare-Metal Hypervisor

In this configuration the hypervisor controls all of the hardware on the system. On top of the hypervisor sits one or more guest virtual machines, which contains an operating system's kernel and its associated user space. The kernel provides networking, scheduling, and many other key processes. The guest's view of hardware is however tightly controlled through the Intel Virtualization suite of VT-x (basic virtualization), VT-d (input output memory management unit virtualization), VT-c (network virtualization), and APICv (Interrupt Virtualization) [5]. This provides the isolation necessary to protect other guests from a potentially compromised guest, but does not protect data resident inside of that guest.

Unfortunately, hypervisors have introduced their own new security challenges: adversaries now actively attempt to detect the presence of an operating hypervisor in order to tailor attacks accordingly [6]. A wide range of hypervisor detection techniques have already appeared against popular systems such as VMWare, VirtualPC, Bochs, Hydra, Xen, and QEMU [7]. Often, these techniques operate by exploiting timing differences between virtualized and non-virtualized operations [8]. Alternatively, they detect unusual memory locations associated with key operating system data structures [9]. For example, the Red Pill technique works by using the SIDT X-86 instruction to determine the location in memory of the interrupt descriptor table; a machine running above a hypervisor will return a location much higher in memory than one that is not [10]. Following hypervisor detection, the adversary then attacks either the operating system, the virtual switch (vSwitch) sharing network connectivity between virtual machines, or the hypervisor itself [11].

The presence of a hypervisor has no impact on the known and unknown zero-day vulnerabilities associated with a particular operating system. As a result, any exploit that leverages a known vulnerability will operate successfully [12] against any virtual machine running the system. Packaging this exploit within a propagating virus provides the adversary an opportunity to compromise every virtual machine in the cloud running the same instance. It is this *vulnerability amplification* that poses the most significant threat to the future of cloud computing.

After the attacker has gained a foothold and determined they are operating in a virtualized environment, they will attempt to compromise the hypervisor. This often entails chaining together multiple small pieces of kernel code known as gadgets in a Return Oriented Programming (ROP) attack [13,14,15]. This allows the hypervisor to be attacked by the code that was meant to protect it. ROP methods are made easier as *attack surface* increases, which equally raises the number of gadgets present.

ROP attacks can be built to create direct attacks against a vSwitch may undermine the operation of multiple virtual machines on a single host by denying connectivity to all of them simultaneously. The vSwitch provides the same functionality as a physical switch and in consequence exhibits the same vulnerabilities, enabling the same exploits [16]. For example, Address Resolution Protocol (ARP) spoofing, involves the interception of valid network packets by sending fake ARP packets to a switch [17].

Going after the hypervisor itself involves the direct exploitation of vulnerabilities in the hypervisor. All virtual machines executing on a hypervisor have distinct data structures, separated in hardware. This separation forms a semantic gap [18] that prevents virtual machines from having visibility or impact upon each other's data structures [19]. Direct Kernel Structure Manipulation (DKSM) bridges the semantic gap by patching virtual machine data structures and redirecting hypervisor accesses to shadow copies. This allows the virtual machine to present false information to the hypervisor regarding the virtual machine state, which allows implants, such as rootkits [20], to persist without detection.

Virtualization provides inherent redundancy and robust, large-scale, cost-effective availability of shared resources [21]. However, this perception is tempered by the risk of vulnerability amplification and the paucity of knowledge regarding zero-day exploitation: history has shown that lack of detection does not imply lack of infection.

To combat these risks, termed *utility virtual machines* separate and isolate the normal responsibilities associated with a congenital kernel as shown in Figure 2.

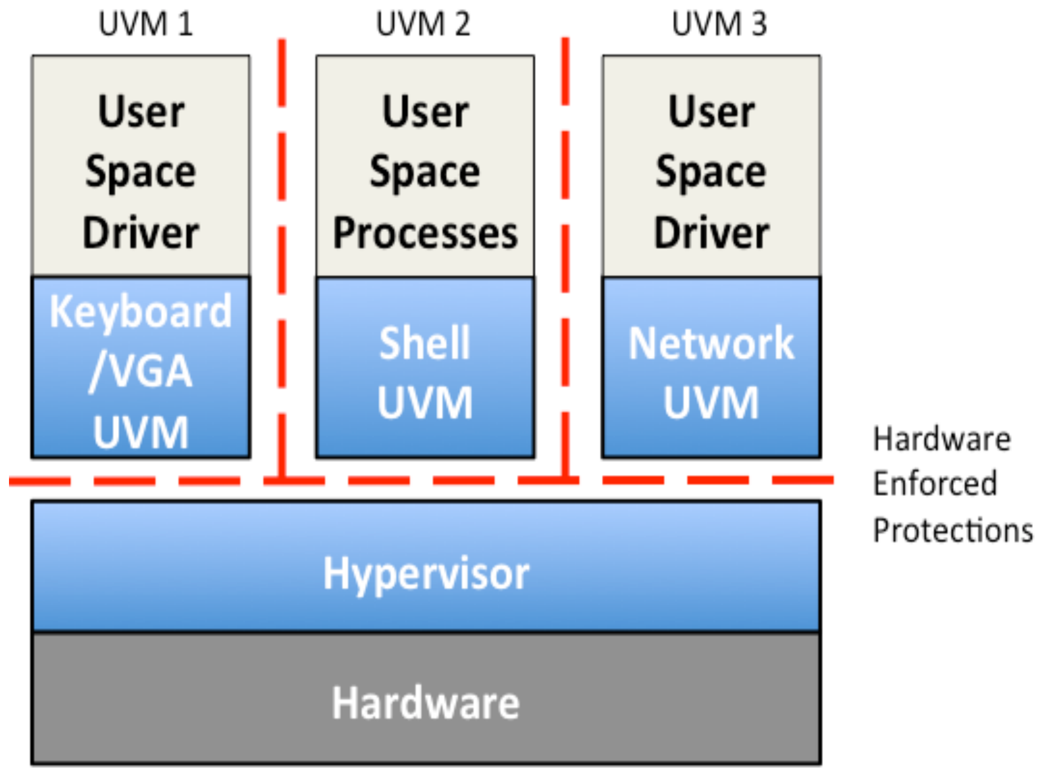


Figure 2 - Bare-Metal Hypervisor with Utility Virtual Machines

Each UVM encapsulates a particular functionality and can include, but is not limited to, user applications, networking and Keyboard/VGA drivers. The benefit of this reorganization is that the standard kernel is eliminated as an entity and hardware isolation is enforced between each component of the system. If any particular UVM is compromised, the attacker has access to only a small subset of the system data taken as a whole. Furthermore, since most data structures and code are unique to the infected UVM, the attacker has little information to glean on other running UVMs. This is in stark contrast to the standard model, where a compromise of the network would give an attacker complete control of the guest and all of its associated data, including many other operating system specific tasks and data structures.

This new approach has only become possible in the last ten years: Moore’s law [22] has provided a rapid growth in the areas of multicore [23] and virtualization [24] technologies. The coupling of these two mechanisms at scale forms the basis for the UVM architecture. Each UVM is assigned a dedicated number of cores for optimal performance of its assigned individual task. Virtualization allows for the operation of non-homogenous VMs making it more difficult for an adversary to cross the semantic gap. Furthermore, the fragmentation afforded by UVMs has the dual benefit of increasing attacker workload and decreasing the attack surface, benefits that will be discussed in detail in chapters 4 & 5.

1.2 Approach

A complete operating system based on the UVM concept has been realized in the latest generation of a research operating system called *Bear* [25] under development at Dartmouth College. This system shares its core motivations of security, modularity, and resilience with MINIX [26], but directly integrates a Symmetric Multiprocessing (SMP) micro-kernel with an associated SMP hypervisor, using Intel x86-64 architectural support including VT-x and VT-d extensions. In previous versions of the Bear system, these technologies were coupled with extensive code sharing between the micro-kernel and hypervisor, which served to reduce the attack surface and space of potential vulnerabilities [27].

Figure 3 shows an overview of the original system that serves to illustrate the interplay of security concepts employed in the design. Like the MINIX micro-kernel, device-drivers are operated from user-space where they can be *refreshed* in a manner similar to the MINIX regeneration process [26]. However, Bear intentionally makes no attempt to detect intrusions or failures; instead, potentially compromised device drivers, when not in use, are non-deterministically refreshed to *deny persistence*, regardless of their infection status. A similar approach is taken to deny persistence in the micro-kernel: the hypervisor non-deterministically refreshes the micro-kernel periodically from a gold-standard, either at pre-arranged or non-deterministic times. Gold-standard images are stored within a *read-only* ramdisk, uploaded using iPXE's signed and encrypted bootstrapping. Each time a component of the system is refreshed it is also *diversified* at load time [28]. This process, an enhanced form of Address Space Layout Randomization (ASLR), ensures that no two running instances of a binary share an exploitable address -- including the hypervisor itself, the micro-kernel, device drivers, system daemons, and user processes. This denies surveillance and reverse engineering while throttling vulnerability amplification caused by using the same micro-kernel throughout a cloud or high-performance computing infrastructure.

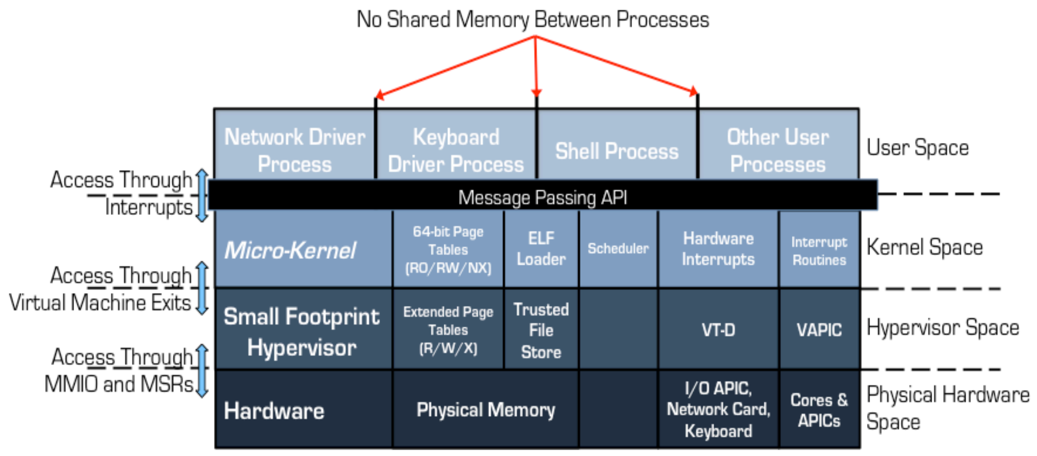


Figure 3 - Bear System Layout

In common with other designs, the original micro-kernel handles multi-core scheduling of user processes and is responsible for protecting the micro-kernel from user processes, and user processes from each other. All processes and layers are hardened by strictly enforcing MULTICS-style read, write, and execute protections [29] using 64-bit x86 address translation hardware. The use of extended page table entries allows micro-kernel to be marked *execute-only*; recall that the normal x86 paging structures do not provide sufficient flexibility [30] to achieve this protection. Finally, although micro-kernel code is replicated in user processes, in common with many other operating system designs, an additional level of indirection allows its location within each process to be obfuscated [31].

Like MINIX, all potentially contaminated user processes, device drivers, and services are executed with user-level privileges and are strictly isolated from the micro-kernel via a message-passing interface. However, unlike MINIX, there is no system task, all system calls, IO, and process scheduling is achieved by interrupt handlers with kernel privileges. These handlers mediate between processes and the kernel to rigidly enforce the interface. Unlike the MINIX rendezvous mechanism [32], Bear uses an asynchronous, bounded buffer interface similar to MPI [33] that provides a single uniform treatment of system calls, inter-process, and off-chip inter-processor communication on blade servers. Similarly, in the rare event the micro-kernel is unable to complete a user process request, a VMExit may be generated, which is handled by the hypervisor in a similar fashion as to the micro-kernel handles interrupts. The hypervisor enforces the protection layer between the micro-kernel and the underlying physical hardware. The hypervisor having full control of the physical hardware accesses functionality as it is needed through memory mapped I/O (MMIO) and Model-Specific Registers (MSR).

The rich collection of modern features in the Bear system provides the transition point to the UVM security model. The most important concept is the rigid enforcement of MULTICS protections [29] through virtualization hardware. These protections ensure that device drivers exist solely in user space as a single process that does not require kernel level privileges. Using this isolation model as a jumping off point, the functionality of these singular device driver processes can be transferred from a *process* to a UVM. However, this can only be accomplished by overcoming the core key research challenges: building a hypervisor message passing system, structural reorganization around utility virtual machines, performance optimization, and scheduling.

Since message passing is central to MINIX and the Bear kernel alike, it will also become central to UVMs as they communicate requests between each other. This communication requires the creation of a message-passing interface to the hypervisor. This is particularly interesting facet of the research, since many members of the research community seek to operate within the hypervisor for the smallest possible time for efficiency. A message-passing interface requires additional compute cycles inside the hypervisor and consequently has the potential to slow the guests operation to some degree. However, since each UVM is not a full-fledged kernel, it was initially unclear how the reorganization would impact performance overall. As we will see, utility virtual machines trade what were once kernel cycles for hypervisor cycles. Unexpectedly, the performance results developed in this thesis show that the separation of duties and direct reliance on modern hardware actually generates a net *improvement* in performance (Chapters 4 & 5).

To demonstrate the concepts, the latest generation of the Bear operating system is realized through a collection of three individual UVMs: a network UVM, a keyboard/VGA UVM, and a shell UVM that handles the responsibilities of a typical shell and is capable of scheduling user processes. Isolation between UVMs is enforced through hypervisor protection hardware and only legitimate communication can occur through the hypervisor messaging system. The hypervisor schedules these three UVMs statically on bootstrapping to a set number of cores present on the system.

A critical aspect of cloud computing and computing in general is the requirement of efficient, reliable, and scalable scheduling of processes. These aspects become more critical as multiple UVM's schedule multiple processes simultaneously. Specifically, one UVM may run a scheduling algorithm to provide fairness, while another could pin processes to specific cores with particular hardware. To meet the criteria of efficiency, reliability, and scalability a diffusive scheduler [34] was explored within the shell UVM. This algorithm had previously been shown to be simple, scalable, and have attractive convergence properties under large scale simulations, but had not been previously been employed in practical systems.

1.3 Performance Metrics

Industry standard methods are used to assess performance in this thesis. The performance of the UVM system has been compared to standard monolithic operating systems with associated hypervisors and the original Bear system, operating on generic Dell workstations. Two benchmark suites were employed: To assess system memory utilization, a test suite developed by Chuck Lever and Chuck Boreham at the University of Michigan measures the performance of *malloc()* in a multithreaded system [35].

In addition, the popular AIM9 benchmark suite is used to measure processor synthetic overhead through its addition, subtraction, and multiplication modules [36]. These tests stress the system by executing instructions that specifically target the internal processor logic.

Beyond system benchmarks, recent studies have confirmed our intuition that the number of vulnerabilities in a code base is proportional to its size [27]. Consequently, attacker workload is estimated in terms of lines of source code loaded in memory. Each utility virtual machine presents a unique sand boxed attack surface, which can be enumerated. The CLOC utility was used to count these lines of code [37].

1.4 Contributions

The contributions of this research are:

- A novel system architecture based on the concept of Utility Virtual Machines in which the operating system kernel is replaced by a collection of virtual machines, each encapsulating a distinct system function. This architecture improves security by sandboxing system functions within virtual machines using hardware protection techniques.

- A body of practical implementation techniques that serve to realize systems based on UVM's.

- A practical demonstration and experimental analysis of UVM architecture that assesses memory utilization, processor performance, and impact on attacker workload based on three exemplars:

    - A Network UVM that comprises all the functionality to encapsulate a network card driver, network file system process, and network stack.
    - A keyboard/VGA UVM that contains all I/O functionality to interact with users [38].
    - A shell UVM that isolates the functionality to fork and schedule user processes on multiple cores [38].

- A generalized rendezvous-style message-passing system between UVMs that operates through the hypervisor, is adapted from those employed to provide system calls [38], and leverages modern APIC-interrupt mechanisms for efficiency

- A practical heat diffusion scheduler to improve performance of process scheduling in the UVM architecture with an associated body of experimental analysis.

- A novel hypervisor shim that inserts itself under a running micro-kernel and leverages virtualization technology to enforce execute-only memory protection [30].

- A practical method for combining iPXE, DHCP, TFTP, and NFS to load binaries across the network and diversify them at load time [28].

- Creation of a micro-kernel that is treated as a dynamically linked library, which enables kernel diversification on a per-process basis [31].

- A body of practical techniques for realizing advanced hypervisor and micro-kernel designs that leverage modern 64-bit multicore and virtualization hardware. (Submitted to software practice and experience).

1.5 Thesis Organization

In outline, the thesis is divided into the following chapters:

Ch 1: *Introduction* states the hypothesis and the associated challenges posed by the thesis, motivates the research, identifies the contributions, and describes the assessment metrics used in analysis.

Ch 2: *Related Work* provides a survey of the relevant literature and demonstrates the opportunity to advance system security through the techniques developed in this thesis.

Ch 3: *Symmetric Multiprocessing* describes the multicore hypervisor and micro-kernel created as part of this thesis. This chapter explores the design choices and specific details of bringing SMP to a system from boot to virtualization.

Ch 4: *Utility Virtual Machines (UVM)* describes the creation of the UVM rendezvous message passing system and its use to couple together the first UVM prototypes through the hypervisor.

Ch 5: *A Further Abstraction: The Network UVM* discusses the challenges and solutions for moving a non-trivial operating system component into a UVM encapsulation.

Ch 6: *Heat Diffusion Scheduling* details the implementation of a novel scheduling technique to improve performance in multicore environments.

Ch 7: *Future Work and Conclusions* makes concluding remarks on the work presented in earlier chapters and describes directions for future work.

Chapter 2 – Related Work

Prior to the start of the UVM work, a survey and analysis of the current security measures implemented with hypervisors to prevent ROP and other attacks was conducted. The viability of an efficient virtualization layer has led to an explosive growth in the cloud computing industry, exemplified by Amazon's Elastic Cloud, Apple's iCloud, and Google's Cloud Platform. However, the growth of any sector in computing often leads to increased security risks. This chapter explores these risks and the evolution of mitigation techniques in open source cloud computing. Unlike uniprocessor security, the use of a large number of nearly identical processors acts as a *vulnerability amplifier*: a single vulnerability being replicated thousands of times throughout the computing infrastructure. Currently, the community is employing a diverse set of techniques in response to the perceived risk. These include *malware prevention and detection*, *secure virtual machine managers*, and *cloud resilience*. These three categories and their roles in preventing an attacker from gaining access to the cloud are illustrated in Figure 4.

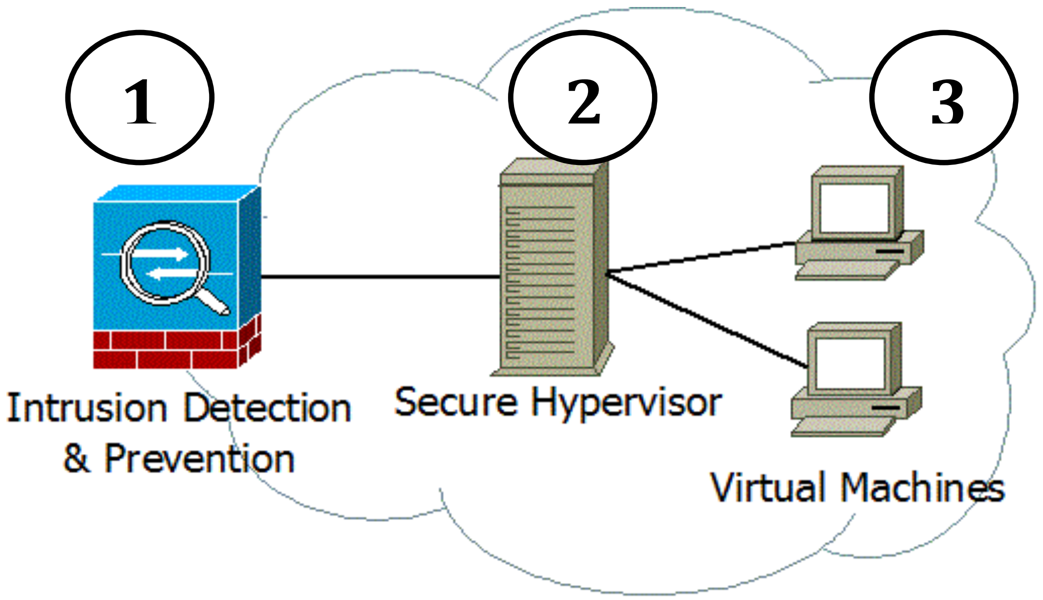


Figure 4 - Example Security Techniques in the Cloud

Omitted from Figure 4 are cloud services that provide authentication such as lightweight active directory protocol servers and trusted computing techniques as they are outside the scope of this survey. Initially, the attacker has to overcome or bypass the intrusion detection and prevention systems typically employed at the cloud boundary. They are then faced with a secure hypervisor usually installed on a single host; whose purpose is to restrict access to kernel and hypervisor data structures. Finally, cloud resilience, is used by a host to restore a single compromised or failed virtual machine to a known good state. Although not currently prevalent throughout the industry, hypervisors offer the opportunity to restrict the attacker's access to the base of the software stack. Since typically the number of vulnerabilities is directly related to the number of source lines of code [27], this would allow tight control of the hardware and allow operating system designers to build successive layers on a secure base of trust. The small size of the hypervisor also opens the door to formal reasoning concerning its security properties [39]. Unfortunately, these ideas have yet to be cohesively integrated and their impact upon security quantified. In the sections that follow we explore the building blocks that are available for improving cloud security and assess them on the basis of their *performance impact,* ability to *reduce the attack surface*, *detect known and zero-day threats*, *resolve detected threats*, and *increase attacker workload* by denying either surveillance or persistence.

2.1 Threat Model

The security implementations analyzed in this chapter and the thesis as a whole, address the threat model for intrusions employing remote control outlined in Figure 5. Attacks involve several steps that begin with *surveillance* to determine which vulnerabilities exist [40]. Vulnerabilities may revolve around specific people, processes, organizations, and network infrastructures. Based on the available vulnerabilities, the coordinated attack involves multiple initial *touches* on target networks that develop points-of-presence through some form of *implant* [41]. The initial touches may use remote user- or kernel-level exploits [42], insiders with legitimate user-level accesses [43], theft of legitimate credentials [44], supply chain interdiction [45], physical attacks at an end-point device [46], and a wide variety compromises based on radio frequency and infrastructure weaknesses. Development from the initial points of presence may involve privilege escalation [42], removing exploit artifacts, and hiding behavior [20]. On-going surveillance may involve obtaining a copy of the binary codes and using reverse engineering [47,48] or fuzzing [49] to open additional attack vectors. The implants then *persist* for a time sufficient enough to carry out some malicious effect, obtain useful information, or propagate intrusion to other systems [50].

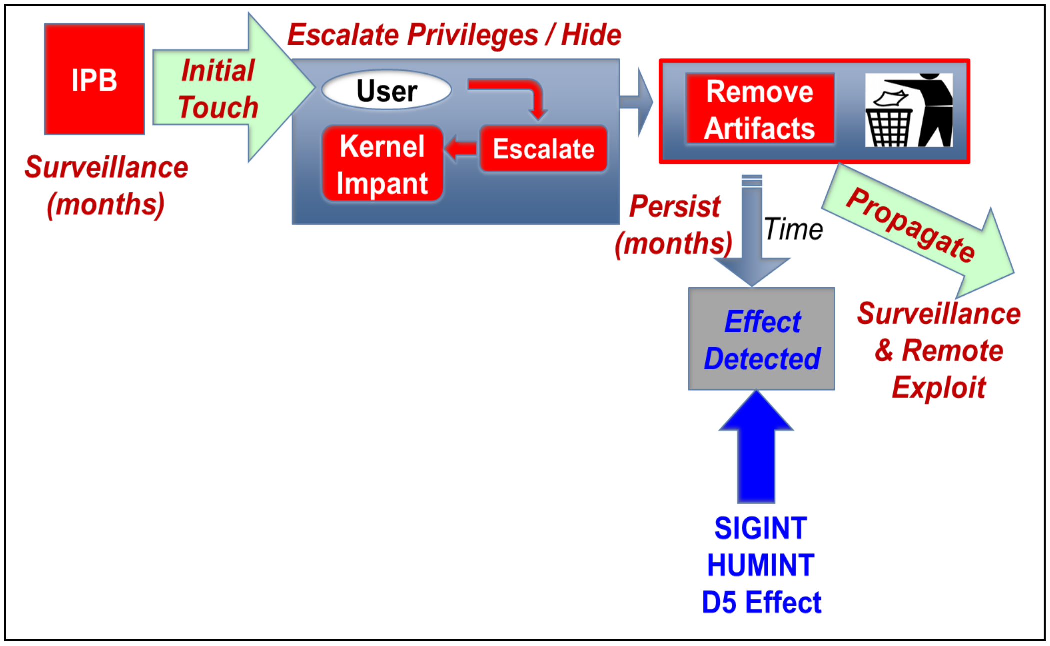


Figure 5 - Threat Model

Unlike the time to execute an exploit, the time spent in surveillance and persistence may range from minutes to months or even years depending upon the desired effects. Moreover, the presence of an intrusion may never be detected by network or on-host defenses but instead may be recognized only indirectly in a deviation from expected behavior, or may be derived from outside sources.

Nevertheless, each cloud security technique represents an integral building block in the multilayered defense of the cloud. Malware detection and prevention systems are the initial line of defense in preventing an attacker from gaining a foothold on a cloud. The secure hypervisors present a hardened code base that restricts access to hardware to all, but the most privileged operations. Lastly, cloud resilient solutions are present to protect against the unknown exploits, which may allow an attacker to operate on a cloud indefinitely.

2.2 Malware Detection and Prevention

Malware detection was one of the first techniques implemented after the introduction of hypervisors. To achieve this, researchers paired the proven technology of Intrusion Detection Systems (IDS) with the ability to hide in a virtual machine. In this scenario, the IDS still performs the same function of identifying patterns of malicious behavior on a system that may be compromised [51]; for example a proof of concept based on the Snort IDS successfully prevented a Distributed Denial of Service (DDoS) attack [52]. This implementation installed a virtual machine that ran Snort on top of the VMware hypervisor to monitor network traffic to all guest virtual machines attached to a virtual switch. Once running, the IDS dealt with DDoS attacks in two steps: Initially, attacking computers were blocked by Snort; subsequently, the virtual server automatically moved the application under attack to a new location in the cloud. This demonstrated that an IDS can function inside the cloud; however, the implementation was just as vulnerable to zero-day attacks as non-virtualized IDS's [53]: attacks were missed due to IDS configuration and the failure of signatures to detect new attacks.

The Hybrid Virtual IDS is a solution that leverages the strengths of the cloud and improves upon the previous Snort implementation [54]. The approach combines resilience of a virtual IDS and the versatility offered by a host based IDS. This is possible through the use of integrity checking [55] and system call trace analysis [56]. Integrity checking is a static detection process in which a changed file is compared to a gold standard to determine if the change is malicious. System call trace analysis dynamically flags anomalous system call behavior as potentially dangerous. These two approaches are implemented inside of a virtual machine to provide an isolated environment. A custom hypervisor is then used to ensure the isolation between all virtual machines. To provide functionality to the IDS, the hypervisor has hooks that allow the inspection of other guest virtual machines running on the hypervisor. This allows the hybrid virtual IDS to remain isolated from other running virtual machines, while still allowing it to access data from the virtual machines it is monitoring. This technique performed well in testing conducted by the authors of the Hybrid Virtual IDS, but returned unexpected performance results: as the IDS decreases the length of time between inspecting of the monitored virtual machine, the workload processing time did not increase linearly as to be expected and instead became erratic. The cause of this erratic performance is open to additional research.

With the introduction of a hypervisor and a virtualized IDS, it was only a matter of time before firewalls were moved into the cloud. One of these virtual firewall implementations is VMwall [57], which runs in the privileged virtual machine that controls the Xen hypervisor and uses virtual machine introspection [58]. This is the process of inspecting the data structures of a separate virtual machine. To enable this functionality, the Xen hypervisor has added hooks that capture all network connections created by a process. The data pertaining to these connections is then passed to VMwall for analysis. The connection is either allowed or blocked by using a whitelist (a list of approved processes and connection types). To deter false data during introspection, kernel integrity checking [59] is used to verify the state of kernel data structures in the guest virtual machines. This is necessary, as the primary method of inspecting traffic is through these data structures; malicious modification may compromise the monitoring of traffic. However, VMwall may be vulnerable to hijacking of a whitelisted process or an already established connection. The only method of detection against the compromise of an approved process is through the checksumming of the in-memory image of that process. This is performed by ensuring that the hash of a process has not changed from that of one contained in the whitelist. Due to the performance impact of hash analysis, this method is generally not implemented. Hijacking an established connection can be partially prevented through time outs associated with kernel rules contained in the whitelist. To fully prevent this type of compromise, deep packet inspection could be used, but is not currently employed by VMwall. Importantly, the employed introspection techniques cause a minimal performance impact: the additional overhead is 7% for file transfers from hypervisor to guest and 1% for file transfers from a guest to the hypervisor. Added overhead for Transmission Control Protocol (TCP) and User Data Protocol (UDP) connections are negligible; increases are measured in microseconds.

An alternative approach to detection techniques, like VMwall and hybrid IDS, are prevention methods. One security appliance that performs prevention is Malaware, which is designed to prevent malware that tailors attacks upon detection of a hypervisor [60]. To deter this initial identification of a virtual environment, a signature based method is used. In this instance, a signature is an instruction that should not be executed by an unprivileged process. As an example, when a process such as Red Pill attempts to run the SIDT instruction, it will be flagged as malicious. However, as the authors of Malaware have stated, a signature based approach is only effective against known types of malware. To combat zero-day threats, two behavior based approaches that utilize dynamic analysis are proposed [61]. This could be accomplished by first learning the current process and its page table base address. With this, it is possible to check if the current instruction register belongs to the process' code pages. If this mapping does not exist, Malaware could flag the process as malicious. The second dynamic analysis method suggested is taint tracking. Changes to the system, otherwise known as taint, are created, when a process modifies any code or memory location. Accordingly, when taint is created in monitored locations, the offending process is immediately flagged as malicious. An added benefit of taint tracking is it defeats malicious code that has been transformed to look harmless, also known as code obfuscation [62]. Once loaded into a monitored region, the obfuscated code is immediately marked as tainted and the associated process is flagged as malicious. Unfortunately, only the signature based piece of the detection has been implemented and no data relating to added overhead has been collected. However, the initial detection results were promising with a malware detection rate of 76%. Lastly, it is important to note that techniques that alter known memory states, such as address space layout randomization (ASLR) may increase the difficulty of this type of taint tracking [63].

Another prevention method, guest view casting [64], moves malware prevention from the guest virtual machine to the hypervisor. This approach reconstructs the data structures of the guest for analysis at the hypervisor level. This is achieved by translating guest virtual memory addresses to physical memory addresses, then reading the raw data from the guest's virtual hard drive. The reassembled state in the hypervisor can then be compared to the guest's state using viewing tools such as Windows Task Manager and memory dump to display all processes in memory. The presence of discrepancies between the two states may indicate the existence of malware in the guest. The authors have labeled this method of searching for discrepancies between states as *view comparison-based malware detection*. An outgrowth of this method is to use anti-virus software to scan the guest's state from inside of the hypervisor. The use of anti-virus outside of the guest shows that it identifies malware more effectively than anti-virus running inside a virtual machine. Additionally, performance of anti-virus is improved outside of the virtual machine. The primary drawback to this approach is the assumption that the hypervisor has not been compromised. The authors agree that malicious code that targets the hypervisor [65] can compromise their approach.

Although detection and prevention are important, the last two decades have demonstrated that it is unlikely that malware can be eliminated completely [66]. Security researchers in an attempt to understand these attacks have to rely on system logs that lack integrity [67] and are often incomplete [68]. The ReVirt IDS [69], which runs on UMLinux [70]; was created in an attempt to improve upon these inadequacies. This is accomplished by creating logs for all of the relevant system level information needed to replay what transpired at an instruction by instruction level for a specific virtual machine. This allows administrators to determine all the relevant information pertaining to an attack. The overhead of performing these functions is 13-58% for kernel tasks and up to 8% for logging tasks.

2.3 Secure Virtual Machine Managers

Hypervisors have afforded researchers with new security capabilities. However, the hypervisor itself has come under attack as a way of gaining control of a system [71]. This has led to the introduction of Secure Hypervisors that reduce the attack surface and increase reliability by reducing the number of lines of code [27]. sHype [72], designed by IBM, increases security by taking the idea of control flow enforcement first seen in SELinux [73] and applying those controls on information flows between virtual machines through a mandatory access control model. Using intricate security policies; unfortunately, these make it difficult to guarantee security and can be over 50,000 lines of code [74]. To remove this level of complexity, sHype affords the same control flow protections, but at the hypervisor level and without the need of a policy administrator. These information flows are maintained through the use of a reference monitor that decides what connections to accept and deny between virtual machines. The sHype approach creates a flexible architecture, which allows it to support many different security modules [75]. This is accomplished in around 11,000 lines of code; SELinux alone is over 85,000 lines of code.

The performance impact of sHype enforcement policies is less than 1% [72]. However, sHype's primary shortfall is that it does not completely protect against unauthorized transfer of information between two virtual machines that are not allowed to share information. Figure 6 illustrates the problem: nodes A, B, and C represent three different virtual machines and all are connected to a reference monitor.

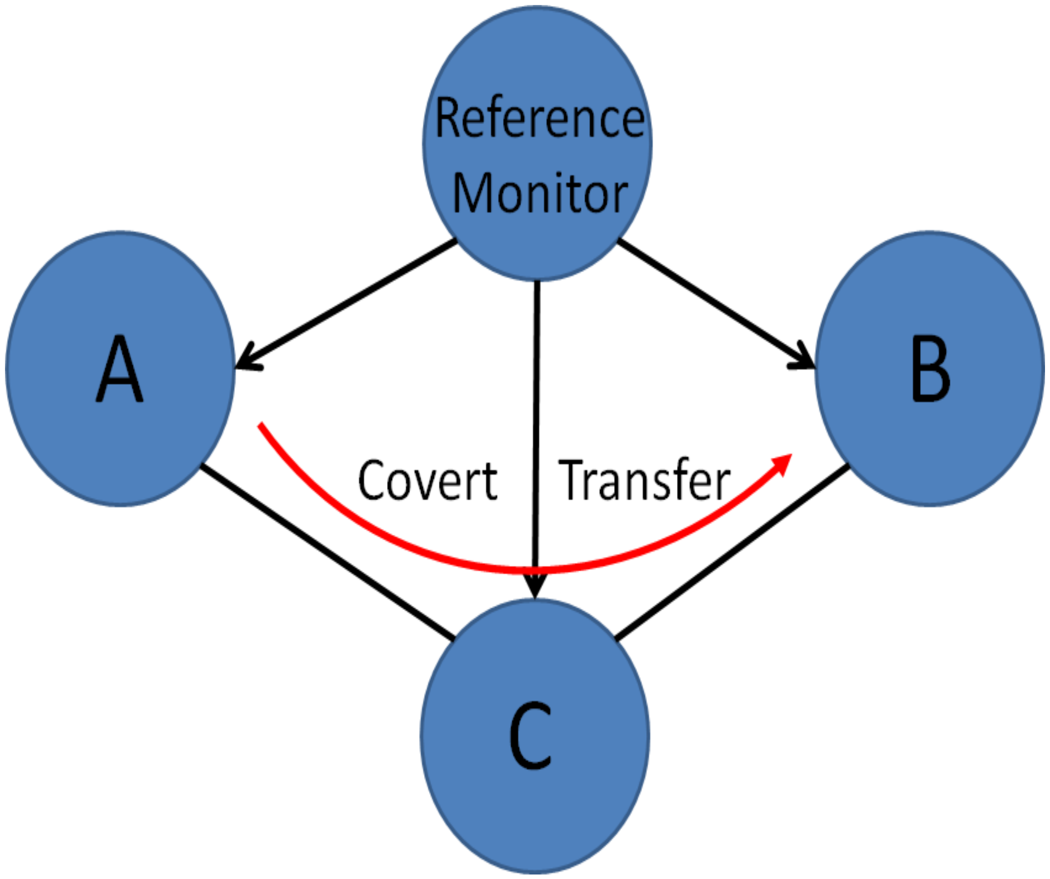


Figure 6 – Example Covert channel

Virtual machines A and B are not allowed to share information, but both are allowed to share information with virtual machine C. A covert channel is created, when virtual machine C acts as an intermediary and passes information between A and B. In this case the reference monitor would not intervene, as it only sees information being transferred from A to C and from C to B. Fortunately, the addition of a Chinese wall (communication rules) can be added to sHype to protect against this covert channel [76]. In this case, the rule would only allow two of the three virtual machines to run at any one time. However, this method has the drawback of causing a decrease in performance of up to 9.1% [77]. This performance impact can be mitigated by performing Chinese wall policy checks at virtual machine creation and then caching these decisions. Since, policy changes are infrequent, this configuration reduces the performance impact to less than 1% [78].

A different direction from control flow enforcement is used in the noHype hypervisor [79]. This minimalist approach removes as much as possible from the hypervisor; unfortunately, no published numbers for lines of code are available. However, the first prototype was based on a stripped down version of Xen 4.0; implying that it falls somewhere less than 1.6 million lines of code [80]. The code count was reduced by shrinking the size of the hypervisor by following four rules. First, noHype pre-allocates processor cores and memory to virtual machines. This allows the virtual machine to control its own hardware, which improves performance. Second, each virtual machine is assigned its own I/O device. Being in the cloud, it is assumed that these virtual machines only need network interface cards (NIC). The issue here is that servers have a limited number of NICs. Thankfully, newer NICs take advantage of Single-Root I/O Virtualization [81], which allows them to present themselves as multiple NICs. Thus, each virtual machine on a server is able to receive its own NIC, even if there are more virtual machines than NICs. Third, noHype provides the user with a predefined guest virtual machine in order to control the discovery of hardware. This also prevents a user from uploading a malicious guest virtual machine, which could attack the hypervisor. Lastly, noHype avoids indirection that occurs through the creation of virtual cores and memory, since cores and memory are assigned directly to each virtual machine. These four principles were tested against a standard Xen 4.0 install and startup time was reduced by 1% in the noHype implementation. However, noHype loses the ability to perform any introspection of the guest virtual machines as the hypervisor is limited in functionality. Thus, a virtual machine in the noHype cloud could become infected without noHype being aware of the infection.

Another popular feature of the cloud is live migration of virtual machines [74]. This can be seamlessly accomplished with little downtime thanks to virtualization. However, migrations lose the states maintained by stateful firewalls [82] and IDS' [83]. These states can be maintained using a network security enabled hypervisor (NSE-H) designed on top of the Xen hypervisor [84]. This builds on the concepts used in secure hypervisors, but adds support for secure file transfers. The performance impact of this method is measured in downtime, which is the time a virtual machine is not available during transfer. The cost of securing these migrations is up to a 15% increase in downtime versus downtime of non-secure transfers [85]. This downtime occurs for two reasons when maintaining the security context of the virtual machines being migrated. The first is the additional time needed to securely copy a virtual machine's memory space from one host to another. The second is the NSE-H security additions, as they are using additional resources on the system.

2.4 Cloud Resilience

An often over-looked aspect of cloud computing is *Resilience*, defined as the ability for a system to recover and continue to provide services when a loss of hardware or software occurs [86]. One such system, Cloud Resilience for Windows (CReW) [87], expands the idea of resilience to the security domain through the use of strong security in guest virtual machines [88], and introspection [89]. Implementation is on top of the 270,000 plus lines of code that comprise the kernel-based virtual machine hypervisor [90]. This has enabled CReW to effectively prevent attacks from some rootkits and repair any damage they may have caused, but at a cost to performance as the number of virtual machines increases or security level is raised. At a strict level with three virtual machines, CReW adds ~48% increase in time needed for CPU tasks and ~279% increase in time required for I/O related tasks. For the paranoid setting, CReW adds ~116% increase in time for CPU related tasks and adds ~347% increase in time for I/O related tasks [87].

A technique that builds upon the ideas presented in CReW and supports other operating systems is that of *hypervisor-based efficient proactive recovery* [91]. This approach makes the assumption that no matter what defense is implemented on the cloud, a machine will eventually be maliciously compromised or taken offline. Thus, after particular failure conditions are met, the guest virtual machine is refreshed from a gold standard. A prototype of these concepts was developed using a modified Xen hypervisor [92]. Testing has shown there is a balance between throughput and availability. Thus, a user of this method can choose between lower throughput and higher availability or higher throughput and lower availability when faults occur.

The Bear operating system is a minimalist implementation that builds resiliency on top of a secure hypervisor [25]. A key design choice is the strong enforcement of separating core functionality into four layers, which is typical of modern micro kernels, like the MINIX operating system [93]. Importantly, the attack surface is reduced with a shared code base (>50%) of 10,903 lines of code shared between the Bear Hypervisor and Kernel. The size is attributable to a small custom hypervisor and small custom kernel. Resiliency is derived from non-deterministically refreshing the virtual machines on the hypervisor to a gold standard after a period of time. This refresh is done by starting a second virtual machine from the known valid state and then transferring functionality to it, all while simultaneously tearing down the first virtual machine. By using this method, control is seamlessly transferred between virtual machines and without an impact to performance. Also, any known or zero-day malware present on the torn down virtual machine will not be present on the newly started virtual machine.

2.5 Comparative Analysis

Table 1 presents a summary comparison, of the approaches based on reduction of the attack surface, prevention of zero-day threats, and overhead. The "Reduces

Attack Surface" category shows that all of the technologies other than sHype and Bear rely on a large code base.

Cloud Security Implementation	**Reduces Attack Surface (lines of code)**	**Malware Detection**	**Mitigates Zero-Day Threats**	**Added Overhead (%)**
Malaware	> 725K	yes	no	no data
Guest View Casting	> 1,600K	yes	no	Reduced up to 70%
Virtual Snort	> 300K	yes	no	no data
Hybrid IDS	> 300K	yes	no	~4-36%
VMwall	~ 1,600K	yes	no	1-7%
ReVirt	~ 1,800K	no	yes	8-58%
NSE-H	> 1,600K	no	no	15%
Shype	~ 11k	no	no	< 1%
Shype with Chinese wall in Critical Path	> 1,600K	no	no	9.1%
Shype with Chinese wall outside Critical Path	> 1,600k	no	no	< 1%
NoHype	< 1,600K	no	no	Reduced up to 1%
CReW	> 270K	yes	yes	~48-347%
Hypervisor-Based Proactive Recovery	~ 1,600K	yes	yes	~8-12.7%
Bear	~ 11k	Not Applicable	yes	< 1%

Table 1 - Comparison Summary of Surveyed Systems

This poses a concern, as demonstrated by the authors of "Reliability Issues in Open Source Software", who have shown that errors occur at a rate of .09 defects per thousand lines of open source code. This problem is worse for closed source systems, with .57 defects per thousand lines of code. Although the numbers will vary with code base naturally, this result that indicates Xen will have 144 defects, KVM 25, UMLinux 162, sHype and Bear each present a single defect. An interesting comparison was provided between open source software and closed source software. Due to the partial unintended release of 300,000 lines of VMware kernel code; the code could contain up to 171 defects, which is more defects then a full install of UMLinux. Obviously, sHype and Bear systems are a bare minimum install and have less functionality when compared to the other hypervisors. This has led to the sHype architecture being ported to the Xen hypervisor by the authors of *"Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor"*, which has the net effect of increasing functionality and potential number of defects. The key takeaway is that a small code size and open source distribution are desirable to prove a system to be reliable and secure. However, closed source systems, which are outside of the purview of this article, do exist and provide similar security features. Two such commercial hypervisors not reviewed are Citrix XenClient and HyTrust.

After evaluating each system on its abilities to perform "Malware Detection" and "Prevents Zero-Days"; there were two clear outliers. Malware detection and prevention methods primarily protect against known threats, because of their use of whitelists and signatures. However, ReVirt is the outlier in this category, as it provides capabilities to remove zero-days; unlike its counterparts, it has no ability to detect malware. Secure hypervisors restrict access to the hypervisor but generally provide no malware detection abilities or zero-day prevention. Lastly, resilient systems such as CReW and hypervisor based proactive recovery have shown promising results in both categories. The model of whitelists and signatures is replaced with restoration upon detection of anomalous system behavior. Thus, both known malware and zero-days are removed from the system when it is restored to a valid state. Resilient systems do not prevent the initial compromise from known threats, unlike malware prevention and detection systems. The outlier in this group is Bear, which makes no attempt to check for anomalous behavior. Instead, it assumes the system will eventually be compromised and therefore refreshes the system non-deterministically. This has the same end result of removing any known or zero-day attacks that may be present, but also invalidates surveillance and prevents persistence. Nevertheless, the effectiveness of resilient systems warrants further research.

The final category of “Added Overhead” is important, as no technique should overly impact system performance. Both secure hypervisors and malware prevention and detection schemes can minimally impact and in some cases improve performance. The larger resilient prototypes such as CReW and hypervisor proactive recovery have not yet reached this level of performance. Bear however, has had a negligible impact on performance when refreshing virtual machines. Research into future resilient system implementations should aim to maintain the performance levels set by intrusion detection and prevention systems, secure hypervisors, and the Bear operating system. This can be achieved by leveraging the proven practices of either adding functionality to the hypervisor as seen in Guest View Casting or reducing the hypervisor foot print as accomplished by NoHype and Bear. Once this performance requirement is met, further capabilities can be added to resilient systems, which allow for the creation of a new cloud security architecture.

2.6 Related Fields of Work

One field of study that has not been included in this survey is the idea of trust [94] in regards to the unauthorized access of data. One approach to handle trust in data security is that of security labels in the cloud [95]. The goal of this approach is to isolate customer virtual machines from each other to prevent data leakage across virtual machines. This work is an enhancement of a trusted hypervisor that extends trust to network storage [96]. In regards to privacy, customers are concerned that their personal information will be leaked to those who should not have access to it. A current solution to this problem is the use of encryption with access control [97]. Using public key cryptography in the cloud, the user can be sure that their data is safe and only they have access to it.

2.7 Summary

All of the techniques reviewed in this chapter have produced gains in making cloud computing more secure. Most of the solutions strive to race to the bottom of the software stack to combat known risks, rather than unknown zero-day risks. Moreover, it is currently left up to the cloud provider to pick from a grab bag of techniques to secure their infrastructure, which often times reduce performance in a time sensitive environment. This has led to a diverse set of approaches in cloud security, each with its own goals.

However, many of these same methods have increased the size of the code base of both the kernel and hypervisor. According to Pandey et al., this represents a real risk in that additional bugs may be present in the code. These same bugs may lead to real vulnerabilities, which coupled with the large *attack surface* of the system provide the perfect environment for ROP attacks. All of this is only made worse by the homogeneity of the cloud running thousands of copies of the same virtual instance.

Thus, a different approach is needed that combines the most successful techniques to leverage multiple cores with hardware virtualization security in new ways. A new cloud infrastructure should follow the example of noHype to minimize the hypervisor *attack surface*. From there to break the monotony of guest virtualization, the standard kernel should be broken into smaller pieces that provide individual services. These smaller VMs can then be sandboxed through hardware isolation in a manner that is similar to sHype. By controlling access and information flow to these systems, anomalies due to malicious behavior can be quickly isolated and resolved. By doing all of this, a new utility virtual machine architecture would be created that increases *attacker workload* and reduces the *attack surface* used in ROP attacks.

Chapter 3 – Symmetric Multiprocessing

The utility virtual machine architecture requires the linchpin of modern computing that is x86 symmetric multiprocessing. These multiple cores are required for static assignment to UVMs to provide both security and timely performance. Therefore, a solid understanding of these technologies is critical not only for this work, but also to any developer entering the field.

Unfortunately, the complexity associated with discovering, enabling, using, and virtualizing multiple cores has created a paucity in the available documentation, transferable knowledge, and readable code exemplars. This chapter describes our experience in overcoming these hurdles in the design of a from-scratch, multi-core operating system – *Bear* – for utility virtual machines. In particular, intricacies involved in the development of a multi-core micro-kernel with an integrated multi-core hypervisor are traced. By exploring the implementation details, from bootstrapping through core virtualization to process scheduling, this paper aims to fill the knowledge gaps, highlight potential pitfalls, and introduce multicore development in a concise start-to-finish exemplar.

3.1 Introduction

Sadly, it has become increasingly difficult for systems programmers and developers to leverage the full features of x86-64 technology effectively. For example, the documents containing the information for finding, bootstrapping, and operating multiple cores are spread across multiple separate large manuals including, the 1,056 page Advanced Configuration Power Interface (ACPI) Specification [98], the 97 page Multi-Processor (MP) Specification [99], and the 3,463 page Intel or 664 page AMD Software Developer's Manual [5,100] respectively. These specifications are in turn maintained by a multitude of parties, each with their own vested interests. For example, the MP Specification written by Intel was deprecated in favor of ACPI, which was written by a consortium of computer hardware and software manufacturers. The ACPI specification was then absorbed into the Intel written 1,084 page Extensible Firmware Interface (EFI) Specification [101]. A similar consortium of hardware and software manufacturers soon absorbed the EFI Specification into the 2,637 page Unified Extensible Firmware Interface (UEFI) Specification [102]. Fortunately, each newer specification is required to be backwards compatible with any of the older specifications to support legacy specifications. This limits the reading material for an entry level SMP programmer to the 5,280 combined pages of the MP specification, ACPI specification, Intel manual, and AMD manual. Unfortunately, Intel and AMD add to this level of complexity with their x86 chips, which support many operation modes, be it through legacy Port I/O, Memory Mapped I/O, or Model Specific Registers. These backwards compatibilities sometimes have a peculiar set of consequences.

Recently, the most significant problem for X86-64 processors, stemming from backward compatibility, resulted in a privilege escalation attack that gives an adversary full control of the system [103]. In this example, the enduring backward compatibility concerns the ability for the processor to move internal memory mapped I/O control registers. This feature was provided so that the processor could move its own registers to a new memory location if software was already using the same memory address; a valuable capability when x86 processors are operated with 32-bit addressing and virtual memory is limited to only 4 GB. In practice, with the introduction of 64-bit addressing (x86-64), the potential for overlap rarely occurs since virtual memory was expanded to 256 TB; however, the feature to move the processors memory mapped I/O registers remained in place. This alone was not a security vulnerability, until the introduction of a new layer of processor operation and security, that operates below the kernel and hypervisor, was introduced: This layer -- the system management mode (SMM) [100] -- has control over the underlying physical hardware of the system, for example, the processor cooling fans. SMM is accessed and configured through memory mapped I/O, but access to it is heavily restricted through specialized x86-64 instructions. Unfortunately, these instructions and restrictions can be completely bypassed by an attack by using the ability of the processor to move its internal memory mapped I/O addresses to memory: moving them to overlap the memory reserved for SMM. Thus, an attacker can gain control of the system, using less privileged processor memory mapped I/O registers, to read and write directly to SMM.

This example, the volume of the combined specifications, and the complexity of hardware support for security -- available through multi-core isolation, virtualization, and 64-bit paging and protection structures -- virtually guaranties that systems designers must operate behind a impenetrable veil of interrelated constraints. Consequently, many developers reach only a superficial level of understanding and then abandon the more sophisticated concepts, relying instead upon existing implementations associated with monolithic operating systems, such as Linux [104] or hypervisors such as Xen [1]. This is particularly true of research operating systems that utilize only the most basic processor support [26,105,106].

This paper redresses these shortcomings by providing a complete path to multicore via an all-in-one description of a minimal, virtualized multi-core system composed of a micro-kernel with an integrated hypervisor. The description uses well-known implementation techniques, to focus attention on the use of the underlying architectural support. For example: cores are statically partitioned among virtual machines, each virtual machine runs a single micro-kernel, each microkernel schedules processes round-robin across cores owned by the virtual machine and mutual exclusion of multiple cores from the scheduling queue uses a single global lock (similar to early versions of Linux). These design choices can readily be improved and optimized using well known techniques that are unrelated to the underlying hardware concepts. Although reasonably straightforward, the design performs surprisingly well when compared with Fedora, Ubuntu, and Xen; mature systems that have undergone hundreds of man-years in development and optimization. This can be directly attributed to extensive use of Intel 64-bit hardware mechanisms employed through their recommended implementation methods.

3.1.1 Basic Concepts

Basic Input/Output System (BIOS) [107] – The BIOS constitutes firmware installed on a system that is the first code to run when the system is powered on. The BIOS is responsible for initializing all of the core hardware components (i.e. VGA, Network, Processor Cores, Memory, etc.) into a known good starting state, specified by the hardware vendor. The BIOS specification for multicore processors, detailed in the Intel Developer Manual [5] states: "*The MP initialization protocol defines two classes of processors: the bootstrap processor (BSP) and the application processors (APs). Following a power-up or RESET of an MP system, system hardware dynamically selects one of the processors on the system bus as the BSP. The remaining processors are designated as APs.*" Upon completing initialization the BIOS loads the Master Boot Record (MBR) from either a physical hard drive, disk, or memory disk (RAMDISK) transferred over the network. Once loaded the BIOS turns execution over to the BSP, which starts execution at the beginning of the loaded MBR.

Master Boot Record (MBR) [108] – The MBR emerged with the PC DOS 2.0 system in 1983 and corresponds to the first 512 bytes of code loaded from disk by the BIOS; it is subsequently executed by the BSP. The MBR is restricted to 512 bytes in size, an historical artifact associated with dividing hard drives into cylinders, heads, and sectors (CHS) for addressing [109]. It is important to know that the MBR must conform to this addressing scheme and that it uses CHS to load the next 31.5KB from disk, 512 bytes at a time. The next blocks of code loaded by the MBR, shall refer to as the stage one and stage two bootloaders; their functionality and purpose will be discussed at length in this paper. Over the years the MBR has become a standard element of modern personal computers and thus, while not discussed further here, it is still an important legacy component of modern computer systems.

Bootstrap Processor **(BSP)** – The BSP, initialized by the BIOS in a multiprocessor system, is the one and only processor core on a multicore processor with the "*IA32_APIC_BASE Model Specific Register (MSR) set*" [5]; this flag signifies the core that is the sole processor to begin execution. The BSP begins execution in *16-bit real mode* (explained below) at the start of the MBR. In this paper the system described runs on an 8-core Intel i7 multicore processor. Thus, throughout the paper, the standard term BSP is used to refer to the core that was designated by the BIOS as the sole startup processor core.

Application Processor(s) **(AP)** – All other processor cores on the system "*have the IA32_APIC_BASE MSR cleared"* and are known as the AP cores [5]. These AP cores are initially placed by the BIOS into a *HALT* state [99]. They can only be used by the operating system after the BSP has made the necessary system configurations, and subsequently signals each individual AP to begin execution.

As with the BSP, this paper uses the standard term AP to describe any of the non-BSP cores on the system.

Symmetric Multiprocessing **(SMP)** – Linking the BSP with a set of identical APs in a "*tightly-coupled system where some or all of the system's memory and I/O facilities are shared"* [110] is the basis for an SMP system. On x86 systems this coupling is provided by a system bus that allows the BSP and APs to access memory and I/O, as well as exchange interrupts [99]. The main line of communication between cores occurs through inter-processor interrupts delivered using an Advanced Programmable Interrupt Controller (APIC), each BSP and APs has its own, dedicated local APIC.

Advanced Programmable Interrupt Controller **(APIC)** – In general, the APIC provides both interrupt *routing* and *redirection* between cores; this was not possible with earlier interrupt controllers such as the Intel 8259 Programmable Interrupt Controller (PIC) [111]. SMP is enabled when the BSP's local APIC generates a start signal to each APs local APIC along the shared system bus [99]. Although, inter-core interrupts can also be used to support multi-threading (by allowing one core to inform another of Transition Lookaside Buffer shootdowns [112]), this paper focuses on SMP initialization as the micro-kernel design replaces shared memory with message-passing to enhance security.

Input/Output APIC **(I/O APIC)** – The APIC architecture, as defined by Intel, is split between two components: the Local APIC associated with each core, and a discrete I/O APIC [99]. The I/O APIC is used to re-direct interrupts generated by external peripherals, such as the network card or keyboard, to one of the cores on the system.

16-bit Real Mode – Recall that, the BIOS hands the hands off control to the BSP in 16-bit real mode. This was the first memory-addressing mode instituted in the Intel 8086 processor [113]. It provides no support for memory protection, multitasking, or operating system ring levels: the BSP is provided with full access to the physical memory and hardware [5]. The importance of this mode is that the BIOS can still be accessed as long as the BSP operates in this mode. This allows the BSP to probe the BIOS for key system information, such as the amount of physical memory installed.

32-bit Protected Mode – Protected mode was introduced with the Intel 80286 processor [114] and featured a number of improvements over real mode. For the first time, operating system ring-level security was supported in hardware, as well as memory protection through segmentation; eventually paging and multitasking (through the Task State Segment Register) [5] were also added. Unfortunately, the memory protection bits on page tables in this mode only provided markings for a page to be read-only, read/write, user, or kernel. Thus, to enforce a full suite of MULTICS-style read/write/execute protections [29], segmentation must be combined alongside paging. In practice, segmentation has not been widely adopted; consequently, this paper uses protected mode only to quickly transition into 64-bit long mode.

64-bit Long Mode – Long mode was introduced with the AMD Athlon 64 line of processors [100] and was meant to address some of the limitations of protected mode; primarily, limited addressing allowing for only 4GB of virtual memory. Long mode is expandable up to 16 exabytes (10^{18}) of virtual memory, though current processors are limited to 256 TB. Segmentation was completely replaced by 64-bit paging in long mode. A no-execute (NX) bit was also added to the paging structures to strengthen MULTICS-Style protections, however, it should be noted that these structures lack the ability to mark a page *execute-only*.

Control Registers **(CR)** [5] – The BSP and APs each have their own set of CR's, just as each has its own local APIC. Each core must setup its CR's before using other processor functionality. This paper focuses on programming of CR0, which provides basic processor operating modes, CR3, which references the physical RAM address of the start of a page table in use by a core, and CR4, which provides advanced processor capabilities. For example, each core must set bit 13 of CR4 to 1 if the Intel virtualization extensions (VT-x), are to be used to support a hypervisor. The complete layout of these three CRs is available in the Intel Programmer's Manual [5].

Model Specific Registers (MSR) [5] – MSRs are similar to CRs in that they can be programmed to enable various core functions in the system. Some MSR registers are set either by the system (for example, the BIOS sets the *IA32_APIC_BASE MSR* to designate the BSP) or they can be set by the operating system (for example, *IA32_VMX_PROCBASED_CTLS2 MSR* designates the virtualization controls for APIC Access Virtualization). Accesses to the MSRs are provided through the x86 *rdmsr* (read MSR) and *wrmsr* (write MSR) instructions.

Memory, Paging, and Memory Management Unit **(MMU)** [5] – Physical memory constitutes all the Random Access Memory (RAM) installed in the system; It is initialized by the BIOS and information concerning its size, and the areas reserved for specific uses, can be obtained through the BIOS when operating in 16-bit real mode. Virtual memory is a conceptually much larger space that processes may utilize during their execution. Virtual memory addresses are translated into physical memory addresses, through page tables, by the memory management unit (MMU). An explanation of 64-bit paging and the associated structures is available in the Intel manual [5]. However, to understand this paper it is sufficient to understand how paging relates to SMP. In particular, to understand that the BSP and APs can each operate their own independent set of page tables, through manipulation of their own Control Register 3 (CR3), which signifies the base of a cores virtual memory. This paper introduces the concepts of *recursive paging* and a *physical address frame allocator* as mechanisms for building these paging structures. Collectively, the ideas ensure that each core provides a unique page table *per process*.

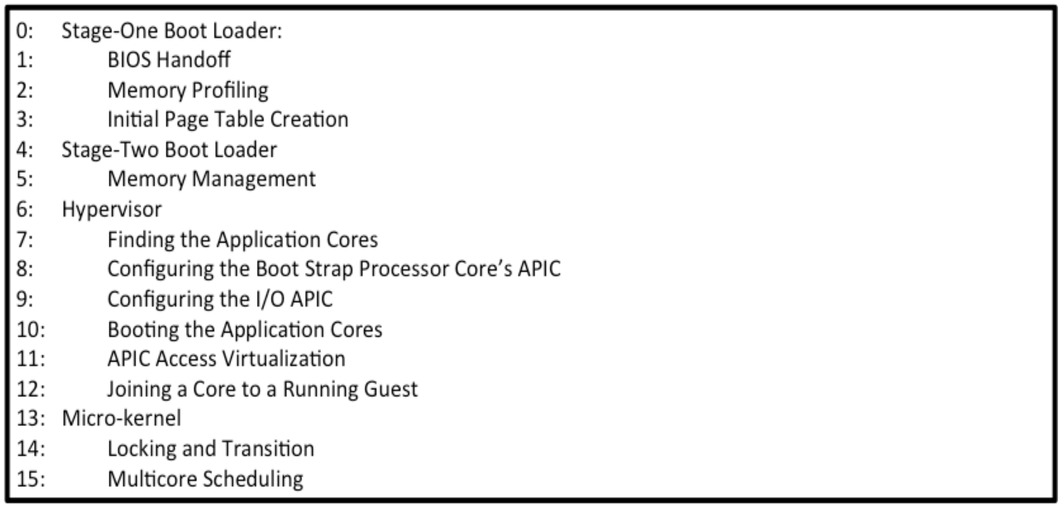
Virtual APIC (VAPIC) and APIC Access Virtualization [5] – Intel VT-x treats the APIC as a resource that must be protected from each guest virtual machine. This is achieved through a technique known as APIC Access Virtualization, which creates a virtual APIC (VAPIC) for each guest. When the guest accesses its VAPIC, the configuration of the hypervisor and virtual machine collectively determine whether a Virtual Machine Exit (VMExit) into the hypervisor should occur. VMexits, generated by the VAPIC, are central to an SMP enabled virtual machine and will be discussed in detail.

Context Switching and Processor State Storage [5] – When the BSP or an AP switches from execution of a virtual machine into the hypervisor, the state of the general-purpose processor registers (*rax, rbx, rcx, rdx, etc.*) must be saved. The associated storage is created and provided by the hypervisor, so as to preserve the running state upon VMExit. This is necessary so as to allow the hypervisor to either perform or deny any action taken by the virtual machine and to allow the hypervisor to resume execution of the virtual machine immediately after where the exit occurs. For example, if the virtual machine accesses its APIC to send an inter processor interrupt, the VAPIC will generate a VMExit, the hypervisor must then generate the associated inter processer interrupt for the virtual machine since it is a normal part of SMP execution. The virtual machine is eventually resumed after the instruction that caused the VMExit.

Extended Page Tables (EPT) [5] – Though not covered in this paper, EPT's are integral to the performance of virtualization: they provide a means of translating what the virtual machines believes is a physical memory address into an actual address in physical memory by the hardware. This removes the requirement that the hypervisor translate every memory access performed by the guest virtual machine. EPTs also provide fine-grained memory access control by adding execute, read, and write bits, which when coupled with 64-bit long mode paging provides the full set of MULTICS-style protections [29].

3.1.2 Overview

To realize the Bear system presented in chapter 1, bootstrapping steps from power-on-reset to a running set of user-processes executing on top of the micro-kernel are implemented. This overall process is outlined through abstract code in Figure 7 and begins when the MBR hand-off to the stage-one bootloader, operating on the designated BSP core (line 0).



```
0:  Stage-One Boot Loader:
1:      BIOS Handoff
2:      Memory Profiling
3:      Initial Page Table Creation
4:  Stage-Two Boot Loader
5:      Memory Management
6:  Hypervisor
7:      Finding the Application Cores
8:      Configuring the Boot Strap Processor Core's APIC
9:      Configuring the I/O APIC
10:     Booting the Application Cores
11:     APIC Access Virtualization
12:     Joining a Core to a Running Guest
13: Micro-kernel
14:     Locking and Transition
15:     Multicore Scheduling
```

Figure 7 - Overview of Steps Required for SMP Enabled System

The stage one bootloader is a mix of 16-bit real mode, 32-bit protected mode, and 64-bit long mode code. For this reason, it is compiled as its own flat binary to facilitate absolute jumps between these different memory-addressing modes. If it were compiled with ELF, the linker would interpret all jumps as 64-bit long mode jumps (*ljmp*). This would generate truncation errors as a 64-bit jump address would be cut in half for the 32-bit jump and cut to a quarter for the 16-bit jump address. The stage-one bootloader is also restricted to 512 bytes in size, so as to fit into one load block based on CHS loading. For these reasons it is written entirely in assembly code to keep it small (183 lines of code); however, it is sufficiently powerful to move the BSP from 16-bit real mode into 64-bit long mode.

The stage-one bootloader handles all interfacing with the BIOS, which can only be accomplished in 16-bit real mode (line 1). It communicates with the BIOS to obtain a map of the underlying physical memory (line 2) and sets all of the necessary CR and MSR bits for the BSP to enter 64-bit long mode and enable use of floating-point instructions. It then creates an initial paging structure, sufficient to support a stage two bootloader, written in C and compiled with ELF for maintainability, that operates in 64-bit long mode (line 3). With these tasks accomplished, the BSP will be running in 64-bit long mode and can initiate the stage-two bootloader (line 4); the system does not return to 16-bit real mode once it has entered 64-bit long mode.

The BSP executing the stage-two bootloader, uses the information obtained by the stage-one bootloader regarding memory layout to implement a full 64-bit paging system with the full range of 64-bit read-write-execute memory protections afforded by the underlying processor architecture (line 5). The stage-two bootloader is also responsible for ELF loading the hypervisor into the paging system it has created (line 6). Once, these two tasks are accomplished the BSP can then begin execution of the hypervisor.

In the hypervisor, the BSP will parse the ACPI tables (line 7) in order to configure its own local APIC (line 8) and the I/O APIC (line 9). The BSP also uses information in the ACPI tables to discover all of the remaining APs present on the system (line 10). The BSP's local APIC can then be used to send IPI's to wake the BIOS *HALTED* APs through their individually owned local APICs. The individual APs will then execute a small block of trampoline code that performs the necessary configurations to set each AP into 64-bit long mode and subsequently enter the hypervisor. Once in the hypervisor, each AP must also configure its own APIC in the same manor as used by the BSP to configure its APIC. The APs then halt execution and wait to be assigned to a virtual machine by the BSP. This process effectively gives ownership of all the available cores on the system to the hypervisor. The BSP then finalizes any virtualization configurations, such as EPT creation, needed to create a virtual machine, and launches a virtual machine on the BSP. Using APIC *Access Virtualization* (line 11) and *state storage* created for the BSP and APs, the hypervisor is then able to assign a virtual machine to any number of APs (line 12). For simplicity, cores are partitioned among a fixed number of virtual machines statically; each virtual machine bootstraps an instance of the micro-kernel, the hypervisor intercepts the APIC boot IPI to assign additional processor cores to the virtual machine rather than bootstrapping an AP as earlier performed by the hypervisor. The process of joining an AP to an existing virtual machine consists of configuring it to the same state as the cores already executing the virtual machine, i.e. the cores that share the same EPT. The micro-kernel executing on the virtual machine may then use the cores it is given by the hypervisor to schedule processes (lines 14 & 15).

3.2 Stage-One Bootloader

Pseudo code, for the operations that must be performed by the stage one bootloader are shown in Figure 8. Recall that the BSP always begins operation after handoff from the BIOS [107] in 16-bit real mode (line 0), as per the standard introduced by the Intel 8086 processor [113]. The BIOS starts the BSP execution at physical address 0x7C00, where it has loaded a Master Boot Record (MBR) [115] from a disk, which, for the system described here, contains a memdisk loaded by PXE-boot. The sole purpose of the MBR is to load the stage-one and stage-two bootloader binaries from the memdisk through CHS loading. Since the MBR is a well-known industry standard, it is not described in detail here. Once in the stage-one bootloader, the BSP performs the necessary CR and MSR register configurations to quickly move into 64-bit long mode. This affords the system the full value of memory protections offered by 64-bit paging as early in the boot process as possible. The BSP operating in the stage-one bootloader also polls the physical memory through the BIOS in preparation for the stage-two bootloader to further develop the 64-bit paging system. Due to the intertwined and complex nature of 16, 32, & 64-bit boot code, it is explained in three parts: CPU configurations, physical memory profiling, and initial page table creations. Line number annotations are used to document where each touch point occurs in the pseudo code in Figure 8. The complete code for the stage-one bootloader can be seen in Appendix A.

```
0:  16-bit Real Mode Operation
1:      Setup a stack
2:      Use BIOS Interrupt 15 to map memory
3:      Use BIOS Interrupt 10 to configure VGA
4:      Load a Global Descriptor Table
5:      Perform jump to 32-bit Protected Mode
6:  32-bit Protected Mode Operation
7:      Configure Control Register 4 for virtual memory paging
8:      Configure the Extended Feature Enable Register for 64-bit Paging
9:      Configure the initial 64-bit page tables
10:     Load the base of the initial page table in Control Register 3
11:     Build a recursive paging entry
12:     Turn on paging in Control Register 0
13:     Load the long mode Global Descriptor Table
14:     Perform  jump to 64-bit Long Mode
15: 64-bit Long Mode Operation
16:     Configure Control Register 0 & 4 for Floating Point & SSE Instructions
17:     Jump to the Stage Two Boot Loader
```

Figure 8 **- Stage-One Bootloader Pseudo Code**

3.2.1 CPU Configurations

Recall that, inside the hardware of the BSP or AP cores reside CRs and MSRs that control the functionality of the core. Setting these registers correctly is critical for building an SMP hypervisor. Any misconfiguration will lead to unexpected behavior at later stages in the development process. To alleviate these challenges, the stage-one bootloader handles much of the implementation required to prepare the BSP for operation in the stage-two bootloader, hypervisor, and virtual machine running the micro-kernel. The complete list of the CRs, MSRs, and their associated fields mentioned in this section can be found in Appendix B. However, several registers are particularly critical to realize the pseudo code in Figure 8 and are discussed here. The first of which is setting the general-purpose stack pointer (*rsp*) for the BSP (line 1).

When starting in 16-bit real mode, the first register field set by the BSP is bit 0 in CR0: the protected mode flag. Setting this field to 1 allows the BSP to enter 32-bit protected mode (line 5). Not setting the bit will result in a system crash as the BSP attempts to transition from 16-bit real mode to 32-bit protected mode.

Once, in 32-bit protected mode (line 6), the BSP configures CR4 (line 7) and the Extended Feature Enable Register (EFER) MSR (line 8) to enable support of 64-bit paging. CR4 bit 5 –the legacy Page Address Extension (PAE) [116] bit -- and bit 7 -- allowing page tables to utilize the global flag – are both be set to 1. Page Address Extension (PAE) is an enhancement to 32-bit paging to support virtual addressing above 4 GB; 64-bit paging, supporting up to 256 TB, which is used in this chapter, succeeded it. However, the PAE flag must be set, per x86-64 requirements, even if PAE is not being utilized. The global flag is a performance improvement associated with paging: Any page that is marked global cannot be evicted from the processor translation lookaside buffer. Utilizing global pages provides a speed up of critical operations, such as interrupt handlers. Next, the EFER MSR has bits 8 and 11 both set to 1: bit 8 enabling x86-64 long mode with 64-bit paging and bit 11 allowing the page table to take advantage of the No-Execute (NX) bit. Not setting bit 8 in the EFER MSR upon transition to 64-bit long mode has the same result as not setting bit 0 in CR0 when transitioning to 32-bit protected mode: it crashes the BSP. Lastly, the BSP enables paging through the MMU by writing 1 to bit 31 (Paging Enable) in Control Register 0 (CR0).

Upon entering 64-bit long mode the BSP makes a few final configurations to support floating point and Streaming SIMD Extension (SSE) instructions (line 16). This is accomplished by setting the monitor co-processor bit 1 and numeric error bit 5 in CR0 to 1. The co-processor bit enables the x86 *wait* and *fwait* instructions for handling floating point exceptions while the numeric error bit enables them. As a floating-point unit was not always present in early systems, the option to emulate them was present in many processors. However, with modern x86-64 systems that is not a problem and such the BSP must be told not to perform emulation by setting bit 2 in CR0 to 0. Lastly, the floating-point unit must be configured to allow and use SSE instructions. This is accomplished be enabling SSE exceptions through setting bit 10 to 1 in CR4 and enabling the use of fast floating-point unit switching by setting bit 9 to 1 in CR4. This is the last of the CPU register configurations made by the BSP

3.2.2 Physical Memory Profiling

Separate from setting registers is the detection of all the physical memory present on the system (line 2). Once found, it is used later by the stage-two bootloader to allocate a physical frame to a virtual memory page, which constitutes the basis of paging. Skipping this step means the hypervisor or micro-kernel cannot be loaded, no processes can be created with their own unique set of page tables, and SMP cannot be utilized as there is no micro-kernel or processes operating in their own memory space. Therefore, the industry standard of using BIOS interrupt 15 in 16-bit mode is used [98]. This call returns the layout of physical memory in the form of a flat memory model where the BSP could potentially directly access all of the installed physical memory [117]. When using x86-64 paging, the information returned from the call details if memory could be assigned to virtual address. The following commented GNU Assembly (GAS) loop shown in Figure 9 builds the physical memory map and implements line 2 in Figure 8

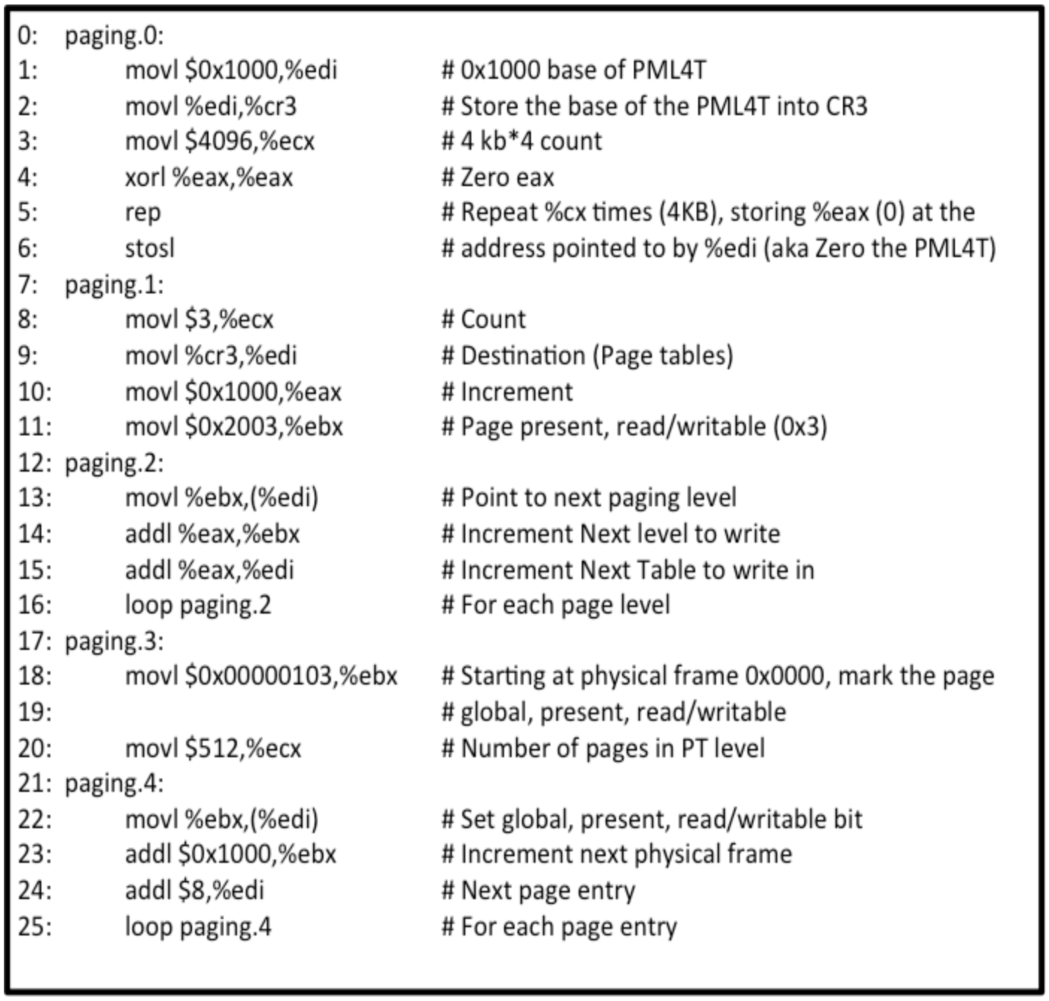
```
mmap.0:
    movw $MEMTABLE-24,%di        # Move start of memory table ($MEMTABLE: 0x804)
                                 # into di register
    xorl %ebx,%ebx               # Zero ebx register
    pushw %si                    # Save si register onto the stack
    movw %bx,%si                 # Zero si register
mmap:
    addw $24,%di                 # Set next map entry by adding 25 bytes to di
    movl $0xe820,%eax            # Get memory map using e820 method
    movl $24,%ecx                # Buffer size
    movl $EMAGIC,%edx            # e820 Magic Signature: 0x534D4150 loaded into edx
    int $0x15                    # Get memory map using BIOS interrupt 15
    jc mmap.1                    # Finished if carry flag is equal to 1
    cmpl $EMAGIC,%eax            # If e820 Magic Signature returned in eax then error
    jne error_sampling_memory    # jump out if error
    incw %si                     # Increment si to count total number of memory map entries
    cmpl $0x0,%ebx               # If ebx is equal to zero the memory profiling finished
    jne mmap                     # if ebx is not zero then resume loop at mmap
mmap.1:
    movw %si,MEM_ENTRIES         # move si to memory location 0x802 to store number
                                 # of memory table entries
    popw %si                     # Restore si register to previously saved value
    jmp next_routine             # Jump to next boot loader stage one routine
```

Figure 9 – Bios Memory Map Creation Assembly Code

After each interrupt call, data is loaded in back to back 192 bit blocks (Base Address, Length, Type, Compatibility Space). This data will later be accessed by the stage two bootloader to add pages to the initial page tables that are presented in the next section.

3.2.3 Initial Page Table Creation

The initial tables are created in the 32-bit protected mode portion of the stage-one bootloader (figure 8 lines 9 & 10) and must meet two important conditions. First, the tables must map in the soon to be virtual to physical memory address translations the stage-one and soon to be stage-two bootloaders have been loaded in by the MBR. Second, the page tables must also be placed in a physical memory location that is also covered by a virtual to physical mapping that is contained within them upon their initial creation. These two requirements are necessary as the BSP will cease operation on absolute memory, which it has done up until this point [5], and instead use the MMU to walk page tables to translate virtual to physical addresses when paging is enabled. Importantly, if the former requirement is not satisfied the BSP will crash and the later will result in the BSP being unable to modify and grow its initial page tables. These two requirements are satisfied with three assembly loops that create a one to one virtual to physical address mapping of the first 2MB of physical memory, otherwise known as an identity map. Due to the critical fact that the BSP cannot proceed to 64-bit operation and eventually SMP operation for the system as whole without these page tables; an explanation of the three assembly loops used to create them follows.



```
0:   paging.0:
1:          movl $0x1000,%edi           # 0x1000 base of PML4T
2:          movl %edi,%cr3              # Store the base of the PML4T into CR3
3:          movl $4096,%ecx             # 4 kb*4 count
4:          xorl %eax,%eax              # Zero eax
5:          rep                         # Repeat %cx times (4KB), storing %eax (0) at the
6:          stosl                       # address pointed to by %edi (aka Zero the PML4T)
7:   paging.1:
8:          movl $3,%ecx                # Count
9:          movl %cr3,%edi              # Destination (Page tables)
10:         movl $0x1000,%eax           # Increment
11:         movl $0x2003,%ebx           # Page present, read/writable (0x3)
12:  paging.2:
13:         movl %ebx,(%edi)            # Point to next paging level
14:         addl %eax,%ebx              # Increment Next level to write
15:         addl %eax,%edi              # Increment Next Table to write in
16:         loop paging.2               # For each page level
17:  paging.3:
18:         movl $0x00000103,%ebx       # Starting at physical frame 0x0000, mark the page
19:                                     # global, present, read/writable
20:         movl $512,%ecx              # Number of pages in PT level
21:  paging.4:
22:         movl %ebx,(%edi)            # Set global, present, read/writable bit
23:         addl $0x1000,%ebx           # Increment next physical frame
24:         addl $8,%edi                # Next page entry
25:         loop paging.4               # For each page entry
```

Figure 10 - Initial Page Table Creation Assembly Code

Before the first loop in Figure 10 above, the code begins by loading the base physical address of the Page Map Level 4 Table (PML4T) into the CR3 register (lines: 1-2). Next, the first loop ensures the entire 4KB frame of the PML4T is set to zero by using the *rep* and *stosl* assembly instructions, which writes 0s across the 4KB physical address space (lines: 3-6). Initialization of the PML4T is completed by the next loop.

As there are four levels of paging structures and the PML4T was previously created; the second loop takes care of creating the Page-Directory-Pointer Table (PDPT), Page Directory Table (PDT), the Page Table (PT) and configuring the top three levels (lines: 7-16). The loop will iterate three times (line: 8), starting at the address of the PML4T (line: 9), increment by 4KB size (line: 10), and create an entry in each of the top three level tables (lines: 11-13). Each entry has the read/write, global, and present bits set as seen in Figure 11. The PT, which contains page frame entries only, is configured in the third loop.

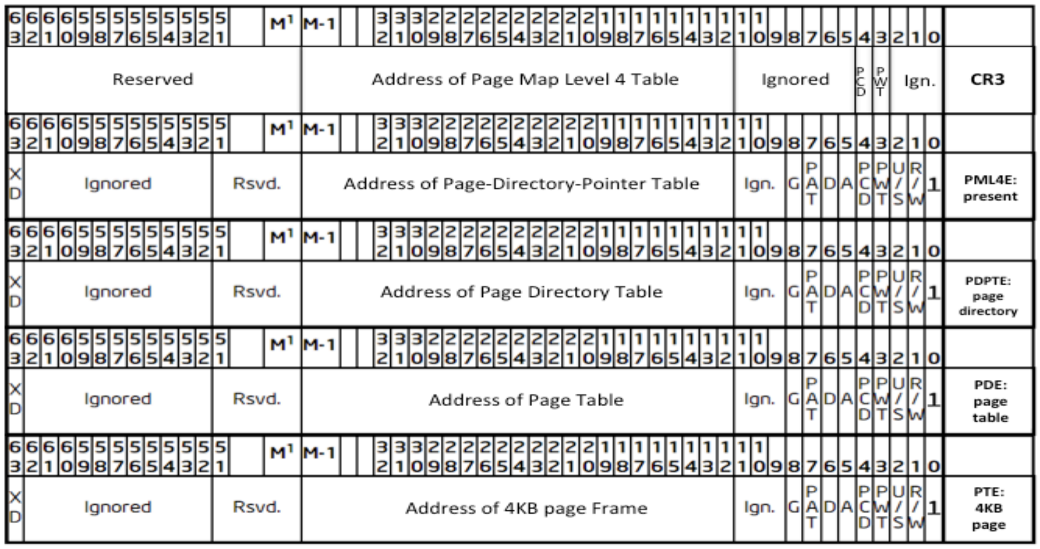


Figure 11 - Four Level Page Table Layout (Intel Manual)

The PT only contains page frames and not table entries, which is the reason it is initialized in a third separate loop (lines 17-25). To meet the two previous requirements mentioned earlier, the PT starts mapping from physical address 0x0000 (line: 18). Also, note that all of the entries in the PT are marked read/write, present (lines: 22-25), and global. All 512 entries of the PT will be initialized by this loop (line 20), which means the memory from 0x0000 to 0x200000 is mapped (2MB). Furthermore, this is an identity map of virtual to physical memory.

Once all three loops have run, a paging system as seen in Figure 12 below is created. It can be seen that the PML4T has one entry that maps to a PDPT, the PDPT has one entry that maps to a PDT, the PDT has an entry that maps to a PT, and lastly the PT has 512 entries that identity maps 0MB to 2MB of memory. Additionally, the two requirements set early in the initial page table discussion are met, as the bootloader and the initial page tables exist entirely between the ranges of 0MB-2MB.

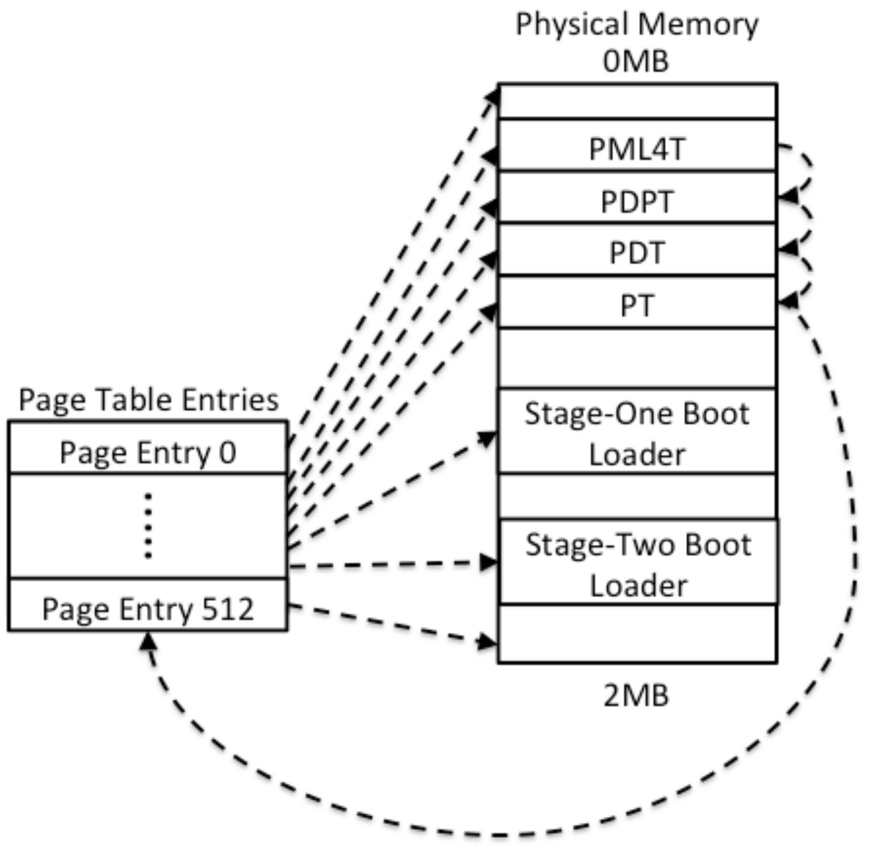


Figure 12 - Initial Page Table Memory Layout and Mapping

The last act of paging and discussion on the stage-one bootloader is to create a recursive entry inside of the initial tables PML4T (figure: 8, line: 11). As the memory manager created by the stage-two bootloader will make primary use of this entry, explanation on it is saved for the next section.

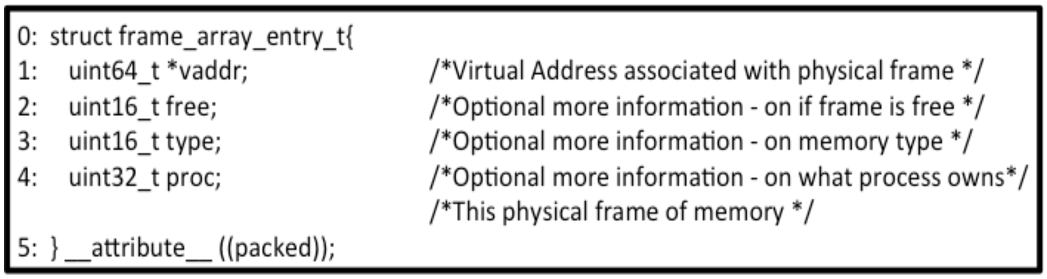
3.3 Stage-Two Bootloader and Memory Manager

The stage-two bootloader marks the transition to a full 64-bit long mode and the transition to using C code for the creation and management of complex systems. The end goal of which is to have a method in place that effectively ties available physical memory to virtual memory, and to lay the groundwork for SMP to make use of the memory system to provide each process with its own memory space. Without an effective memory manager, SMP cannot be utilized by the system as a whole. The memory manager presented here manages physical addresses as 4Kb frames of physical memory through an array of frames. The virtual memory component is managed through a technique known as recursive paging. These are the physical and virtual memory management methods used in this system, but are not the only means of doing so. Thus, before embarking on a solution for memory management, it is critical to thoroughly weigh the pros and cons of this or any other system that is chosen.

3.3.1 An Array of Structures

The job of the memory manager is to maintain mappings from physical to virtual memory addresses. When new virtual addresses need to be mapped to underlying physical address frames, the system needs to know which physical address frames are in use and which are free. In this implementation, a frame array maintains this information. The frame array itself is comprised of an array of structures.

Conventionally, a bitmap is used to maintain this information. A bitmap has a few advantages, including a low memory footprint and constant time to look up a given frame [118]. In the case of 4KB page size, bitmaps use 1 bit to represent 32,768 bits of memory, which accounts for an overhead of 0.003%. However, this method also has a major drawback. With a bitmap, the only information that is stored about a given frame is whether it is free or not. An alternative is to encode the information about the frames in an array of structures, with each structure representing one frame on the system. The C code for this structure is seen in Figure 13.



```
struct frame_array_entry_t{
    uint64_t *vaddr;          /*Virtual Address associated with physical frame */
    uint16_t free;            /*Optional more information - on if frame is free */
    uint16_t type;            /*Optional more information - on memory type */
    uint32_t proc;            /*Optional more information - on what process owns*/
                              /*This physical frame of memory */
} __attribute__ ((packed));
```

Figure 13 - C Frame Array Structure

This method maintains the benefit of constant lookup time for a given frame, and adds the capacity to store more information about each frame. For example, a constant time *phys2virt* physical to virtual address translation can be achieved by storing the virtual address in the corresponding structure for the physical frame to which that virtual address maps. In addition, it is possible to store the process identifier to which a frame has been mapped in the frame structure, allowing a comparison between the frame array and the page tables of a given process. This allows the micro-kernel to search for inconsistencies that would indicate some form of memory corruption.

The cost of the additional information, of course, is a larger memory footprint. If n is the number of bytes available on the system, P is the size of a page, and s is the number of bytes stored in the frame array *per frame*, the frame array will occupy $\frac{n}{p}s$ bytes: $\left\lceil\frac{ns}{P^2}\right\rceil$ pages. In terms of the overall physical memory (which has

$\frac{n}{p}$ total frames available) this corresponds to a fraction of $\frac{s}{p}$ of main memory dedicated to bookkeeping. As shown in Table 2, for typical values of P, this percentage is still small, even with generous sizes for s.

S	P	Overhead
16 bytes	4096 bytes	0.4%
8 bytes	4096 bytes	0.2%

Table 2 - Overhead Cost of Frame Array

Of course, the frame array is just like any other data on the system. It is stored on physical memory and accessed by virtual addresses. Unfortunately, since the frame array is the foundation of the virtual memory manager, initializing the frame array cannot use typical virtual memory management. This makes initializing the frame array a bit like changing a bike tire while riding the bike, which is only complicated by the need to grow the initial page tables to map the frame array.

3.3.2 Growing the Initial Page Tables Through Recursion

To grow the size of paged memory to accommodate the frame array, the operating system must write the physical address of some new unused physical frame of memory into a PT entry corresponding to the virtual address that the frame array is located at. The question is: how does the system write to that page table?

The PT is an arbitrary 4k physical frame. In order to write to a location in memory, even a PT, software provides a virtual address that is translated through a PML4T, PDPT, PDT, and PT until finally the address whose page-size base is stored in that specific Page Table entry is written. Therein lies the recursive problem of memory management. In order to write a PT, there must be a PT entry already written. The same is true at every level of paging structures. Naively, writing a set of paging structures before virtual memory is “turned on” by activating the MMU can solve this as seen in section *2.0.3*. This provides a “base case” for the recursive problem that has been stumbled upon. In fact, simply growing this initial table does this very trick. However, this method leads to a fair share of problems. To illustrate this, imagine the following scenario: the system wants to set up a large and contiguous virtual memory region, for example the user’s heap. Ideally, the micro-kernel could simply walk along an array of physical frames, writing the address of free frames into a PT. When a PT runs out, the micro-kernel needs to write the address of the next physical frame into a PDT to serve as a new PT. However, with this method, that physical address must also be written into some PT entry. Which one? Answering that question is not trivial, and encourages dangerous solutions such as hard-coding the address of paging structures.

Instead, a more streamlined approach known as recursive paging or self-referencing page tables can be used. Recall again that each of these paging structures is simply a 4k frame of physical memory. In other words, the only thing that makes a PT any different from a random block of memory is that its address is stored in a PDT entry. Similarly, a PDT is defined only by its presence in a PDPT entry, and a PDPT only by its presence in a PML4T entry. Finally, a

PML4T is differentiated from a random block of memory only by the fact that its address is stored in the CR3 register. Recursive paging proposes the following idea: consider what would happen if the base address of the PML4T (the same one found in the CR3 register) is stored in the PML4T itself. Essentially, this is an entry in the PML4T that points back to the base of the PML4T. However, a PDPT is defined only as something whose physical address is stored in a PML4T. Now, the CR3 target is both a PML4T and a PDPT. Furthermore, a PDT is only something whose physical address is stored in a PDPT. Since the CR3 target is a PML4T and a PDPT, and it contains the physical address stored in the CR3 target, it can be deduced that the CR3 target is a PML4T and a PDPT and a PDT. Following a similar logic shows that the CR3 target itself can be essentially "cast" as any of the levels in the page table walk, including a PT and a physical frame.

The capacity to "cast" the CR3 target as any level of the paging walk means that one can manipulate the virtual address to cause the MMU to "loop" over the self-referencing pointer during one or more steps of its walk. For example, it is possible to "cast" the CR3 target as the 4k physical frame itself by setting the indices in bits 47-39, 38-30, 29-21, and 20-12 Figure 14 ALL to the index of the self-referencing entry of the PML4T. This allows the use the offset stored in bits 11-0 to write to arbitrary locations in the PML4T. Similarly, it is also possible to "cast" the CR3 target as the PT by setting the indices in bits 47-39, 38-30, and 29-21 all to the index of the self-referencing entry of the PML4T. Then, the index in bits 20-12 (the "Page Table Offset") can be used to point to a given PDPT, which will then be indexed by bits 11-0, allowing us to write to arbitrary addresses within a PDPT.

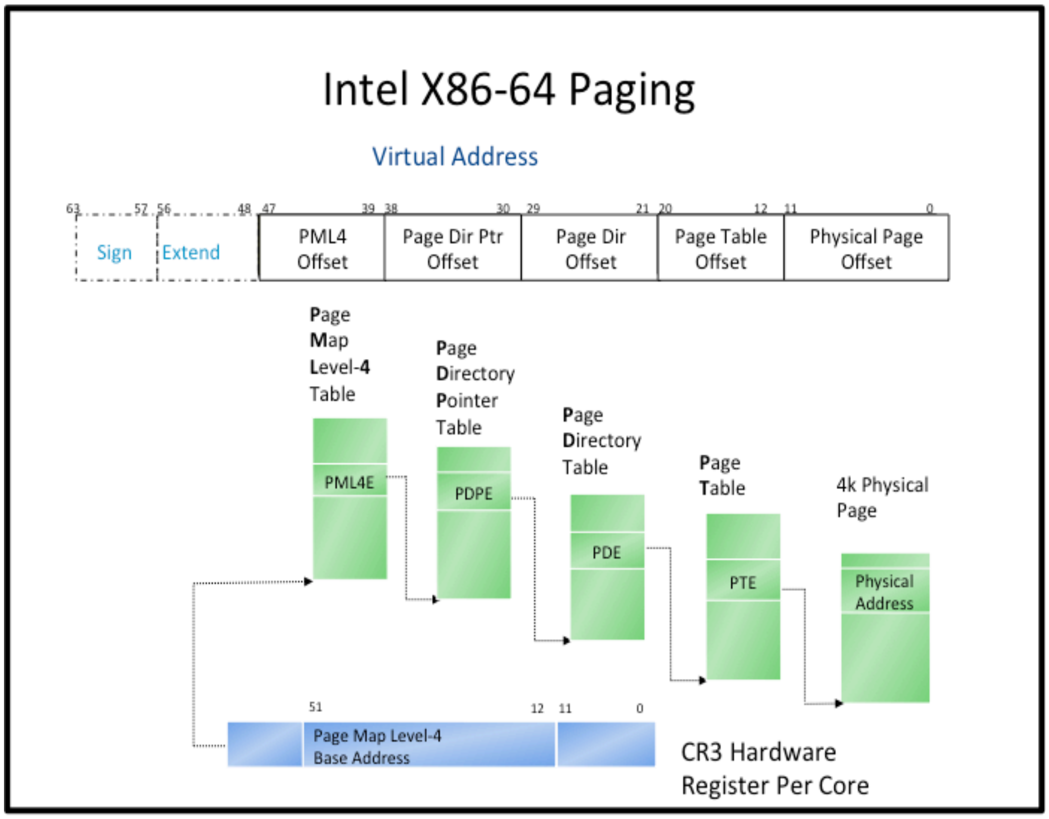
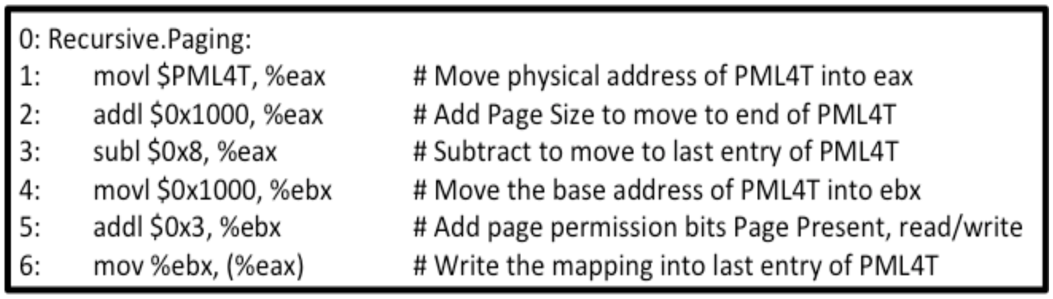


Figure 14 - X86-64 Virtualization (Intel Manual)

In other words, the self-referencing entry in the top-level paging structure allows automatic access to every paging structure linked to the currently active CR3 target. The physical address of a PDPT does not need to be stored in a PT entry in order to write to it. Instead, the MMU can be “tricked” by “looping” through the self-referencing pointer, so that it treats the PML4T itself (which already stores the physical addresses of all active PDPTs) as the PT.

This solution is elegant and has many attractive features. In the case examined before, paging in memory for the user heap, the system can use any arbitrary frame for the new PT when it is needed. It still needs to write the physical address of that frame into a PDT entry, but it no longer needs to write the address in a second place: some other PT entry. Instead, subsequent attempts to write to that PT will "cast" the PDT as a PT, giving a valid virtual mapping to the new PT.

The stage-one bootloader (figure 8 line 11) has provided the recursive page-table mapping in six lines of assembly code seen in Figure 15. Effectively, the last of the 512 entries in the PML4T (lines: 0-3) holds the base address of the PML4T itself (lines: 4-6).



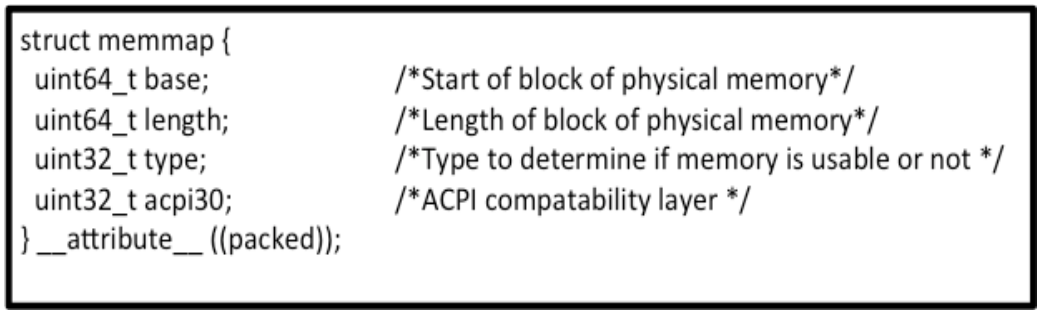
```
0: Recursive.Paging:
1:      movl $PML4T, %eax         # Move physical address of PML4T into eax
2:      addl $0x1000, %eax        # Add Page Size to move to end of PML4T
3:      subl $0x8, %eax           # Subtract to move to last entry of PML4T
4:      movl $0x1000, %ebx        # Move the base address of PML4T into ebx
5:      addl $0x3, %ebx           # Add page permission bits Page Present, read/write
6:      mov %ebx, (%eax)          # Write the mapping into last entry of PML4T
```

Figure 15 - Recursive Pointer Page Table Entry Assembly Code

This recursive entry now allows the system to walk any level of the page tables using the C Macros seen in Appendix C.

3.3.3 Mapping and Populating the Frame Array

First, the frame array initialization function in Appendix D must calculate the total memory present on the system. This information is provided in the memory map generated during 16-bit real mode, as described in section *3.2.2* of this chapter and is accessible thanks to the initial page tables created in section *3.2.3*. The last entry in the memory map can be found by reading the value previously stored at 0x802. To offset correctly into the memory map stored starting at 0x804, the value is multiplied by the size of the structure seen in Figure 16.



```
struct memmap {
  uint64_t base;              /*Start of block of physical memory*/
  uint64_t length;            /*Length of block of physical memory*/
  uint32_t type;              /*Type to determine if memory is usable or not */
  uint32_t acpi30;            /*ACPI compatability layer */
} __attribute__ ((packed));
```

Figure 16 - C Bios Interrupt 15 Memory Map Structure

By taking the last entry in the memory map and adding the length to the base, it is possible to calculate how many frames will be needed to store the frame array. Subsequently, it also possible to calculate how many paging structures (PTs, PDTs, & PDPTs) will be needed to address the frame array. The sum of these two amounts is the total number of frames needed to initialize the frame array. This number is used to find a block from the BIOS provided memory map that is big enough to hold the needed frames.

The existence of the initial page tables makes it possible to start at the base address of the found block of memory and increment by 4KB frames to grow them. This is similar to the initial page tables that were created, except that any virtual address may be chosen to map the frame array and the recursive pointer is used to map new tables themselves. The major benefit from the recursive pointer is that as page tables are exhausted, the next free frame can be used for the next paging structure. This has the effect of producing a continuous block of virtual memory for the frame array even if it is not physically contiguous. This is best illustrated in Figure 17, which shows the completed virtual mapping of an example frame array and its underlying physical memory.

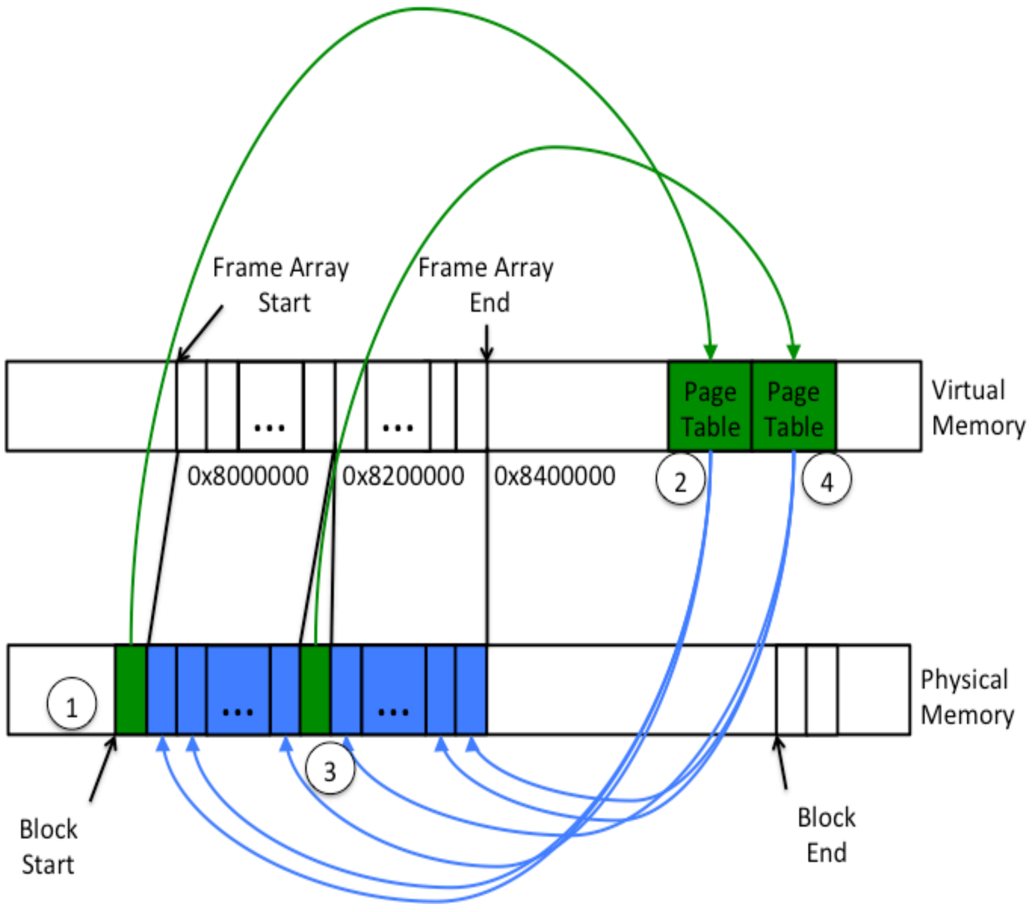


Figure 17 - Virtual Mapping of the Frame Array via Recursive Pointer

This example frame array covers 4MB of virtual memory from address 0x8000000 to 0x8400000. Circle 1 shows what happens first at the start of the found physical memory block used to map the frame array. At this point the first frame is used to create a PT that is added to the initial page tables through the recursive pointer. Circle 2 shows that this PT then maps the first virtual 2MB of the frame array. At that point, all of the entries in the PT are exhausted. Thus, the next free frame is then used in circle 3 for adding another PT to the initial page tables through the recursive pointer. The PT in circle 4 then maps the final virtual 2MB of the frame array. Thus, the frame array is mapped contiguously in virtual memory, but not contiguously in physical memory, as the recursive pointer provides on demand PTs as needed.

Once the frame array is mapped, the initialization routine needs to populate the array so that it reflects the current state of the system. First, it looks through the BIOS memory map and uses the information about the blocks to populate the sections in the array that were reserved by the BIOS. Additionally, the system needs to walk its own page tables to mark the frames that have already been used in creating paging structures up to this point. These frames will be marked not only as taken, but the virtual address corresponding to the frame will be stored in the appropriate structure in the array. Note that this page table walk, vitally, includes walking through the recursive entry in the PML4T. This allows for all frames that are used for the paging structures themselves to be accounted for. With the frame array mapped and fully populated, the systems memory manager may use it with the recursive pointer to create new or grow existing page tables. The creation of new tables is a process management question, which this chapter does not address, because just as there are multiple options for memory management, there are an even greater number of options for process management.

This implementation solves the previous problem of running multiple processes concurrently, as the frame array and recursive pointer can leverage the inherent hardware capabilities of the processor to create different sets of page tables for each process to access memory. The processor cores support this through multiple CR3 targets. Recall the CR3 tells the memory manager where the start of the page tables exists. From there the MMU translates the virtual address to physical memory frames using the tables loaded into the CR3, hiding the implementation details from the user application developer. The micro-kernel maintains the pointer to this CR3 target for each process and simply updates the register inside the core upon process load. SMP is supported as each processor contains a unique CR3, but the only processer running on the system as of now is the BSP.

3.4 Hypervisor

The hypervisor itself is only run by the BSP after being loaded by the stage-two bootloader. The hypervisor will run the SMP micro-kernel as a virtual machine above it as presented in chapter one. Prior to this however, the hypervisor must configure its interrupt controller as well as find, boot, and configure the APs.

3.4.1 Utilizing the Application Processor Cores

The main driver of an x86 processor and arguably the most important part, whether it is scheduling, handling unplanned events, or responding to user input, is the interrupt system [119]. The first of these interrupt controller chips, the Intel 8259 Programmable Interrupt Controller (PIC) was introduced to be compatible with the Intel 8086 processors [111]. The PIC however was not designed to be used in a SMP architecture. This coupled with the fact that the PIC used Port-Mapped Input/Output (PMIO), which is slower than the newer standard of MMIO meant a new hardware interrupt controller had to be implemented.

The solution introduced by Intel came in the form of two chips. The Intel 82489DX or better known as the Advanced Programmable Interrupt Controller (APIC) [120], replaced the software interrupt functionality provided by the PIC. The APIC uses MMIO [5] to provide timing, exception handling, and software interrupts to the processor. Additionally, the APIC also provides the brand new feature of Inter-Processor Interrupts (IPIs), where one processor core can send a software interrupt to another or all processor cores. These IPIs were made possible by providing each processor core with its own dedicated local APIC. Thus, enabling each core to handle interrupts independently of any other core. The other chip, the Intel 82093AA, better known as the Input/Output Advanced Interrupt Controller (I/O APIC), would become the new interface between hardware generated interrupts and the processor.

A whole new paradigm of programming the interrupt system was introduced, with the introduction of the split APIC and I/O APIC using MMIO. This can be seen as both chips are no longer at a constant location as was with the PMIO based PIC. However, the added benefits far outweigh this additional burden. The presence of multiple APICs allows a developer to enumerate the number of cores on the system while being able to perform SMP related tasks that were simply not possible with the PIC architecture. The process of finding and using the APIC and I/O APIC can be distilled into the five steps seen in Figure 18 below

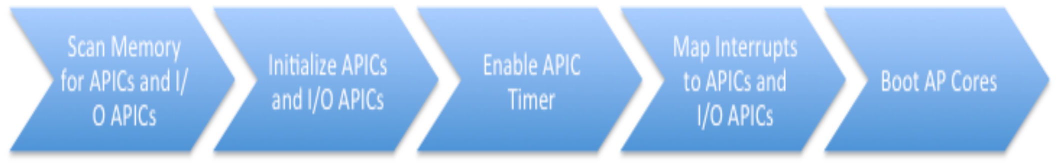

Scan Memory for APICs and I/O APICs
Initialize APICs and I/O APICs
Enable APIC Timer
Map Interrupts to APICs and I/O APICs
Boot AP Cores

Figure 18 - Process for Booting Application Cores

3.4.2 Finding the Application Processor Cores

The first attempt to introduce APICs, I/O APICs, and multiple processor cores to the x86 world can be seen in the of how they are found in physical memory. The heavy lifting of identifying present APICs and I/OAPICs is done by the BIOS prior to turning the system over to the bootloader. The challenge here becomes digging through the information stored haphazardly by the BIOS in memory and the associated standards and documentation that describe how to read this data.

3.4.2.1 Advanced Configuration Power Interface Table

Hewlett-Packard, Intel, Microsoft, Phoenix Technologies, and Toshiba developed ACPI in 1996 to provide an industry standard for an ever growing collection of BIOS code, power management, multiprocessor support, and many other system hardware interfaces [98]. Though the ACPI specification covers all manners of system hardware, the piece needed for SMP is known as the Multiple APIC Descriptor Table (MADT). The steps for finding the MADT are outlined in Figure 19.

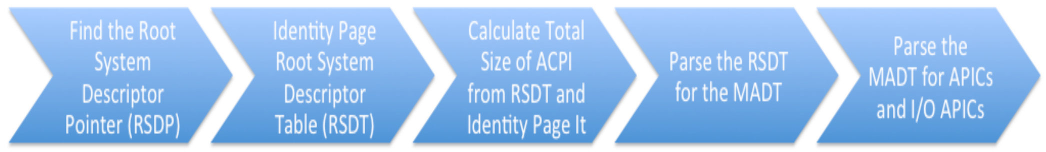
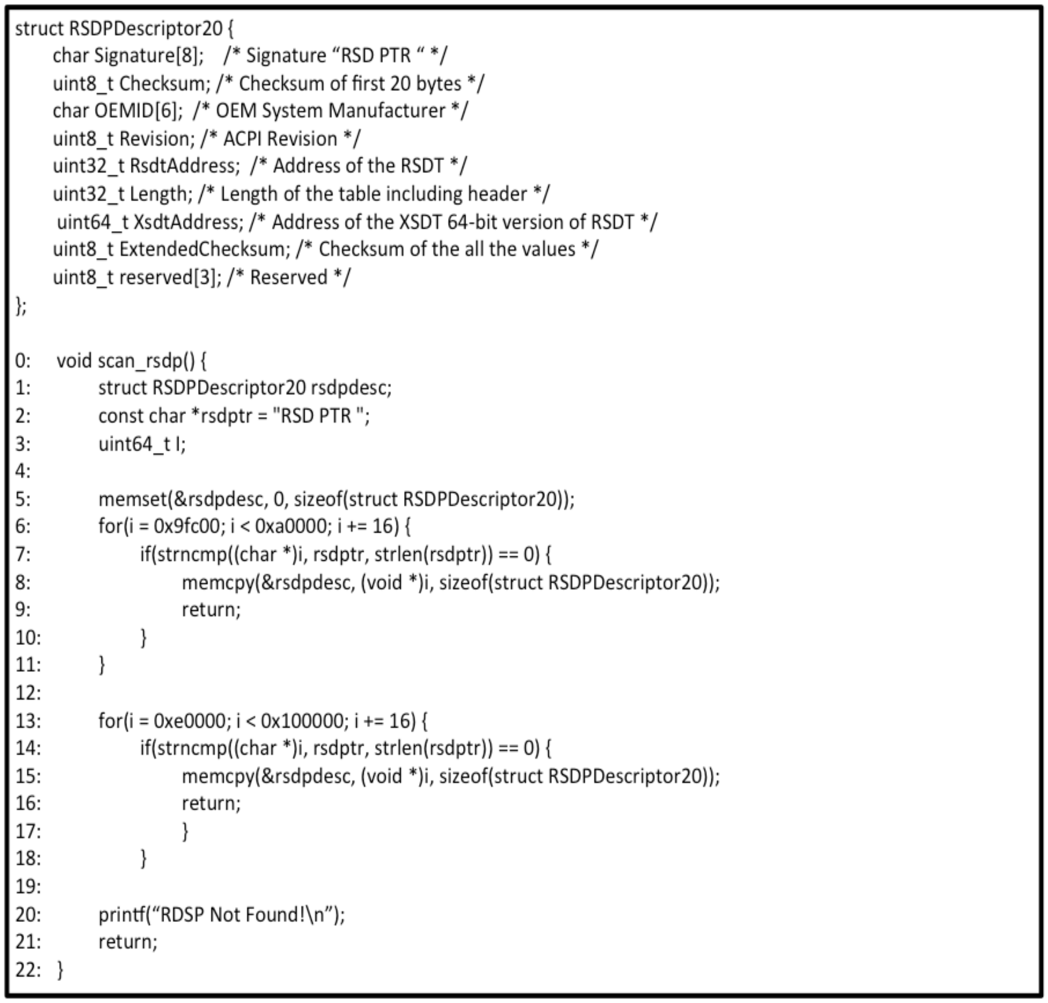

Find the Root System Descriptor Pointer (RSDP)
Identity Page Root System Descriptor Table (RSDT)
Calculate Total Size of ACPI from RSDT and Identity Page It
Parse the RSDT for the MADT
Parse the MADT for APICs and I/O APICs

Figure 19 - Process for Finding and Parsing ACPI Tables

The first step is to find a pointer stored in memory by the BIOS. As per the ACPI specification at one of the following two locations:

- Physical memory 0x9FC00 to 0x13FC00
- Physical memory 0xE0000 to 0x1E0000

The search signature has is “RSD PTR ”, which it is important to note the trailing white space. The C code and the structure of the RSDT to accomplish this search are given in Figure 20 below.



```
struct RSDPDescriptor20 {
    char Signature[8];    /* Signature "RSD PTR " */
    uint8_t Checksum; /* Checksum of first 20 bytes */
    char OEMID[6];  /* OEM System Manufacturer */
    uint8_t Revision; /* ACPI Revision */
    uint32_t RsdtAddress;  /* Address of the RSDT */
    uint32_t Length; /* Length of the table including header */
     uint64_t XsdtAddress; /* Address of the XSDT 64-bit version of RSDT */
    uint8_t ExtendedChecksum; /* Checksum of the all the values */
    uint8_t reserved[3]; /* Reserved */
};

void scan_rsdp() {
        struct RSDPDescriptor20 rsdpdesc;
        const char *rsdptr = "RSD PTR ";
        uint64_t I;

        memset(&rsdpdesc, 0, sizeof(struct RSDPDescriptor20));
        for(i = 0x9fc00; i < 0xa0000; i += 16) {
                if(strncmp((char *)i, rsdptr, strlen(rsdptr)) == 0) {
                        memcpy(&rsdpdesc, (void *)i, sizeof(struct RSDPDescriptor20));
                        return;
                }
        }

        for(i = 0xe0000; i < 0x100000; i += 16) {
                if(strncmp((char *)i, rsdptr, strlen(rsdptr)) == 0) {
                        memcpy(&rsdpdesc, (void *)i, sizeof(struct RSDPDescriptor20));
                        return;
                        }
                }

        printf("RDSP Not Found!\n");
        return;
}
```

Figure 20 - Remote System Descriptor Pointer (RSDP) and Search Code

With the RDST found, it is now possible to parse the ACPI tables as a whole. The ACPI tables start at the RSDT, which is comprised of multiple entries known as ACPI headers, which itself contains a pointer to sub-tables. These sub-tables are expanded ACPI headers that may either consist of more tables or a variable number of data entries. For the case of MADT, it consists of just a variable number of entries that describe the number of APICs, I/O APICs, and Interrupt Overrides present on the system. The layout in memory if the ACPI tables looks as follows.

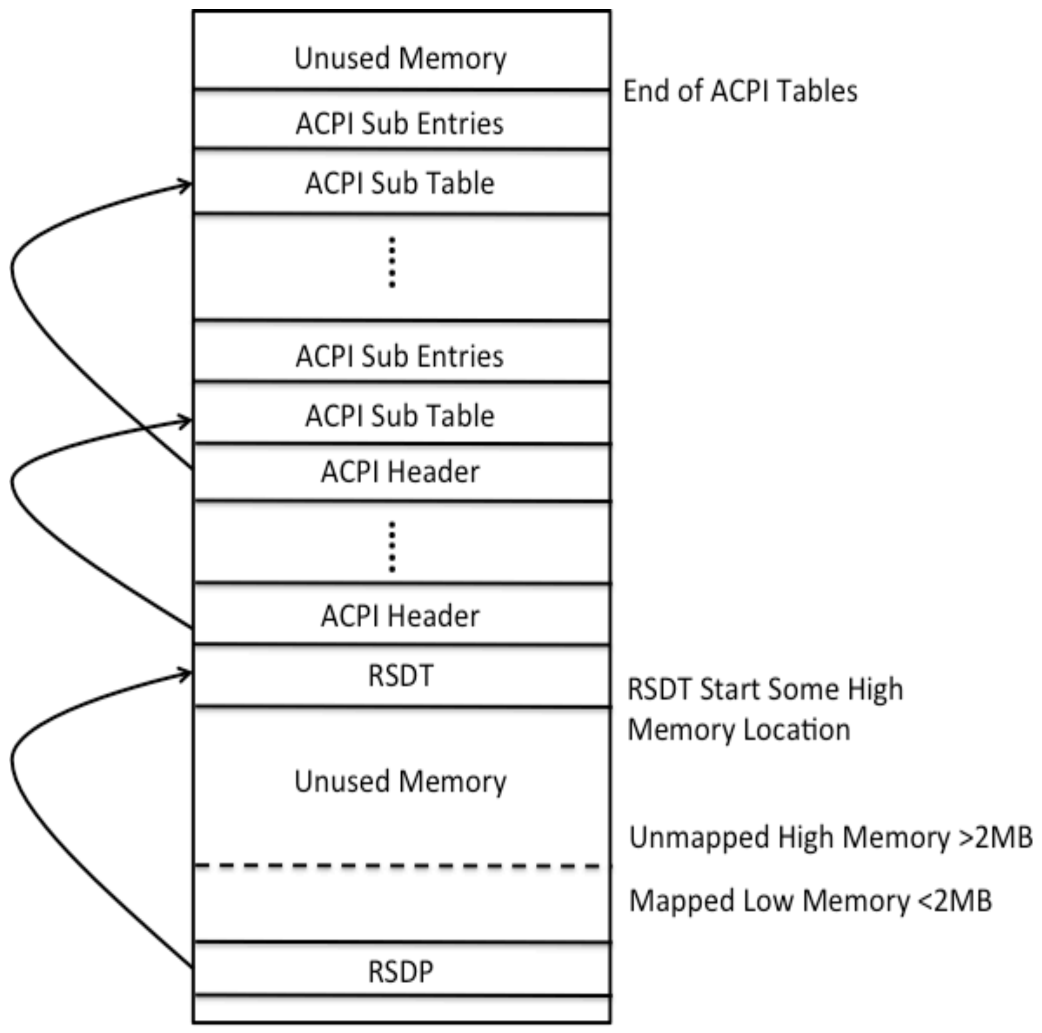


Figure 21 - ACPI Layout in Memory

From this diagram two challenges of parsing the ACPI tables can be seen. First, and one reason why a memory manager was built, is that the ACPI tables fall outside of the initial 2MB of memory mapped by the identity tables created by the bootloader. Second, from the RSDP at least three pointers must be followed across memory to eventually find and parse the MADT, which is an ACPI sub table.

The processes of growing the initial page tables to map the ACPI tables as well as parsing them are inherently intertwined. The information contained in the RSDP allows the RSDT to be mapped and the information contained in the RSDT is used to map the rest of ACPI space. The code contained in Appendix E first maps the RSDT and then calculates the size of the entirety of the ACPI space. As the BIOS often will place ACPI tables across page boundaries, the end of the ACPI tables is calculated to be two 4KB pages past the point of the address calculated from the RSDT. Thus a while loop is used to identity map the start to end of the ACPI region.

Once completed, a second while loop iterates over all of the ACPI header entries contained within the RSDT. For each entry it is determined what table that specific ACPI header is pointing to. In this case the signature "APIC" represents the MADT and when found must be parsed to obtain information regarding to the APICs and I/O APICs on the system.

The MADT consists of an expanded ACPI header, which contains the MMIO address of the APIC. From there it consists of a variable number of APIC, I/O APIC, and Interrupt Override entries. Thus, one last while loop parses this data. The loop starts past the expanded header entry and searches for each sub-entry type based on the device ID type. When this loop finishes, the needed information pertaining to enabling SMP on the system has been obtained.

3.4.3 SMP configuration for the Bootstrap Processor

With the BSP and hypervisor armed with the ability to manipulate paging and the knowledge of APICs and I/O APICs on the system, the hypervisor may now begin to take the steps to configure its own APIC and the system I/O APIC. The former is necessary to boot other cores and provide software interrupts while the latter is necessary to provide hardware interrupts.

3.4.3.1 Enabling the APIC

The APIC is enabled by reading/writing to the APIC MMIO register address that was found previously in the ACPI tables. Notable APIC register fields and values that can be written are provided in the Intel Software Developer's Manual [5]. Notable APIC register fields and values that can be written are provided in Appendix F. The APIC read/write C functions and code to enable the APIC are as follows.

```
0:  uint32_t lapic_read(uint32_t offset){       /* offset is a field in the APIC MMIO Register */
1:        return *(uint32_t *)(lapicaddr+offset);   /* Return value read from the APIC MMIO Register+offset */
2:  }
3:
4:  /* offset it the field in the APIC register to write to and data is the information to write */
5:  void lapic_write(uint32_t offset, uint32_t data){
6:        *(uint32_t *)(lapicaddr+offset) = data;         /* Write data to APIC MMIO Register+offset */
7:  return;
8:  }
9:
10: void lapic_init(){
11:
12:       lapic_write(APIC_DFR,0xFFFFFFFF);      /* Set destination format register to flat model */
13:       lapic_write(APIC_LDR,(lapic_read(APIC_LDR)&0x00FFFFFF)|1); /* Set local destination register to flat model */
14:       lapic_write(APIC_LVT_TMR,APIC_DISABLE);         /* Disable APIC Timer for Now */
15:       lapic_write(APIC_LVT_PERF,APIC_NMI);             /* APIC Performance Counter Overflow Non-Maskable */
15:       lapic_write(APIC_LVT_LINT0,APIC_DISABLE);       /* Disable local Interrupt 0 */
17:       lapic_write(APIC_LVT_LINT1,APIC_DISABLE);       /* Disable local Interrupt 1 */
18:
19:       /*turn on the APIC and route spurious interrupts to a black hole */
20:       lapic_write(APIC_TASKPRIOR,0);
21:       lapic_write(APIC_SPURIOUS,63|APIC_SW_ENABLE);
22:
23:       return;
24: }
```

Figure 22 - Code to Enable Advanced Programmable Interrupt Controller

The code above assumes the address of the APIC MMIO register address has been identity mapped by the memory manager. This address is represented by the value *lapicaddr* (Lines: 1 & 6), which typically defaults to 0xFEE00000, but should be compared to the address found either in the MP or ACPI tables. The *lapic_init* function enables the APIC to a known good state for handling interrupts.

The base state is initialized by first writing the Destination Format Register (DFR) (Line: 12) and Logical Destination Register (LDR) (Line: 13) so that the APIC delivery mode is set to flat. This means that when sending IPIs between processor cores, the DFR is first read to see that a flat model is in use by ensuring bits 28 through 21 are set to 1111. Then the local APIC ID programmed into bits 24 through 27 of the LDR is compared on each core to the APIC ID sent in the IPI via a bitwise *AND*. If true, that local APIC accepts the IPI. Bits 28 through 31 of the LDR are used for cluster IPIs where more than one core can be sent the same IPI. As the IPIs in this chapter are directed to a single core, these bits are set to zero to signify that cluster IPIs are not in use.

The local interrupt vector registers are then addressed (Lines: 14-17). These vectors determine if an interrupt is generated by an APIC if certain conditions are met. As the APIC, which will be used later for scheduling, has not been calibrated, it is disabled temporarily. The Performance Counter interrupt is set to non-maskable, which means that if performance counters are in use, they will generate an interrupt upon overflow. Lastly, both the local Interrupt 0 & 1 pins are disabled. These pins are used to chain legacy devices to the APIC and are not needed.

All that is left is to turn the APIC on, so that it starts receiving and generating software interrupts. This is done by setting the Task Priority Register to 0 (Line: 20) and ensuring spurious interrupts generated by the APIC are routed to an unused interrupt line (Line: 21).

3.4.3.2 Configuring the APIC Timer

The calibration of the APIC timer is accomplished by counting the number of times it fires over a known time quantum. This is in done in conjunction with the

Time Stamp Counter (TSC) [121], which counts the number of cycles executed by a core since it was last reset. The time quantum is based on the frequency of the processor core. The frequency is determined by three MSRs, which are read to see if Dynamic Acceleration or Turbo Boost is enabled, and the ratio that the core is running at [5]. Ratio is referred to by the Intel clock speed definition “ratio*100MHz”, which means a ratio of 10 results in a processor core running at 1GHz. To find what ratio is used, the *platform_info* MSR is first checked to see if bit 31 set to 1. If it is, Dynamic Acceleration is enabled, which means the ratio can be found in bits 40-44 of the *performance_status* MSR. If the bit was not set, then the maximum ratio is found in bits 8-15 of the *platform_info* MSR. Next, bit 16 in the *flex_ratio* MSR is read to see if it is set. With a value of 1 indicating that Turbo Boost is active and the ratio reported by the *flex_ratio* MSR should be used to calculate core speed. If the *flex_ratio* MSR does not report a ratio, then the default maximum ratio reported by the *platform_info* MSR should be used to calculate core speed. If Turbo Boost is not active according to bit 16 in the *flex_ratio* MSR, then the maximum ratio reported by the *platform_info* MSR is used. The code for this process can be found in Appendix G.

The value determined from the above ratios will be used to determine how fast process switching occurs and has to be handled delicately. For example, if the timer operates at the same frequency as the core (*freq/Large_Divisor*) the system will never execute user space code as a timer interrupt will occur almost the same time as the user process starts running. The opposite is also true, if this frequency is too slow (*freq/Small_Divisor*), the system will fire timer interrupts at an extremely slow rate, which will result in one user process hogging all of the processor cycles. In testing, a divisor of 100 has proven optimal for calibrating the APIC Timer on a 3.4 GHz i7 Intel processor with Bear's software load, which includes a hypervisor, micro-kernel, a number of user processes, and user space drivers. What this means is if the processor operates at 3.4 billion cycles per second (3.4 GHz), then a timer will fire roughly once every 34 million cycles.

To perform this calibration, the APIC Time Current Register is set to a very large number that will be decremented as the timer fires at a periodic interval. Two initial values are then sampled from both the TSC and APIC Time Current Register prior to entering a do while loop, which then continues to sample both of these timers. The loop will run until the TSC reports a difference between the initial and current TSC value greater than the desired frequency of the APIC Timer (*frequency/100)*. Once, the loop is exited the difference in count between initial APIC time value and the last APIC time value can be programmed into the APIC Timer. The APIC Timer will then generate an interrupt every time this count reaches zero and the count will begin again after the interrupt is handled by the either the hypervisor or micro-kernel. This APIC generated interrupt is the basis for time slice scheduling used by the micro-kernel. The code to perform this calibration can be seen below.

```c
void calibrate_apic_timer(){
        unsigned apic_current, apic_start;
        uint64_t tsc_hz, tsc_current, tsc_start, HZ;

        tsc_hz = get_tsc_freq();

        /* This is the frequency the timer will fire, which sets up how fast processes switch  */
        HZ  = tsc_hz / 100;

        /* place a long value in the APIC timer so it starts counting we can then calibrate it  */
        /* against the tsc known frequency. Note the APIC timer counts down                      */
        lapic_write(APIC_TMRINITCNT, 4000000000/APIC_TDIV1);
        /* set as periodic and the local vector to 32 */
        lapic_write(APIC_LVT_TMR, 32 | TMR_PERIODIC);
        /*set the divisor for the APIC to 1*/
        lapic_write(APIC_TMRDIV, APIC_TDIV1);

        /*read the intial values */
        apic_start = lapic_read(APIC_TMRCURRCNT);
        tsc_start = readtsc();

        /*Loop for the known quantum of time based on the tsc freq */
        do{
              tsc_current = readtsc();
              apic_current = lapic_read(APIC_TMRCURRCNT);
        }while ((tsc_current - tsc_start) < HZ );

        /* calculate the number of apic cycles that occured in the fixed time window and set      */
        /* the apic timer interrupt to fire on those now. The interrupt will fire at cycles/divisor    */
        lapic_write(APIC_TMRINITCNT, (apic_start - apic_current)/APIC_TDIV1);

        return;
}
```

Figure 23 - C Code to Program APIC Timer

3.4.3.3 Configuring the I/O APIC

With the APIC and its timer calibrated, all that remains before booting the application processor cores is configuring the I/O APIC. That recall, replaced the PIC for handling interrupts generated by peripheral devices such as the keyboard and network card. To handle these interrupts it is an actual piece of hardware on the motherboard and configurable via MMIO, but has one unique feature. The MMIO region contains a redirection table for 24 external hardware interrupts. The entries are 64 bits in length and can be accessed between MMIO bits 0x10 to

0x3F, where the first entry's low 32-bits are programmed through MMIO register 0x10 and the high 32-bits through MMIO register 0x11 The second entry uses 0x12 for the low and 0x13 for the high. This incremental process repeats for all of the interrupts supported, which can be found by reading the version register located at MMIO address 0x01. The layout of a single redirection table entry is seen below.

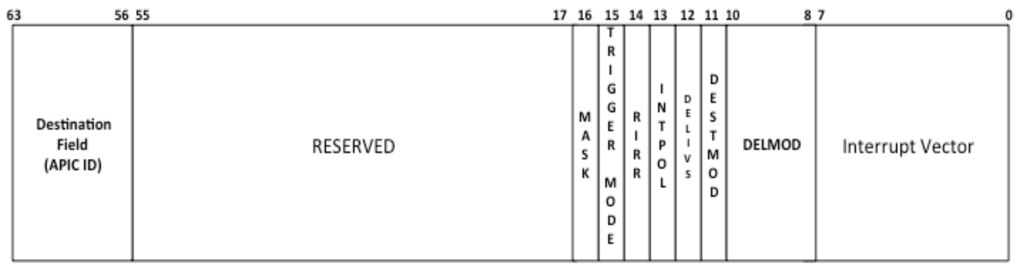
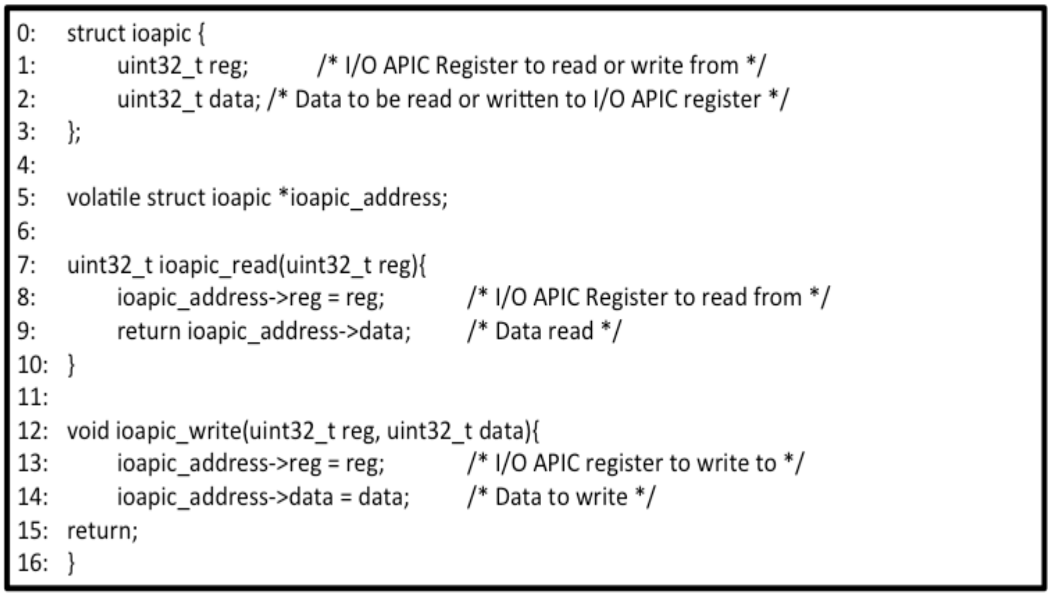


Figure 24 - I/O APIC Registry Table Entry

Hardware interrupts 0-23 are handled through this table, which necessitates the entries to be remapped past the exception interrupts 0-31. The remapping occurs by writing the new interrupt vector to the interrupt field described by bits 0-7. An additional benefit of this table can be seen in that the Destination Field described by bits 56-63, which allows for external interrupts to be routed to any processor core's local APIC on the system. Thus, a system could be built that has a single processor core in charge of handling all of the interrupts generated by a network card. The general course of action however is to first configure the I/O APIC in a default state with all hardware interrupts disabled.

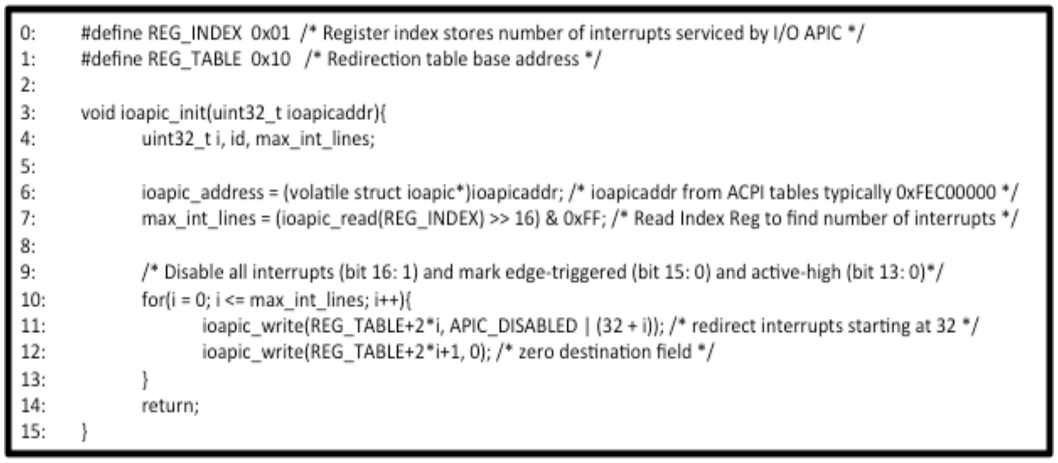
Just as with the APIC, Read and Write helper functions do the heavy lifting for configuring the I/O APIC. This address on most systems generally defaults to 0xFEC00000, but should always be verified with the address reported by the ACPI tables. Also, as configuration is primarily done via the remapping tables, two 32-bit high and low registers; a structure of two *uint32_t* can be used by these helper functions.



```
struct ioapic {
        uint32_t reg;          /* I/O APIC Register to read or write from */
        uint32_t data; /* Data to be read or written to I/O APIC register */
};

volatile struct ioapic *ioapic_address;

uint32_t ioapic_read(uint32_t reg){
        ioapic_address->reg = reg;            /* I/O APIC Register to read from */
        return ioapic_address->data;          /* Data read */
}

void ioapic_write(uint32_t reg, uint32_t data){
        ioapic_address->reg = reg;            /* I/O APIC register to write to */
        ioapic_address->data = data;          /* Data to write */
return;
}
```

Figure 25 - C I/O APIC and I/O APIC Read Write Helper Functions

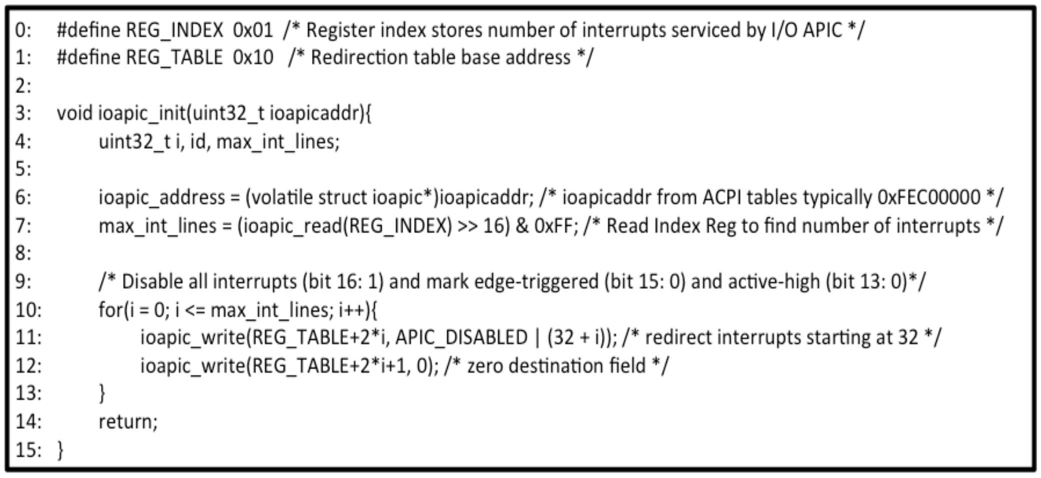
Using these helper functions the initial configuration of the I/O APIC can be streamlined down to 10 lines of code or 12 counting *#defines* for readability. This is simplified further by Intel sharing the same default hex values for enable, disable, assert, etc. between the APIC and I/O APIC as documented in the Intel Software Developer's Manual [5].



```
0:   #define REG_INDEX  0x01  /* Register index stores number of interrupts serviced by I/O APIC */
1:   #define REG_TABLE  0x10   /* Redirection table base address */
2:
3:   void ioapic_init(uint32_t ioapicaddr){
4:           uint32_t i, id, max_int_lines;
5:
6:           ioapic_address = (volatile struct ioapic*)ioapicaddr; /* ioapicaddr from ACPI tables typically 0xFEC00000 */
7:           max_int_lines = (ioapic_read(REG_INDEX) >> 16) & 0xFF; /* Read Index Reg to find number of interrupts */
8:
9:           /* Disable all interrupts (bit 16: 1) and mark edge-triggered (bit 15: 0) and active-high (bit 13: 0)*/
10:          for(i = 0; i <= max_int_lines; i++){
11:                  ioapic_write(REG_TABLE+2*i, APIC_DISABLED | (32 + i)); /* redirect interrupts starting at 32 */
12:                  ioapic_write(REG_TABLE+2*i+1, 0); /* zero destination field */
13:          }
14:          return;
15:  }
```

Figure 26 - C Code to Initialize the I/O APIC

The only register that is read is the Index register, which contains the number of entries in the redirection table (line 7). This is used by the loop (Lines: 10-13) to iterate over all the entries, which disables them (Low 32-bit Register - Line: 11) and then routes them to local APIC zero (High 32-bit Register - Line: 12), which is the BSP. The I/O APIC is now in a known good state and leaves it to the hardware driver software to enable its own interrupt lines. The function that provides this service is below.

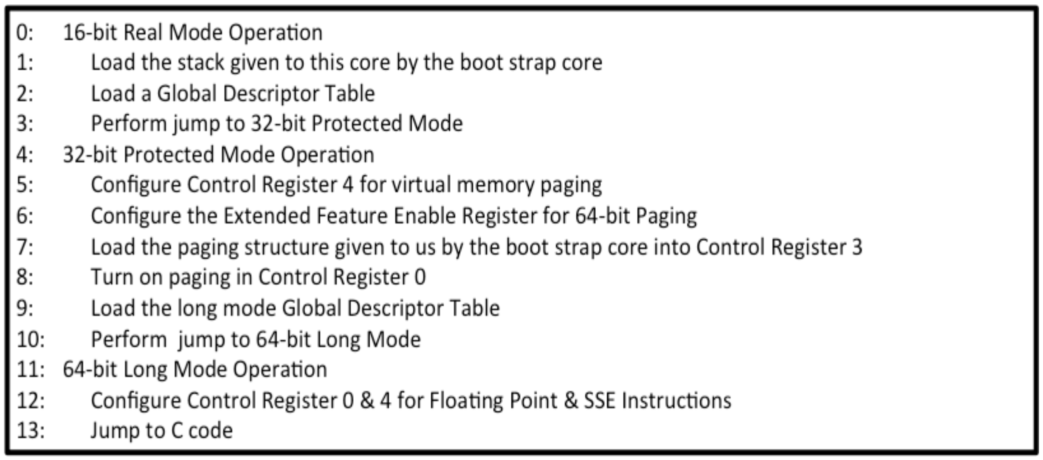


```
0:   #define REG_INDEX  0x01  /* Register index stores number of interrupts serviced by I/O APIC */
1:   #define REG_TABLE  0x10   /* Redirection table base address */
2:
3:   void ioapic_init(uint32_t ioapicaddr){
4:           uint32_t i, id, max_int_lines;
5:
6:           ioapic_address = (volatile struct ioapic*)ioapicaddr; /* ioapicaddr from ACPI tables typically 0xFEC00000 */
7:           max_int_lines = (ioapic_read(REG_INDEX) >> 16) & 0xFF; /* Read Index Reg to find number of interrupts */
8:
9:           /* Disable all interrupts (bit 16: 1) and mark edge-triggered (bit 15: 0) and active-high (bit 13: 0)*/
10:          for(i = 0; i <= max_int_lines; i++){
11:                  ioapic_write(REG_TABLE+2*i, APIC_DISABLED | (32 + i)); /* redirect interrupts starting at 32 */
12:                  ioapic_write(REG_TABLE+2*i+1, 0); /* zero destination field */
13:          }
14:          return;
15:  }
```

Figure 27 - C Code to Enable an I/O APIC Entry

3.4.4 Booting the Application Cores

All of the pieces that support starting and using SMP are now in place. This leaves the BSP in charge of putting a framework in place for these additional AP cores to boot and operate. This work is constrained by the fact that all of the AP cores are placed in a halted state in 16-bit real mode by the BIOS. The BSP rectifies this by placing a small piece of code in low memory that the application core will execute to trampoline itself quickly to 64-bit long mode. The trampoline code is essentially a stripped down version of the stage one bootloader code. As the AP core need only load the bare minimum registers to make it to 64-bit long mode. This pseudo code for this can be seen below and the two important lines to note are 1 and 7 as the bootstrap processor provides these.

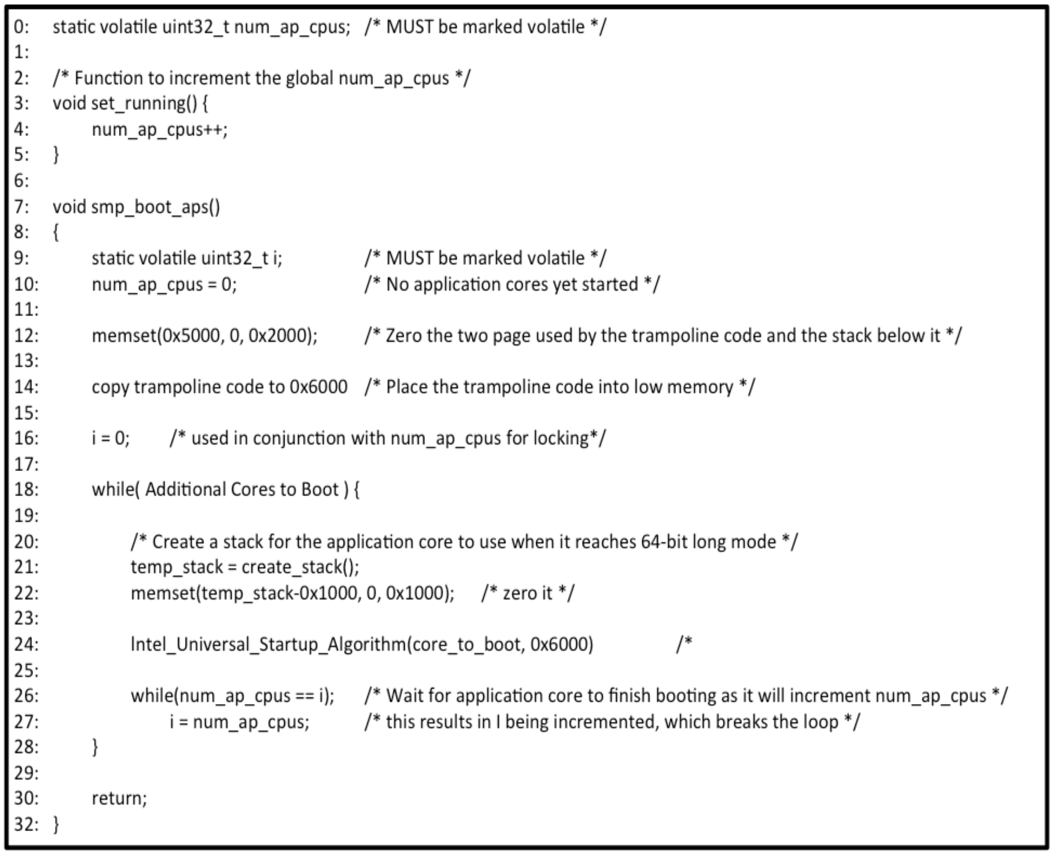


```
0:   16-bit Real Mode Operation
1:       Load the stack given to this core by the boot strap core
2:       Load a Global Descriptor Table
3:       Perform jump to 32-bit Protected Mode
4:   32-bit Protected Mode Operation
5:       Configure Control Register 4 for virtual memory paging
6:       Configure the Extended Feature Enable Register for 64-bit Paging
7:       Load the paging structure given to us by the boot strap core into Control Register 3
8:       Turn on paging in Control Register 0
9:       Load the long mode Global Descriptor Table
10:      Perform  jump to 64-bit Long Mode
11:  64-bit Long Mode Operation
12:      Configure Control Register 0 & 4 for Floating Point & SSE Instructions
13:      Jump to C code
```

Figure 28 – Application Core Trampoline Pseudo Code

The stack (Line: 1) is provided through strategically choosing a location to load trampoline code. Furthermore, this location can’t impact anything else that already exists in low memory and is in use. Recall, the initial identity mapped page tables that address low memory that were created by the stage-one bootloader. These tables start at 0x1000 and go to 0x4FFF. Also remember, that the BIOS leaves the AP cores *HALTED* in 16-bit real mode, where they can only address physical memory. Thus, the trampoline code must not overwrite any part of the BSP's page tables. As well as preventing the AP from performing any action that could corrupt those tables as it executes the trampoline code. Thus, to meet these requirements the trampoline code is loaded at 0x6000.

The AP core also has the added benefit of being able to leverage the already created page tables, which it does by loading the PML4T base 0x1000 into its CR3 (Line: 7). The concern that jumps to mind with shared page tables is that the AP core can now modify the same memory addressed by the BSP core, which could cause the system to crash via non-deterministic behavior. This is addressed in the process used to start the application core via locking and is best seen by walking through the code used to start all the AP cores.



```
0:    static volatile uint32_t num_ap_cpus;   /* MUST be marked volatile */
1:
2:    /* Function to increment the global num_ap_cpus */
3:    void set_running() {
4:          num_ap_cpus++;
5:    }
6:
7:    void smp_boot_aps()
8:    {
9:          static volatile uint32_t i;               /* MUST be marked volatile */
10:         num_ap_cpus = 0;                          /* No application cores yet started */
11:
12:         memset(0x5000, 0, 0x2000);         /* Zero the two page used by the trampoline code and the stack below it */
13:
14:         copy trampoline code to 0x6000    /* Place the trampoline code into low memory */
15:
16:         i = 0;        /* used in conjunction with num_ap_cpus for locking*/
17:
18:         while( Additional Cores to Boot ) {
19:
20:               /* Create a stack for the application core to use when it reaches 64-bit long mode */
21:               temp_stack = create_stack();
22:               memset(temp_stack-0x1000, 0, 0x1000);     /* zero it */
23:
24:               Intel_Universal_Startup_Algorithm(core_to_boot, 0x6000)              /*
25:
26:               while(num_ap_cpus == i);      /* Wait for application core to finish booting as it will increment num_ap_cpus */
27:                     i = num_ap_cpus;          /* this results in I being incremented, which breaks the loop */
28:         }
29:
30:         return;
32:   }
```

Figure 29 - Process to Start Application (AP) Cores

First, note the variables marked volatile (Lines: 0 & 9), this is required due to the shared page tables between cores, caching, and the compiler. This marking ensures that there will be no optimization of these two variables that could result in non-deterministic behavior when the BSP and AP cores access them at the same time. The BSP then zeros the region of low memory used by the AP cores as well as loading the trampoline code (Lines: 12 & 14). The BSP then enters the loop that will boot all of the AP cores on the system (Lines: 18-28). For each AP core it also creates a new stack that will be used by the AP core once it enters 64-bit long mode. This is necessary, as two AP cores cannot use the same low memory stack at the same time. The process of booting the AP cores is iterative as only one AP core boots at a time. The BSP uses the Intel Universal Startup Algorithm [99] to start each AP core booting (Line: 24), followed by the BSP then waiting for the AP core to signal it has finished booting (Lines: 26-27). The loop on lines 26 & 27 is exited once the AP core has made it to 64-bit long mode and called the function *set_running()*, which signals the BSP that is allowed to boot the next AP core.

The signal to boot and tell the application core where to start executing is provided by the Intel Universal Startup Algorithm that is documented in the MP Specification [99]. The algorithm uses special IPIs through the APIC Interrupt Control Register High (ICRH) and Interrupt Control Register Low (ICRL). The ICRH and ICRL are both 32 bit registers, which the APIC converts into a 64-bit long IPI, with the ICRH holding the upper 32-bits and the ICRL holding the lower 32-bits. Bits 56-63 in the ICRH contain the Destination Field that holds the APIC ID to message. ICRL bits 8-10 are the Delivery Mode type, which tells the sending APIC if the IPI is a special IPI (RESET, STARTUP, etc) or a software interrupt. If the IPI is special, the interrupt vector field can be ignored or used to provide additional information. In the case of the Universal Startup Algorithm, it is three serialized special IPIs in the format of: *INIT, sleep(1000), STARTUP, sleep(200), STARTUP, sleep(100)*.

3.4.5 Hypervisor Modifications to Support APIC Access Virtualization

To support the APIC in the virtual machine the hypervisor acts as an abstraction layer. Meaning the changes made in it are completely unknown to the virtual machine operating above. This abstraction is built by modifying the hypervisor's VMExit handler and its transition and state storage code. These changes boil down to the five steps seen below.

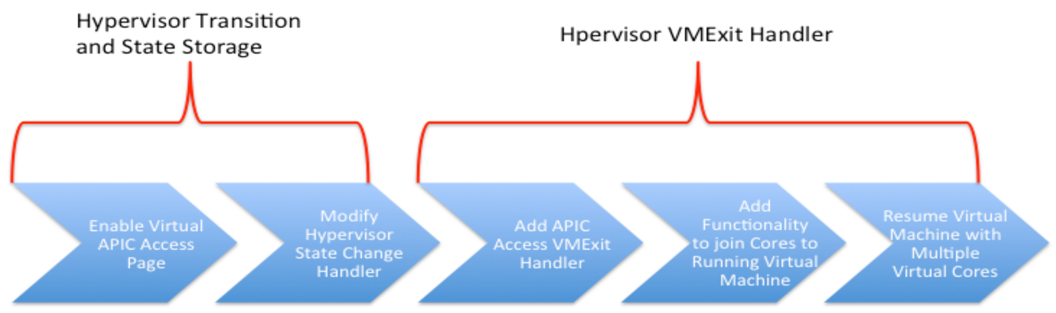

Hypervisor Transition and State Storage
Hpervisor VMExit Handler
Enable Virtual APIC Access Page
Modify Hypervisor State Change Handler
Add APIC Access VMExit Handler
Add Functionality to join Cores to Running Virtual Machine
Resume Virtual Machine with Multiple Virtual Cores

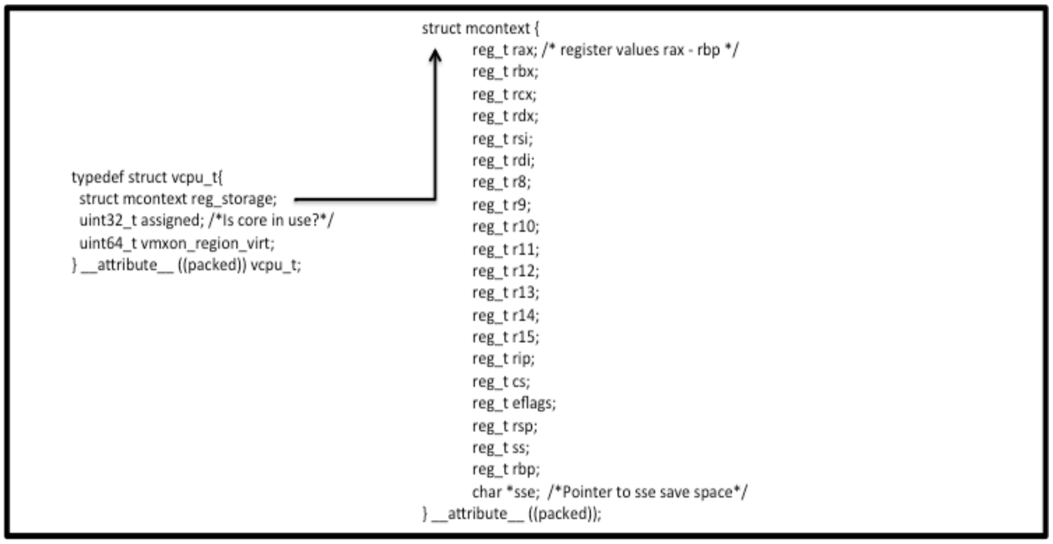
Figure 30 - Steps to Support SMP Guest Virtual Machines

3.4.5.1 Hypervisor Locking and Processor State Storage

To protect hypervisor-controlled resources, such as the EPT, which are only modifiable by a single core at a time, locking must be used during the transitions to the hypervisor from the virtual machine. The method of using a spin-lock to protect the hypervisor is also used by the micro-kernel for scheduling and is fully detailed in section *3.5.1*. Each core transitioning to the hypervisor grabs the lock on entry and releases it upon exit. This prevents any core from modifying the hypervisor state at the same time as another. Thankfully, the spin-lock performance impact on the hypervisor is minimal as transitions to and from the hypervisor are minimized as much as possible by virtualization technologies like EPT and APIC Access Virtualization.

The real difficulty lies in creating storage for each core. Previously, in a single core implementation, a block of the heap was used to store the guest virtual machine state for a transition into the hypervisor. This state is then reloaded prior the guest being resumed. In an SMP hypervisor using this method, every core would try to write to the same heap location. If this were not changed, then every core operating inside the virtual machine would have a corrupted state as they all modify the same block of memory. Thus, each core receives its own block for local state information. This data is also copied into a permanent virtual machine storage location, which provides the guest the ability to resume execution on a different core if need be.

The storage is accessed through a specific core APIC ID (section *3.4.3.1*), which is used to index into the storage array. The layout of the data structure that is stored locally for each core is seen below.



```
typedef struct vcpu_t{
  struct mcontext reg_storage;
  uint32_t assigned; /*Is core in use?*/
  uint64_t vmxon_region_virt;
} __attribute__ ((packed)) vcpu_t;

struct mcontext {
        reg_t rax; /* register values rax - rbp */
        reg_t rbx;
        reg_t rcx;
        reg_t rdx;
        reg_t rsi;
        reg_t rdi;
        reg_t r8;
        reg_t r9;
        reg_t r10;
        reg_t r11;
        reg_t r12;
        reg_t r13;
        reg_t r14;
        reg_t r15;
        reg_t rip;
        reg_t cs;
        reg_t eflags;
        reg_t rsp;
        reg_t ss;
        reg_t rbp;
        char *sse;  /*Pointer to sse save space*/
} __attribute__ ((packed));
```

Figure 31 - C Structure of Hypervisor Core Specific Local Storage

While the *rsp* and *rip* are saved on transition, it should be noted that Intel provides access to both of these values through MSRs, which can be accessed at any point a core is within the hypervisor. They are stored here as a matter of practice to ensure all registers are consistently stored at the same time. The initialization of this structure is accomplished by a "for" loop that runs for the number of cores present on the system and can be seen in the code below.

```
/*create an array the length of cores on the system to store local register*/
/*context for each core                                       */
vcpu_ptr_array = (vcpu_t**)malloc( smp_num_cpus*sizeof(vcpu_t*));

kmemset(vcpu_ptr_array, 0, smp_num_cpus*(sizeof(vcpu_t*)));

for( i = 0; i < smp_num_cpus; i++){
      /*This is not a long term solution. As guests are swapped we will need to */
      /*move some varying length/amount of data into the vproc. As of now though*/
      /*no guests are swapped across cores and each core can maintain it's own */
      /*fiefdom for book keeping                                  */
      vcpu_ptr_array[i] = (vcpu_t*)kmalloc_track(HYPV_SITE, sizeof(vcpu_t));
      vcpu_ptr_array[i]->reg_storage.sse = pes_new_save(); /* malloc space for SSE storage */

      /*Init the vmxon_region for each core and store it in the vcpu_t structure*/
      vcpu_ptr_array[this_cpu()]->vmxon_region_virt =(uint64_t)vkmalloc(vk_heap,1);

      vmem_alloc((uint64_t*)vcpu_ptr_array[this_cpu()]->vmxon_region_virt,
                        PAGE_SIZE, PG_RW | PG_GLOBAL);

      /*By default we assign the BSP to the first guest */
      vcpu_ptr_array[i]->assigned = (this_cpu() == i ? 1 : 0);
}
```

Figure 32 - VCPU Structure Initialization

3.4.6 Enabling APIC Access Virtualization

The next step requires the hypervisor to force exits when the virtual machine tries to boot the AP cores. This is done, by first setting the APIC Access Virtualization bit 0 to 1 in the *Secondary Processor-Based VM-Execution Controls* MSR. Next the hypervisor must allocate one single page of virtual memory for the APIC Access page to reside on. The physical frame corresponding to this virtual page is then mapped into the virtual machine's EPT table at the physical address where the virtual machine's APIC's MMIO register would reside. Thus, when the guest virtual machine's cores read or write to its local APIC MMIO space it generates a VMExit, which the hypervisor can handle.

3.4.6.1 APIC Access Virtualization Exit Handling

When the micro-kernel inside the virtual machine accesses its APIC it now generates a VMExit that reports exit reason 44 for APIC Access when the *VM Exit Reason* MSR is read. The hypervisor now must handle any potential APIC read or write request generated by the guest virtual machine's cores. The code that performs this is located in Appendix H and is explained by the flow chart below.

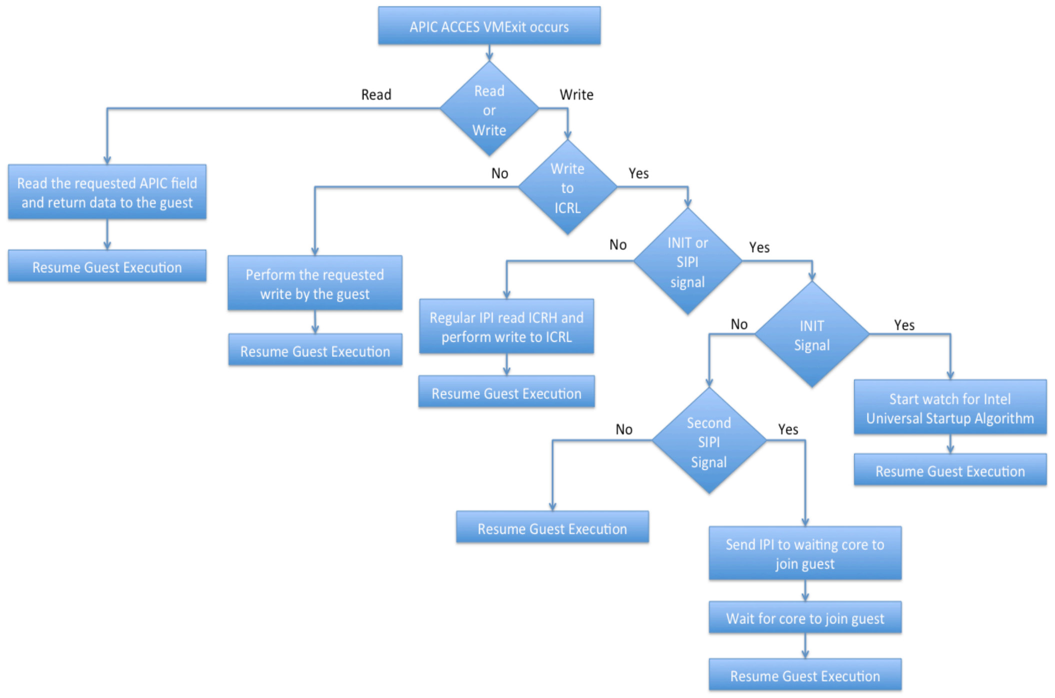

APIC ACCES VMExit occurs
Read
Read or Write
Write
Read the requested APIC field and return data to the guest
Resume Guest Execution
No
Write to ICRL
Yes
Perform the requested write by the guest
Resume Guest Execution
No
INIT or SIPI signal
Yes
Regular IPI read ICRH and perform write to ICRL
Resume Guest Execution
No
INIT Signal
Yes
Start watch for Intel Universal Startup Algorithm
Resume Guest Execution
No
Second SIPI Signal
Yes
Resume Guest Execution
Send IPI to waiting core to join guest
Wait for core to join guest
Resume Guest Execution

Figure 33 - Hypervisor APIC VMExit Handling Flow Chart

The first step in handling the exit is to determine if the access was a read or a write. Reading the *VM Qualification* MSR does this, where if bit 12 is 1 then the guest performed a write and if bit 12 is 0 then the guest performed a read. In the event of a read, the hypervisor determines the APIC field that is being read by reading bits 0-11 of the *VM Qualification* MSR. The hypervisor then performs the read for the guest, placing the read value in a location the guest will use, and then resumes execution of the guest.

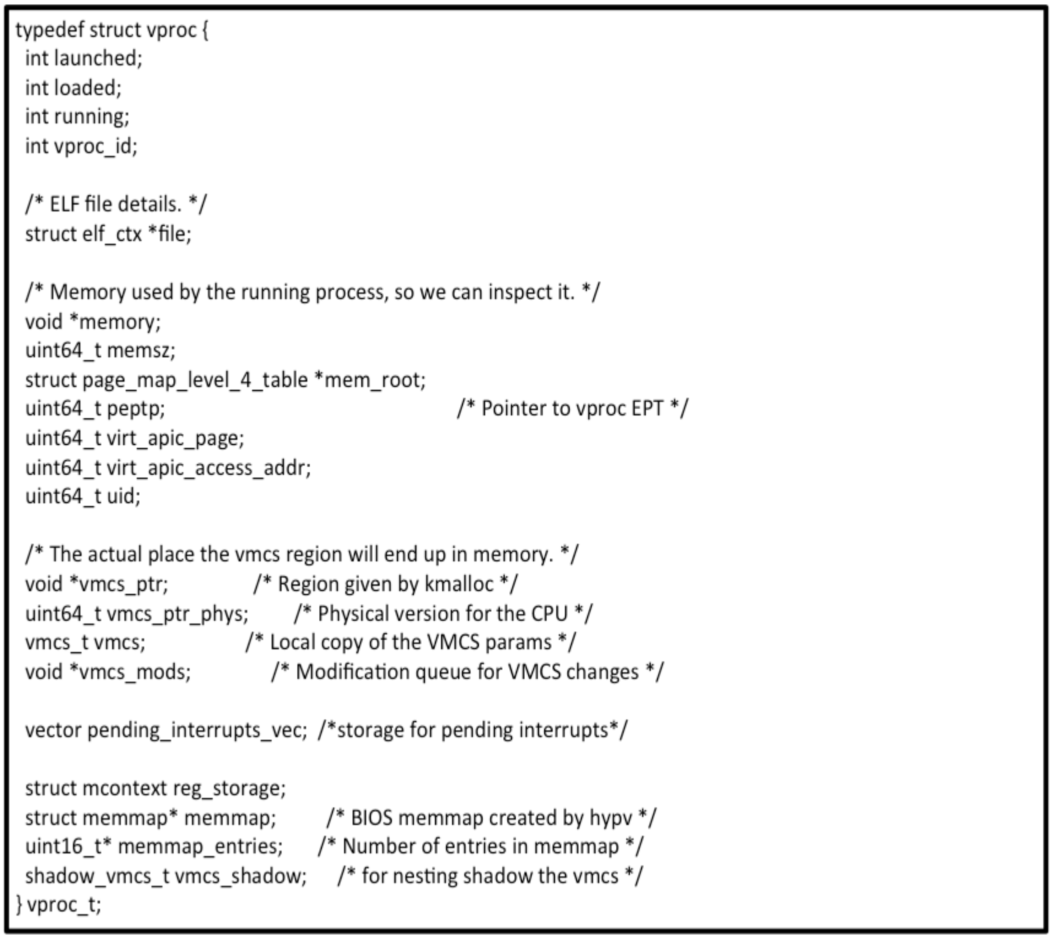
If the guest tried to write to the APIC MMIO register, then it must determine if the write was a write to ICRL. The *VM Qualification* MSR bits 0-11 again contain the field that was being written to. If it wasn't a write to ICRL, the hypervisor then performs the write to any other field for the guest and resumes guest execution. If instead it was a write to the ICRL, then the hypervisor must determine if the write was an IPI or not.

The write can be determined to be an IPI using the 64-bit long mode function calling conventions. Recall the *lapic_write* function (section *3.4.3.1*), where the second variable passed to it is a data variable. In 64-bit long mode, this value is passed in the *rsi* register. The hypervisor can then read the *rsi* register to see what data is being written to the APIC. If the data matches the INIT or SIPI signal, then the write is a special IPI. In the case that it doesn't match, it is a regular IPI, which is handled by writing the guest requested ICRL data. It is important to note that the guest, prior to writing to the IRCL, has already written the ICRH, which was handled by the hypervisor as a normal write and contains the destination of the IPI.

In the case of an INIT or SIPI signal the hypervisor begins to track the guest to see if it is using the Intel Universal Startup Algorithm. The first INIT starts a count to look for the three startup IPIs. The INIT itself is ignored by the hypervisor and the guest then resumes execution. Next, the hypervisor will grab the next SIPI in the algorithm, which it will also ignore. Once the receipt of the second SIPI, the hypervisor will then take action to join another AP to a virtual machine. This is accomplished by sending an IPI to a waiting AP to inform it needs to join a guest. The process of joining to the guest is explained in further detail in the next section. The BSP sends the IPI and waits for the AP by reading the APIC Delivery bit 12 in its own local APIC ICRL. This bit will remain 1 until the AP writes zeros to its own local APIC End of Interrupt (EOI) register, which also zeros the delivery bit on the sending local APIC. The sending core can then resume guest execution.

3.4.6.2 Joining Cores to a Running Guest

With a system in place to catch and handle APIC accesses, all that is left to support SMP guests is to join another AP to the running system. As mentioned in the previous section, a waiting AP in the hypervisor is sent an IPI to start this process. As IPIs are limited to just the software interrupt vector in terms of data, a global variable is often used with them to pass data from one core to another. In this case the data is a pointer to the running virtual machines virtual process structure, which can be seen below.



```
typedef struct vproc {
  int launched;
  int loaded;
  int running;
  int vproc_id;

  /* ELF file details. */
  struct elf_ctx *file;

  /* Memory used by the running process, so we can inspect it. */
  void *memory;
  uint64_t memsz;
  struct page_map_level_4_table *mem_root;
  uint64_t peptp;                         /* Pointer to vproc EPT */
  uint64_t virt_apic_page;
  uint64_t virt_apic_access_addr;
  uint64_t uid;

  /* The actual place the vmcs region will end up in memory. */
  void *vmcs_ptr;           /* Region given by kmalloc */
  uint64_t vmcs_ptr_phys;     /* Physical version for the CPU */
  vmcs_t vmcs;              /* Local copy of the VMCS params */
  void *vmcs_mods;           /* Modification queue for VMCS changes */

  vector pending_interrupts_vec;  /*storage for pending interrupts*/

  struct mcontext reg_storage;
  struct memmap* memmap;        /* BIOS memmap created by hypv */
  uint16_t* memmap_entries;     /* Number of entries in memmap */
  shadow_vmcs_t vmcs_shadow;     /* for nesting shadow the vmcs */
} vproc_t;
```

Figure 34 - C Structure for Guest Virtual Machine

This structure having already been populated by the hypervisor to start the already running virtual machine makes joining another core to the same virtual machine a straightforward endeavor. The virtual machine control registers on the joining core are loaded with all the same data as the already running guest (VMX registers, APIC Access Page, EPT pointer, etc), which is performed by the *join_to_vproc* function on line 17 in the code below. This function takes the running guests vproc structure as an argument and *mallocs* a clone of it for the core joining the already running guest. Additionally, this core's interrupts are disabled and its APIC EOI register is cleared before joining the core to the running guest using the *VMLAUNCH* command.

```
void ap_join_guest(){
    vproc_t *vp;
    int rc;

    asm volatile("cli"); /* ensure interrupts are halted on this core */

    lapic_eoi(); /* clear APIC EOI register */

    acquire_lock(sem_hypv); /* make sure this core has the hypervisor lock before modifications */

    /*check vmx features and call vmxon and assign it to this cores data structure*/
    hypv_entry(vcpu_ptr_array[this_cpu()]->vmxon_region_virt);

    vcpu_ptr_array[this_cpu()]->assigned = 1; /* mark core in use by a virtual machine */

    /* use information of already running virtual machine */
    /* to populate this core's VMCS structure */
    vp = join_to_vproc(SIPI_vp);

    rc = load_vproc(vp); /* load this core's vmcs structure with VMPTRLD*/

    if (rc != 0){
        kprintf("load_vproc return code %d.\n", rc);
        asm volatile("hlt");
    }

    rc = launch_vproc(vp); /* launch virtual machine with VMLAUNCH */

    /* Unreachable, unless there was a major error launching */
    kprintf("error; return code %d.\n", rc);
    kputs("Halting now.");
    asm volatile("hlt");

    return;
}
```

Figure 35 - Function to Join AP to Virtual Machine

3.5 Micro-Kernel SMP Scheduling Considerations

After all the AP cores have joined the micro-kernel virtual machine, scheduling of user processes can begin. The scheduling system must provide each core its own idle process, which is run when no other user process is ready to be scheduled.

This section assumes that a system is in place to create new page tables for each new process, which results in multiple CR3 targets.

3.5.1 Locking and Transitions

The micro-kernel is sacrosanct in its control over memory and must be protected. Where any processor core may make changes in a process' user memory without worry of impact to another core, this is not true for the micro-kernel memory. Every process has the same micro-kernel mapped into its page tables and if two cores were to access protected features at the same time, the system could crash. For example if two cores accessed the micro-kernel heap at the same time, both cores could be assigned the same block of memory to use. The standard solution to this problem and the one used by this here is a single spin-lock for the micro-kernel, which Linux had a form of with the Big Kernel Lock up until the 2.6.39 Linux kernel was released [122].

```
typedef struct spin_lock_t{
        uint64_t s_counter;             /* Counter 0 or 1 for spin locks */
        uint8_t owner;                          /* What processor core has the lock */
} spin_lock_t;

static uint64_t exchange(volatile uint64_t *lock_address, uint64_t value){
        uint64_t ret_value;

        /* assembly instruction xchgq is an atomic swap instruction provided by Intel */
        asm volatile("lock; xchgq %0, %1" :
                        "+m" (*lock_address), "=a" (ret_value) :
                        "1" (value) :
                        "cc");
        return ret_value;
}

void acquire_lock(volatile spin_lock_t *lock) {

        while(exchange(&lock->s_counter, 1) != 0);  /* Spin trying to acquire the lock */

        lock->owner = (lapic_read(APIC_APICID)>>24) & 0xFF;  /* Owner is the Core that grabbed the lock */

        return;
}

void release_lock(volatile spin_lock_t *lock) {

        exchange(&lock->s_counter, 0);    /* Release the lock */

        lock->owner = NO_OWNER;   /* NO_OWNER = -1 */

        return;
}
```

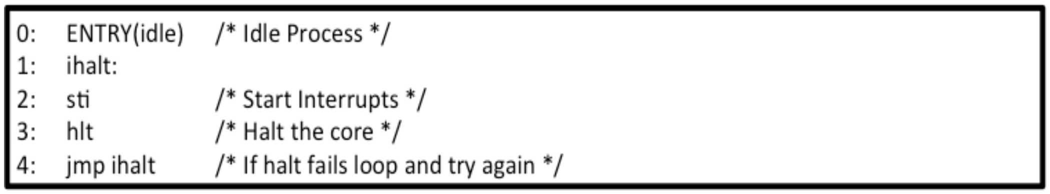
Figure 36 - C Code to Implement Spin-Locking

The heart of the spin-lock code is the x86 assembly instruction *xchgq*, which is an atomic instruction, meaning that it is guaranteed only one processor core will execute it at a time. In this case of the *exchange* function, the instruction swaps either a 1 or a 0 with the value stored in *lock_address*. If the operation succeeds a 1 is returned and if it fails a 0 is returned. The *exchange* function is called by *acquire_lock*, which will continuously call *exchange* until a 1 is successfully swapped in. The *exchange* function is also called by *release_lock*, which will either swap in a 0 or have no effect if a 0 has already been swapped in. These three functions form the basis of the spin-lock, with *acquire_lock* called whenever a core transitions to the micro-kernel and *release_lock* called whenever a core exits.

During this transition to the micro-kernel, the core saves the user space registers. Since this is process specific, these registers are stored inside the process structure of a user process. These registers are then restored prior to a process resuming user space execution. The more interesting transition occurs when the micro-kernel registers are saved when a process transitions inside its kernel space to another process' kernel space. In this case the registers are again saved in the process structure, but the current process' CR3 target is also saved into the process structure. Subsequently, the CR3 target of the next process to run is then written into the core's CR3. This results in the old processes page tables being flushed from the TLB and the new processes page tables becoming the MMU's active set of tables for translation, which are stored in the TLB as the MMU traverses them. [30] Furthermore, the system does not crash, because each process has the same micro-kernel mapped into its page tables. So, while the new process remains in the same location in micro-kernel virtual memory, it has an entirely new user virtual memory space.

3.5.2 User & Idle Process Scheduling

The last issue to contend with is what to do when there are fewer user processes to run than there are cores on the system. In this case the cores not running a process have to idle. This is achieved by creating a four line idle process with its own CR3 for each core to run in this scenario.



```
ENTRY(idle)   /* Idle Process */
ihalt:
sti           /* Start Interrupts */
hlt           /* Halt the core */
jmp ihalt     /* If halt fails loop and try again */
```

Figure 37 - Assembly Code to Idle a Core

In the well-known round robin scheduler algorithm [123] the next process in the scheduling queue is always the next to be run. In a single core system, the scheduling queue almost always remains full of user processes, but also has been loaded with a single idle process in case no user processes are available to run. In a SMP system, the multiple cores can quickly drain the scheduling queue of user processes and the single idle process to the point where one core may have no process to run next. For that exact case, the idle process is removed from the scheduling queue. Instead, each core has its own idle process assigned to it, which is run when the queue is empty.

3.6 Benchmarks and Analysis

The memory and AIM 9 benchmarks described in chapter 1 were used to measure performance. In addition to the Dell OptiPlex 9010, a MacBook Pro with 8GB RAM and a 4-core 3.2GHz Intel i7 processor was also used for benchmarking. The Dell system ran the Micro-Kernel, Micro-Kernel on Custom Hypervisor, Fedora with 3.17.4-301 Linux Kernel, and Fedora 3.17.4-301 Linux Kernel on the

Xen 4.4 Hypervisor. The MacBook Pro ran VMware Fusion, a type 2 hypervisor [124] with an Ubuntu with 2.6.32-38-generic Linux Kernel guest that has 4Gb of ram and 4 processor cores provided to it from the MacBook Pro.

Table 3 below provides the average number of processor cycles and the time from twenty runs of the memory and AIM 9 tests described in Chapter 1. The Table itself is broken into the Cycles it took to complete the memory benchmark, AIM9 benchmark, and total for both. The Table also provides the time in seconds it took to complete the respective benchmarks and total time for both.

	Memory Cycles	AIM9 Cycles	Total Cycles	Memory Time (s)	AIM9 Time (s)	Total Test Time (s)
Micro-Kernel	2.9574E+11	1.4455E+11	4.4029E+11	86.98	42.52	129.50
Micro-Kernel - Custom Hypervisor	2.9956E+11	1.5442E+11	4.5398E+11	88.11	45.42	133.52
Fedora Kernel	4.7962E+11	1.4368E+11	6.2330E+11	141.06	42.26	183.32
Fedora Kernel - Xen Hypervisor	4.0746E+11	1.9247E+11	5.9994E+11	119.84	56.61	176.45
Ubuntu Guest - VMware Fusion	4.6057E+11	0.9390+E11	5.5447E+11	143.92	29.34	173.27

Table 3 - Memory and Processor Benchmarks

Notice that the memory and the recursive paging system described here, on both the micro-kernel and the micro-kernel executing on the custom hypervisor is faster than Fedora, Fedora on Xen, and Ubuntu on VMware. Some of this performance gain can be attributed to the fact that a micro-kernel is a much lighter weight operating system than a full Linux kernel and thus can create processes at a faster rate. However, the purpose of benchmark, the creation of 100 processes with a large number of *malloc()* and *realloc()* iterations, is to focus the performance measurements on virtual memory for user space as a whole. This provides an additional level of confidence that the performance gains can be attributed to recursive paging.

Processor performance based on the AIM9 benchmarks had three noteworthy points of comparison: Although it was expected that the Micro-Kernel would outperform the larger Fedora Kernel, this is not the case. In fact, both systems scored roughly the same in terms of the number of cycles and time, with the Fedora kernel edging out the micro-kernel by ~.251 of a second to complete the AIM9 benchmark. This can be attributed to the superior scheduling offered by Fedora, while the micro-kernel performs well due to it simplicity: Both approaches result in the AIM9 tests running at all times directly on a core.

It is important to notice the impact of a newer processor on the AIM9 benchmark. The type 2 VMware hypervisor running Ubuntu is running on a slower processor and with 4 less cores, but that processor was released ~15 months after the processor shipped with the Dell. The difference a year can make is staggering: the Ubuntu guest finishes the AIM9 benchmarks almost a full 13 seconds faster than any configurations running on the Dell.

Finally, the presence of a hypervisor slows performance of the AIM9 benchmark on all of the systems. The micro-kernel has the smallest impact, which is due to configuring the hypervisor to operate the guest as close to real time as possible. Larger hypervisors such as Xen and VMware are designed to manage multiple guests, which implies some configurations, which are suitable for a micro-kernel is not suitable for them. This can be seen in the larger performance impact when comparing Fedora to Fedora on Xen.

3.7 Summary

The implementation of a full SMP system, let alone one that also supports an SMP hypervisor requires, a great deal of research into both hardware and software architectures. Once implemented though, it provides fine grain control of the system through ACPI, the APIC, I/O APIC, Paging, Virtualization, and so on. It provides an unparalleled ability to quickly redefine the system to any new specification, which for this work is the break up and reduction of the micro-kernel into the UVM architecture.

Chapter 4 – Utility Virtual Machines

To reduce the *attack surface* of the micro-kernel and replace it with a collection of UVMs, three key challenges were resolved: isolation of specific functionality within separate UVMs, communication and synchronization between virtual machines, and the allocation of virtual machines to processing cores to balance load across the cores. Isolating functionality is a solved problem, as any unique UVM service exists in the micro-kernel. Furthermore, being a micro-kernel the driver components of the system are heavily modularized and have little kernel specific code other than messaging interfaces. This allows any driver to be removed from the system without major modification to the micro-kernel itself. In fact, multiple different testing configurations of the drivers existed prior to UVMs, to include: networking with NFS, no networking, keyboard only, and so on. This makes the creation of UVMs a process of choosing one of these configurations and then compiling out any unneeded functionality.

The task of communication and synchronization between virtual machines required some outside the box innovation. As the message passing system, which uses an asynchronous model [33] to implement system calls and inter-process communication, was initially kernel only. Furthermore, the semantic gap [18] that protects the hypervisor from virtual machines and virtual machines from each other is now a factor. As inter-process communication is central to UVMs, message passing was extended to the hypervisor and new facilities were be built to cross the gap.

The problem of assigning cores to specific UVMs required a complete redesign of the original Bear concept of a lightweight hypervisor [25]. The version one hypervisor was designed to support the operation of a single running guest micro-kernel. With UVMs the hypervisor had to adapt to support the simultaneous operation of multiple lightweight task specific guests. Because the concurrent operation of VM, as well as the use of shared functionality guests, had never been previously executed, this represented an immense technical leap.

4.1 Building the first UVM

The first task was to build and run a solitary instance of the keyboard/VGA UVM on bare-metal without the hypervisor. The reasoning being that the keyboard and VGA drivers in order to run relied on the following assumed small subset of facilities:

- System Calls – Fork, Exec, Get Process Identifier, User Malloc, Map Video Ram
- The hardware Interrupt for the Keyboard – 0x21
- User Space Drivers – VGA and Keyboard

To reach this reduced working set, the procedure was largely manual, as build scripts had to be modified to remove the compilation of unnecessary pieces into the micro-kernel. Additionally, the micro-kernel itself was modified to remove obsolete system calls. This leaves the remaining services: fork and exec two driver processes, the drivers to map their process identifier and Video RAM, and user space *malloc* for the standard I/O message passing. Combining these minimalistic configurations results in the creation of the architecture seen in Figure 38.

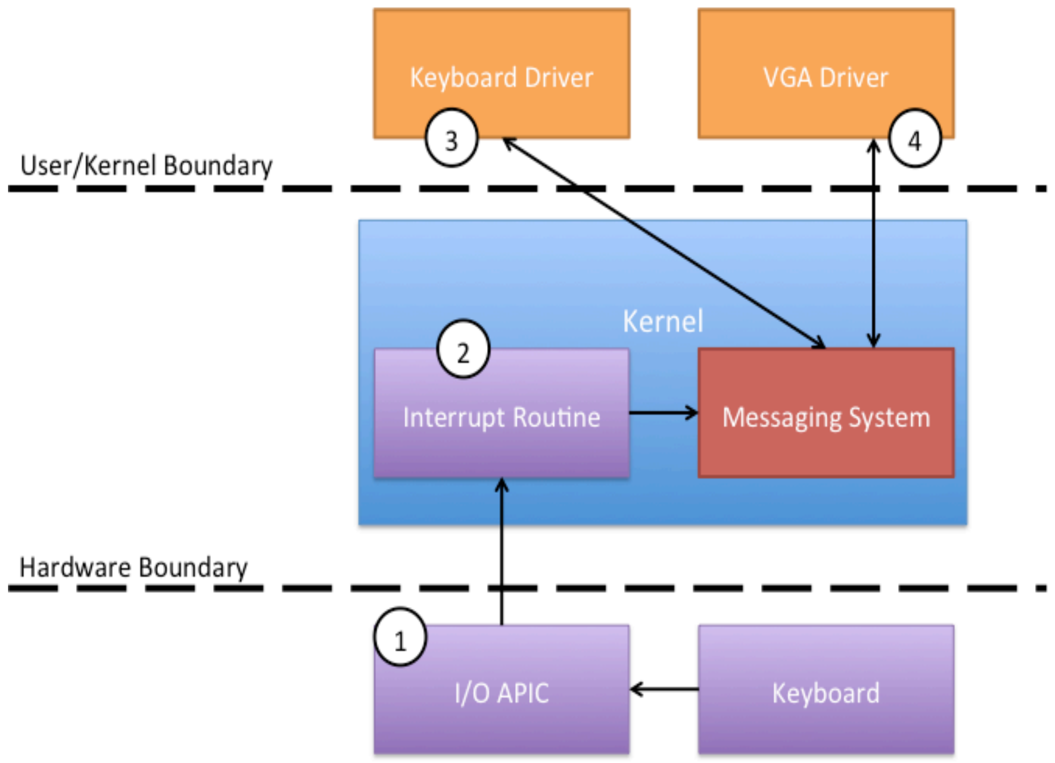
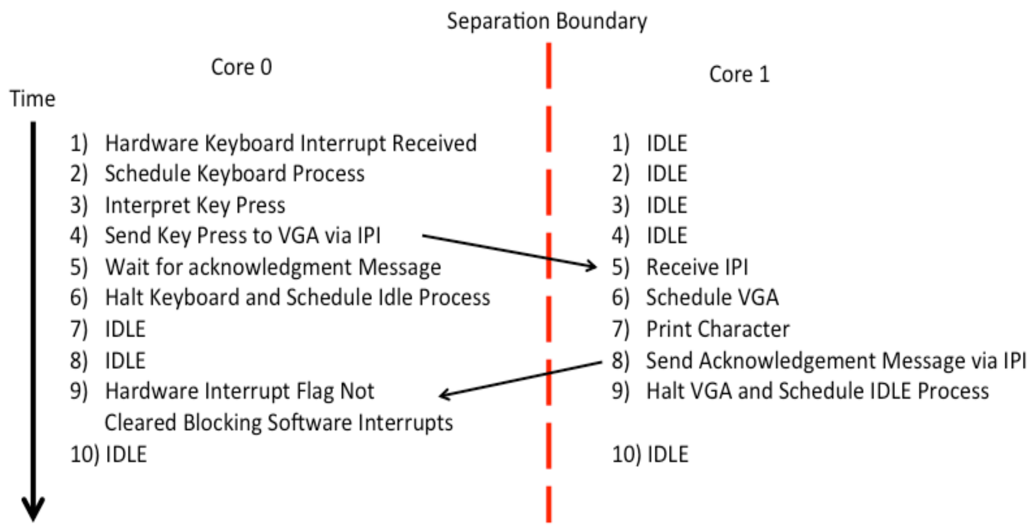


Figure 38 - Keyboard/VGA UVM Kernel Operation

This setup only handles keyboard input and then prints it to screen, which all starts with a key press in (1). This key press triggers the I/O APIC to send the hardware interrupt 0x21 signal to the micro-kernel. The micro-kernel (2) contains the code to handle this interrupt and acknowledge that it was received. The messaging system then sends a generic keyboard interrupt message to the keyboard driver in (3). The keyboard driver upon receipt interprets what key was actually pressed and then sends a message to the VGA driver containing the ASCII [125] character. The VGA driver (4) upon receipt of the message prints the character to screen.

However, there was one impediment with the messaging system in this architecture that was not immediately realized until testing. As the asynchronous model for messaging is in use, both the drivers start up and issue a message receive, which is a blocking call, meaning both processes are halted until the messaging system unblocks their execution when they have a message ready for them. Until they can run, the idle process is executed, which recall just waits for an interrupt to be triggered. So, upon key press the keyboard and VGA drivers are unblocked and scheduled due to the hardware interrupt triggering the scheduling routines of the micro-kernel. After the letter is printed to screen the system hangs due to a cascading set of circumstances related to scheduling and interrupts, which is best illustrated in Figure 39.



Separation Boundary
Core 0
Core 1
Time
1) Hardware Keyboard Interrupt Received
2) Schedule Keyboard Process
3) Interpret Key Press
4) Send Key Press to VGA via IPI
5) Wait for acknowledgment Message
6) Halt Keyboard and Schedule Idle Process
7) IDLE
8) IDLE
9) Hardware Interrupt Flag Not Cleared Blocking Software Interrupts
10) IDLE
1) IDLE
2) IDLE
3) IDLE
4) IDLE
5) Receive IPI
6) Schedule VGA
7) Print Character
8) Send Acknowledgement Message via IPI
9) Halt VGA and Schedule IDLE Process
10) IDLE

Figure 39 - Interrupt Timing in Prototype UVM

The process of communication between cores works as expected in steps 1 through 7. Messaging and IPIs only break down in steps 8 and 9 when the VGA driver responds to the now halted keyboard driver. The keyboard driver fails to reschedule after receipt of the IPI, because IPIs are software interrupts, which will not be received if the hardware interrupt flag is set. The hardware interrupt flag is still set on core 0, because it can only be cleared after the acknowledgement receipt, which tells the keyboard it can now receive subsequent key presses. This prevents out of order receipt or loss of key press by the keyboard driver. Thus, with the response IPI never being received, the scheduler is never run and both cores are stuck idling.

Thankfully, the solution lay in the adjustment of the initial assumptions to include the timer interrupt, which itself is a hardware interrupt and cannot be overridden like a software interrupt. Initially, it was absent, as the goal was to reduce the overhead of unneeded interrupt context switches when neither the VGA or keyboard drivers were in use. Unfortunately, the need to break out of the idle process persists, which requires it to remain. Upon reintroduction of the timer interrupt, the system performs as expected and multiple key presses can be handled with no hangs.

4.2 Extending Message Passing to the Hypervisor

Next the UVM for Keyboard and VGA had to be run on top of the hypervisor and pass messages between drivers through the hypervisor. Running a single UVM on the hypervisor is no different than running a single micro-kernel guest on the hypervisor. As such, discussion of this is omitted in this thesis, but a detailed description is available by Kanter [126]. The main technical tasks covered here are the extension of the messaging system and crossing the semantic gap to interpret the messages. This can be visualized by thinking of Figure 38 above as the top orange component of the larger system in Figure 40 below.

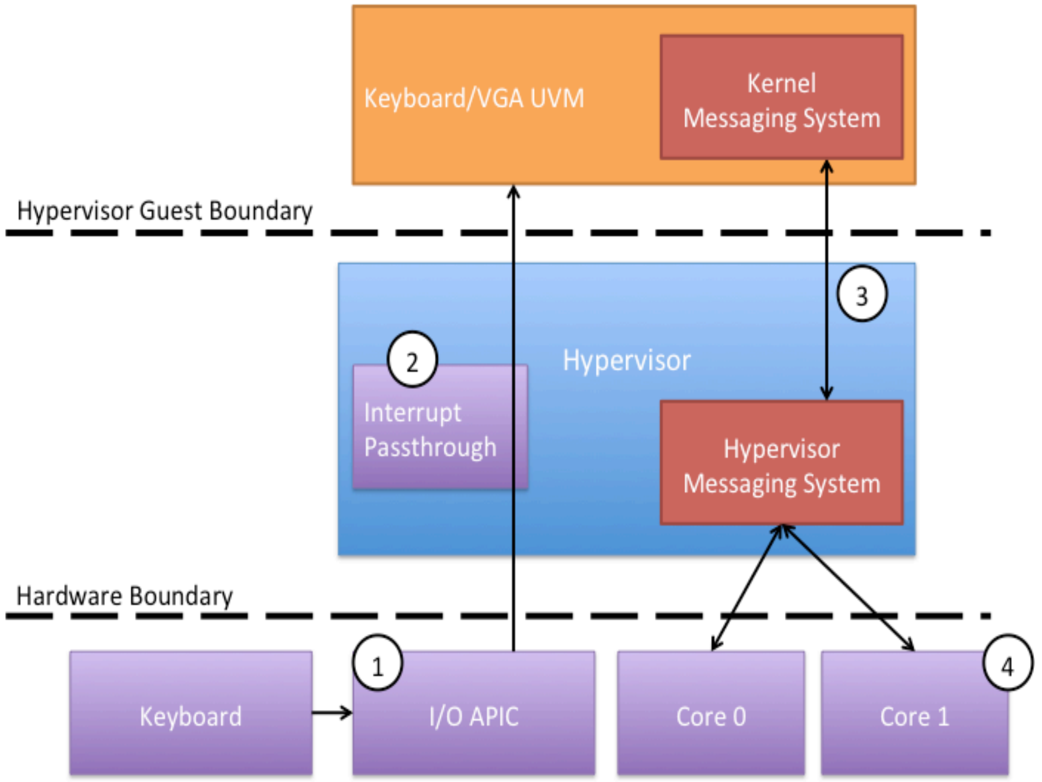


Figure 40 - Keyboard/VGA UVM Running on the Hypervisor

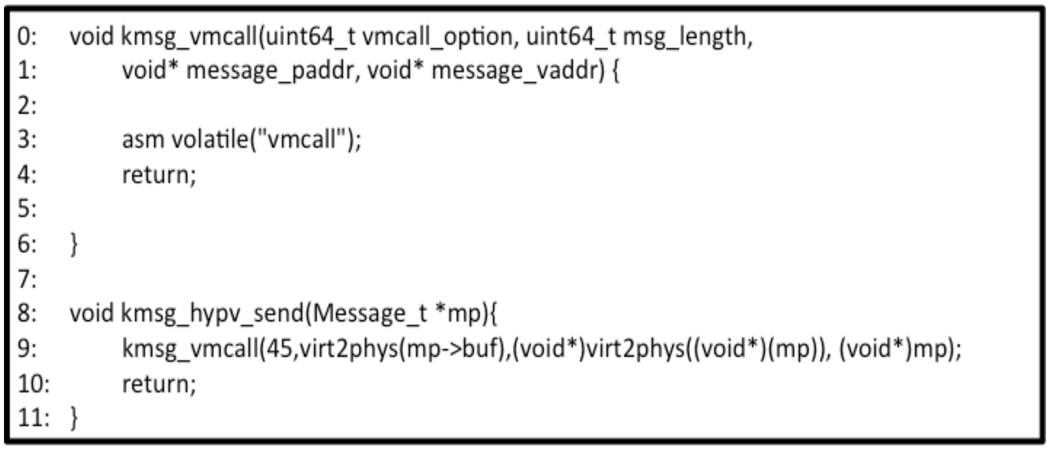
In this new architecture the I/O APIC (1) uses interrupt passthrough (2) [5] to deliver the hardware interrupt directly to the running keyboard/VGA UVM. Consequently, no additional handling overhead has to be added to the hypervisor for interrupts. The extensive changes come from coupling the micro-kernel messaging system to the hypervisor messaging system through *VMCALLs* (3) and the passing of messages between cores (4) via IPIs. Development of this initial prototype pegged the keyboard process to core 0 and the VGA process to core 1 with all messages between them routed through the hypervisor.

Step (3) has two components, which are the new hypervisor messaging queue and the vmexit handler that is used to cross the semantic gap to interpret messages. The queue code can be seen in Appendix I and is comprised of four functions:

- *init_util_msg_queue(void)* opens the queue and is called during the startup of the hypervisor.
- *add_util_msg(Util_msg_t* msg)* adds messages to the queue.
- *remove_util_msg(void)* removes messages from the queue that are meant for the core it is called on.
- *static int core_msg_cmp(void* msg, const void* core_number)* is the helper function called by *remove_util_msg(void)* to find messages assigned to a specific core.

The hook into the hypervisor for inter-VM message sends and receives is provided by a *VMCALL*, which is a special virtualization instruction that forces a guest to exit to the hypervisor [5]. To fully understand what is happening in step (3) and how it relates to IPIs in (4), it is best to go through the process of the keyboard driver sending a message to the VGA driver.

The message send with the ASCII character to be printed still goes into the standard micro-kernel messaging system. However, before calling a normal message send it instead calls the *hypv_msg_send* function seen in the Figure 41.



```
0:    void kmsg_vmcall(uint64_t vmcall_option, uint64_t msg_length,
1:          void* message_paddr, void* message_vaddr) {
2:
3:          asm volatile("vmcall");
4:          return;
5:
6:    }
7:
8:    void kmsg_hypv_send(Message_t *mp){
9:          kmsg_vmcall(45,virt2phys(mp->buf),(void*)virt2phys((void*)(mp)), (void*)mp);
10:         return;
11:   }
```

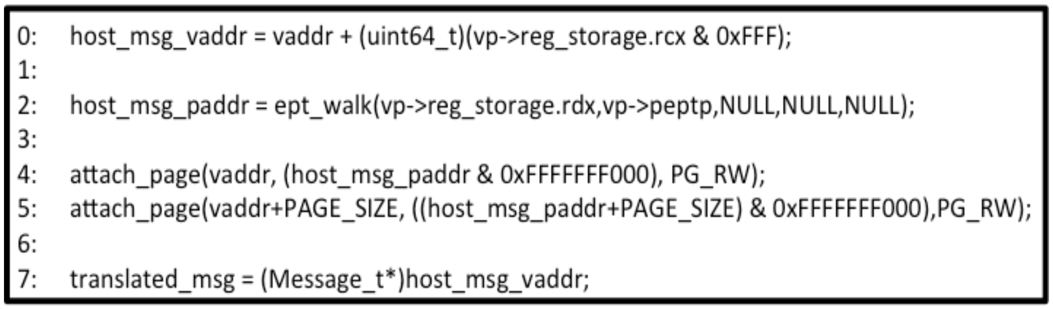
Figure 41 - kmsg_hypv_send code

This function is a wrapper for *kmsg_vcall* on line 9, which takes the following arguments:

- 45 – Specifies this is a message send for the *VMCALL* vmexit handler.
- virt2phys(mp->buf) – The guest physical address of the message buffer.
- (void*)virt2phys((void*)(mp)) – The guest physical address of the message header.
- (void*)mp – The guest virtual address of the message.

After the vmcall instruction on line 3 is issued, core 0, which is running the keyboard driver is dropped into the hypervisor. The arguments passed to *kmsg_vmcall* are accessed by the hypervisor through the guest state variables according to x86-64 calling conventions: 45 in *rdi*, virt2phys(mp->buf) in *rsi*, (void*)virt2phys((void*)(mp)) in *rdx*, and (void*)mp in *rcx*. The hypervisor then uses these variables to cross the semantic gap and decipher the message that is being sent.

Those four variables are enough to cross the gap, because they provide all the information as to where the message is located in host physical memory. Thus, the hypervisor need only to map the host physical address relating to the guest physical address into its virtual memory to obtain the message, which is accomplished by the code in Figure 42.

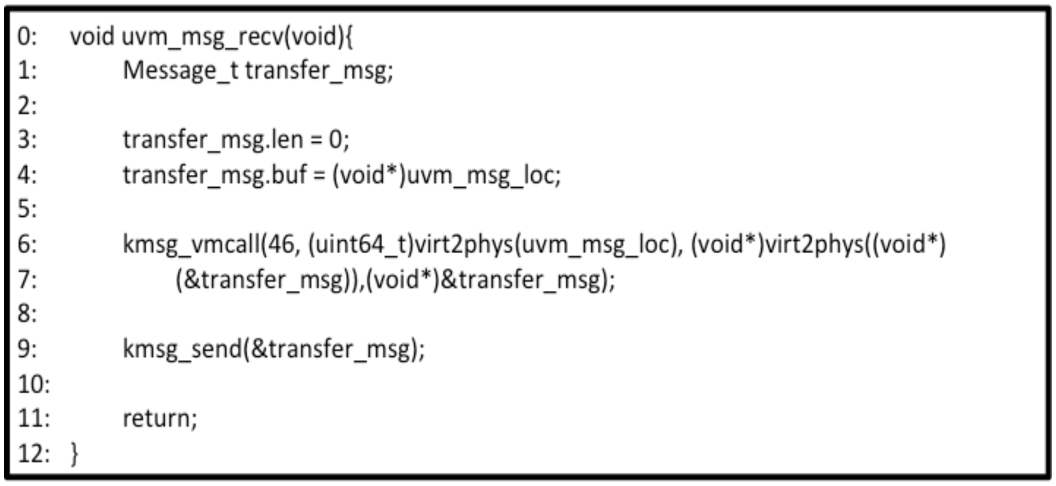


```
0:    host_msg_vaddr = vaddr + (uint64_t)(vp->reg_storage.rcx & 0xFFF);
1:
2:    host_msg_paddr = ept_walk(vp->reg_storage.rdx,vp->peptp,NULL,NULL,NULL);
3:
4:    attach_page(vaddr, (host_msg_paddr & 0xFFFFFFF000), PG_RW);
5:    attach_page(vaddr+PAGE_SIZE, ((host_msg_paddr+PAGE_SIZE) & 0xFFFFFFF000),PG_RW);
6:
7:    translated_msg = (Message_t*)host_msg_vaddr;
```

Figure 42 - Crossing the Semantic Gap

Any unassigned virtual address has the offset of the guest virtual address added to it (line 0). This is because messages are stored in the kernel heap and are not paged aligned. Then the EPT is traversed to find the host physical address (line 2) that relates to the guest physical address. This host physical address is then attached to the unassigned virtual address (lines 4 & 5). Then and only then can the hypervisor access the guest’s message (line 7). However, This process has to be repeated for every pointer contained in a message, which is the reason the message buffer guest physical address is also passed to the *hypv_msg_send*. Note the virtual address of the guest message buffer is not needed as that is contained in the message header.

Once, the message is fully deciphered it is stored in the hypervisor messaging queue and tagged with the core it is destined for, which in this case is core 2. Core 2 is notified by core 0 that it has a message through the software IPI 0x8F. The IPI forces Core 2 to execute the interrupt handler code in Figure 43 for a hypervisor message receive inside the micro-kernel of the UVM.



```
0:   void uvm_msg_recv(void){
1:           Message_t transfer_msg;
2:
3:           transfer_msg.len = 0;
4:           transfer_msg.buf = (void*)uvm_msg_loc;
5:
6:           kmsg_vmcall(46, (uint64_t)virt2phys(uvm_msg_loc), (void*)virt2phys((void*)
7:                   (&transfer_msg)),(void*)&transfer_msg);
8:
9:           kmsg_send(&transfer_msg);
10:
11:          return;
12:  }
```

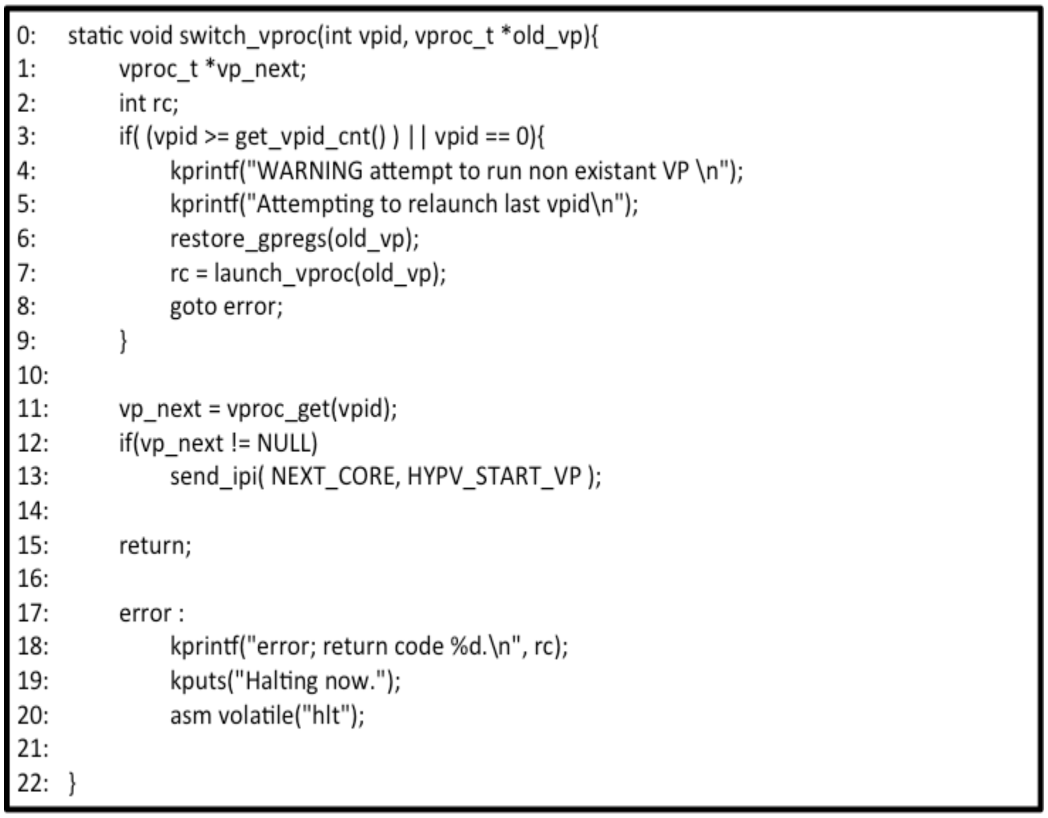
Figure 43 - UVM Message Receive Interrupt Handler

The handler sets up an empty message that will be filled in by the hypervisor (lines 1, 3, & 4). The *uvm_msg_loc* variable is a block of memory allocated inside of a UVM for message transfers. As it is impossible to know how large the message buffer will be, two contiguous 4KB pages are used. Then the handler issues a *VMCALL* (line 6 & 7), which is almost identical to the *VMCALL* in Figure 43, except it passes 46 instead of 45. This tells the hypervisor to treat it as a message receive in the vmexit handler. Core 2 now in the exit handler again performs all the same steps to cross the semantic gap as were done in Figure 42, but with the addition of copying the previous message send into the empty translated message variable. The hypervisor returns core 2 to normal execution in the UVM it will issue a local message send (line 9). The VGA driver will then receive the message to print a character from the keyboard driver. This whole process then repeats again in reverse when the VGA driver sends the acknowledgement receipt back to the keyboard driver.

4.3 Pairing Two UVMs Together

With the keyboard/VGA UVM in place, it was time to pair it with the Shell UVM, which contains all user space processes. However, to do this, support for concurrent operations of two virtual machines was needed. To speed this work, the decision was made to statically assign cores to specific UVMs. In this way the keyboard/VGA UVM could be assigned core 0 and the Shell UVM could be assigned the remaining cores 1-7. This also allows for the UVMs to chain load with the next loading after the preceding one has finished its micro-kernel initialization.

Prior to either UVM starting, two vprocs are created that each has their own independent virtualization control structures. The only actual difference is the tailored code for the jobs they support. The keyboard/VGA UVM starts on core 0 first and has one additional line added to the end of its kernel initialization, "*kvmcall(0xA,2,0);*", which is used to chain load the next UVM. This *VMCALL* has core 0 execute the hypervisor code in Figure 44.



```
0:   static void switch_vproc(int vpid, vproc_t *old_vp){
1:          vproc_t *vp_next;
2:          int rc;
3:          if( (vpid >= get_vpid_cnt() ) | | vpid == 0){
4:                 kprintf("WARNING attempt to run non existant VP \n");
5:                 kprintf("Attempting to relaunch last vpid\n");
6:                 restore_gpregs(old_vp);
7:                 rc = launch_vproc(old_vp);
8:                 goto error;
9:          }
10:
11:         vp_next = vproc_get(vpid);
12:         if(vp_next != NULL)
13:                send_ipi( NEXT_CORE, HYPV_START_VP );
14:
15:         return;
16:
17:         error :
18:                kprintf("error; return code %d.\n", rc);
19:                kputs("Halting now.");
20:                asm volatile("hlt");
21:
22:  }
```

Figure 44 - Launch Second UVM Code

The majority of the code is error handling (lines 3-9 & 17-20), as there are a few cases that must be accounted for. First, no UVM can be assigned virtual process ID 0 as that is reserved per Intel instruction [5]. Also, the virtual process ID given from the *VMCALL* cannot be higher than the actual number of UVMs present. Avoiding this, the number two is passed, which represents the virtual process ID for the shell UVM. From there the vproc is found (line

11) and then an IPI is sent to the core 2 (line 13), which starts the Shell UVM. Note core 2 and not core 1 is started first as this conforms to the start order provided by ACPI [98].

Once both UVMs are up and running they immediately start communicating with each other. The keyboard driver sends messages to the Shell when the enter key is pressed. The VGA driver receives messages locally from the keyboard and between VMs from standard output such as *printf()*. The complete UVM architecture can be seen in the Figure 45.

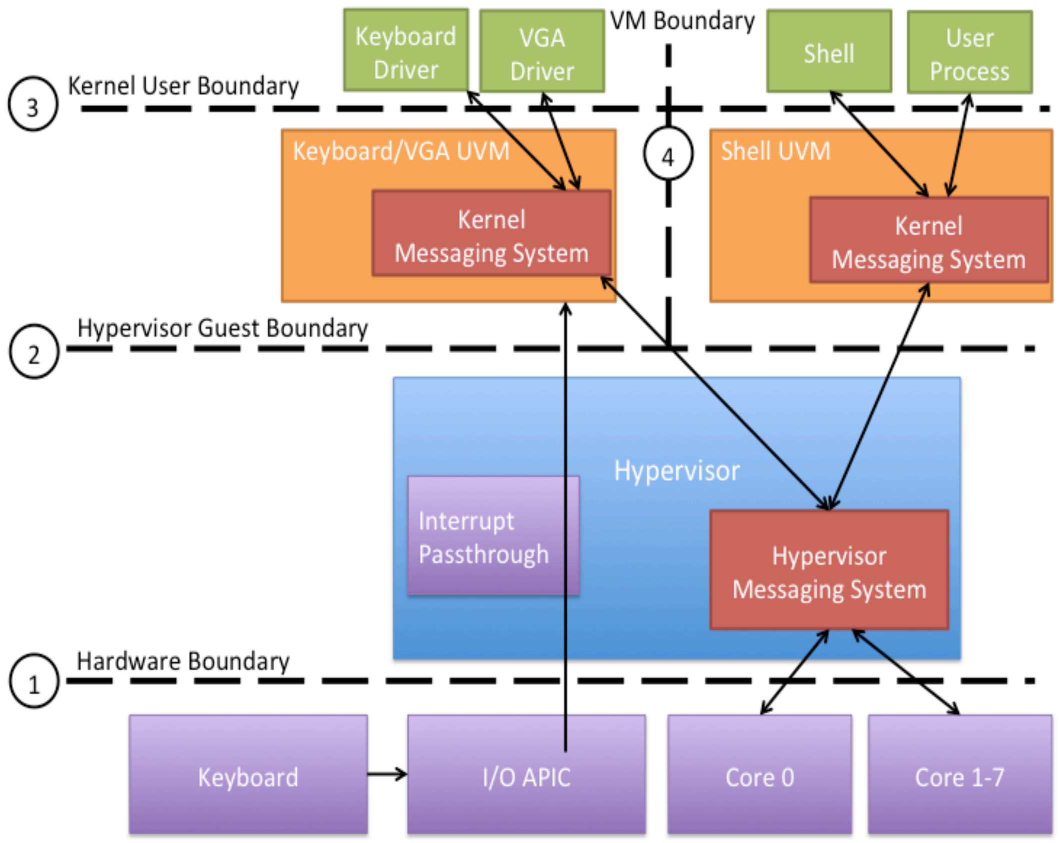


Figure 45 - Complete UVM Architecture

Now the true security benefits can be clearly seen through the dividing lines in this diagram. The hardware boundary (1) results in only the hypervisor having access to the hardware. The semantic gap (2) enforced by the EPT further abstracts the hardware from the UVMs, but also protects the hypervisor from malicious guests. The last abstraction is the kernel and user boundary (3), which isolates the UVM micro-kernels from their user processes or device drivers. The UVMs themselves are protected from each other through virtualization (4) and all communications between them is protected through first their messaging system and then the hypervisor messaging system. Furthermore, these inter-UVM communication channels are strictly enforced by the hypervisor where anomalous sends between UVMs are disallowed. For example, if the keyboard driver attempted to communicate with any other process than the shell the hypervisor would halt the execution of the Keyboard/VGA UVM.

4.4 Benchmarking and Analysis

Again the memory and Aim 9 benchmarks were used and the results can be seen in Table 4.

	Memory Cycles	AIM9 Cycles	Total Cycles	Memory Time (s)	AIM9 Time (s)	Total Time (s)
Bear Micro-Kernel	2.9574E+11	1.4455E+11	4.4029E+11	86.98	42.52	129.45
Bear Micro-Kernel & Hypervisor	2.9956E+11	1.5442E+11	4.5398E+11	88.11	45.42	133.52
Keyboard/ VGA UVM (no network)	3.0288E+11	1.4533E+11	4.4821E+11	89.08	42.74	131.83

Table 4 - Keyboard/VGA UVM Benchmarks

The performance of the hypervisor messaging system and the UVMs was better than expected. As each inter-UVM message adds two *VMCALLs* (send & receive) it was predicted that the system would slow a significant amount, because more time would be spent in the hypervisor and crossing the semantic gap can be expensive [87]. However, the UVMs performed about equal to the micro-kernel with hypervisor. Being about ~1 second slower in memory and ~3 seconds faster in AIM9, which results in ~2 second decrease in total test time.

The memory performance was within margin of 1.1% and cannot definitively be attributed to UVM messaging, less cores for the multiple process test, or just margin of error. However, the AIM9 performance was 6.1% greater and can be attributed to the UVM architecture, which completely offloads printing to a separate VM. This free the Shell UVM to capitalize those few extra cycles to finish the AIM9 test faster.

As for attack surface and the mitigation of zero-day threats it is important to note all the differing attack surfaces in the UVM architecture. First, the keyboard/VGA

UVM micro-kernel has shrunk from 2,904 lines of code to 2,072, which is a decrease of 33.4%. Furthermore, the lines of code contained in user space are 1,210 lines versus the full micro-kernels user space of 50,592, which is a difference of 190.7%. The beauty of the UVM architecture is that much of the missing user components are provided in a separate Shell UVM. The Shell UVM saw its micro-kernel decrease to 2,311 lines, which is a reduction of 7.1%. The Shell UVM's user space shrank to a size of 2,600 lines for a difference of 180.4%. However, these large user space reductions primarily come from eliminating the network functionality from these UVMs, as it comprises 46,612 lines of user space code. The next chapter deals with returning this network functionality to the architecture through an additional network UVM. Lastly, the hypervisor saw an increase from 2,489 lines to 2,654 to support the UVM messaging system and simultaneous UVM operation. This was an increase of 6.4%, but these are also the hardest to reach for an attacker as they are protected by the semantic gap.

4.5 Summary

Besides some initial difficulty, the hypervisor messaging system and inter-UVM communication component were completed to form the basis of the UVM architecture. The hypervisor now contains much of the same MPI based messaging system that was originally built for the micro-kernel. The exceptions being that messages are routed based on core ID instead of process ID, and that only designated processes can initiate messages between UVMs. This provides an added layer of security by enforcing communication as one-way channels. More importantly, it also proves that hypervisor messaging is possible with current hardware, which was an early-identified challenge.

The inter-UVM communication was effectively built through the coupling of IPIs with *VMCALLs*, thus, providing an effective means for one VM to interrupt another when it is ready to receive a message that it is being sent. Notably, this method, while increasing time spent in the hypervisor, actually increased performance of the overall system. Allowing for more tasks to execute simultaneously through the introduction of multiple lightweight UVMs. This current architecture precludes the largest portion of the system, which is the networking component. The lack of a complete system means the question of it being possibly to fully fragment a kernel has not been completely answered.

In terms of security advancements, the initial architecture is very promising. The kernel and user code bases have been significantly shrunk, which minimizes the *attack surface* and increases *attacker workload* by reducing the number of gadgets available for ROP attacks. Attackers are further hindered through sandboxing provided by the semantic gap through EPT, which is harder to cross as the code base shrinks.

Chapter 5 – A Further Abstraction: The Network UVM

As mentioned earlier the networking subsystem, which consists of the e1000 driver and Network File Sharing Daemon (NFSD) has not been added. These are the largest and most complex pieces of the operating system as they interface with everything from hardware to user space libraries. Separating them from the system as a whole required more architectural engineering and introspection than that of the previously implemented keyboard/VGA UVM.

The primary service provided by the networking component is to load binaries from a trusted store that resides within the cloud [28]. The transfer occurs in a newly *forked* process' user space *execve* system call prior to a program running. The code that performs this uses four message types to interface with the NFSD: *Stat* – checks if the file is there; *Open* – creates a network path to the file; *Seek* – moves the file system pointer to the start of the file; and *Read* – copies the file to a local object. Once the binary is transferred, the *execve* code sends a message to the micro-kernel to copy the user space binary and then load it into memory for it to be executed. Therefore, the hypervisor must pass these same NFSD messages back and forth between any network UVM and the already existing shell UVM.

This is complicated in by the *Read* interface, which is actually an Application Programming Interface (API) [127] wrapper that hides multiple low level data read messages. While the e1000 card supports data transfers of 2KB or 2048 bytes, the NFSD only allows for a max read per request of 1KB or 1024 bytes. To ensure a file is read fully, multiple read messages are sent in a *do while loop* that continues until the entire file has been copied locally.

Under these constraints, 210 read messages would be needed to load a file like the shell, which is 213,895 bytes in size. According to the UVM architecture presented in chapter 4 this would entail 418 *VMCALL*s for the read operation and this excludes any additional messages needed for *Stat, Open,* and *Seek*. Thus, a different approach was chosen as the overhead for the inter-VM messages and multiple memory copies throughout the hypervisor would be immense.

5.1 Network Utility Virtual Machine Helper Daemon (NUVMHD)

One option would have been to rewrite the NFSD and its interfaces to implement the network UVM. However, potentially breaking an interface to implement the UVM architecture goes against the adopted principle of modularity [25] and hinders future portability. Instead, the solution chosen entails the encapsulation of the API into the NUVMHD process seen in Figure 46, which was first implemented on top of the micro-kernel without the hypervisor.

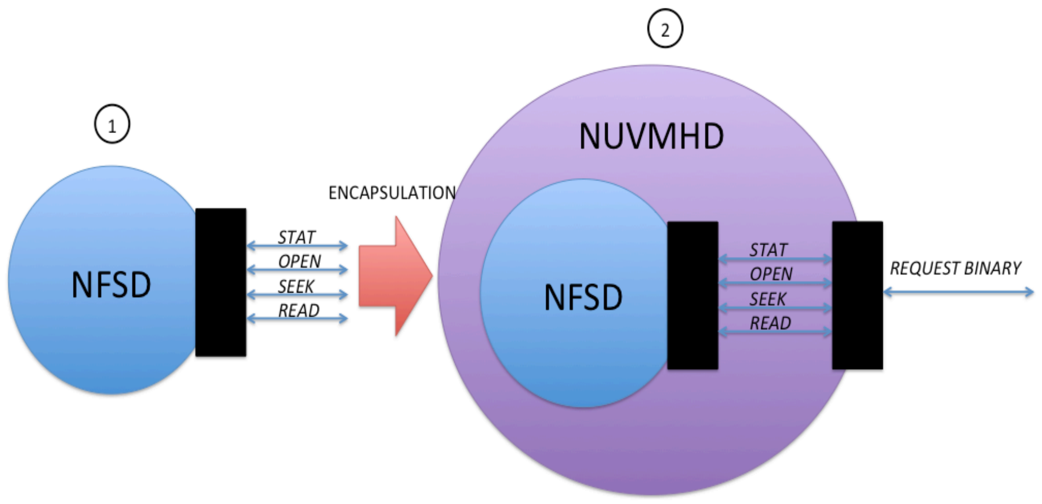


Figure 46 Encapsulation of NFSD into NUVMHD

The significance of this is to picture these processes themselves contained within the network UVM and the black box being the message interface into the UVM. Where by going from (1) to (2) the interface complexity is reduced by a factor of four and *Stat*, *Open*, *Seek*, and *Read* messages are all bottled into two messages. The send and reply code itself is 8 lines, which can be seen in Figure 47.

```
0:    typedef struct {
1:          int type;
2:          char path[MAXPATHLEN];
3:    } nuvmhd_req_t;
4:
5:    typedef struct {
6:          int type;
7:          int size;
8:          char  *addr;
9:    } nuvmhd_resp_t;
```

Figure 47 - NUVMHD Send and Reply Structure

To obtain a binary from the NUVMHD another process sends the string (line 2) of the binary it is trying to download. Once received, all of the same steps that were taken in the user portion of *execve* are followed, which ends in the binary being loaded into the NUVMHD user virtual memory. The catch here is the process that requested the binary does not have access to the loaded binary, because every user process has its own unique virtual address space due to multiple CR3 targets. This is alleviated by the NUVMHD transferring the binary to the shared micro-kernel through a system call. This transfer was originally performed by *execve* so the elf loader could load the process, which means no new work is done as it has been relocated from *execve* to the NUVMHD. Upon completion, the reply message that is sent back to the requesting process contains the address (line 8) of the now loaded binary.

5.2 Implementing the Network UVM

The creation of the Network UVM followed the same manual process that was involved in building the keyboard/VGA and shell UVMs. The newly created VM was tailored to include the e1000 driver, the NFSD, and the NUVMHD. It was also modified to chain load the next UVM after it had been initialized.

There still remained one impediment to it being fully functional though, which again related to the binary. The NUVMHD interface between VMs worked as intended in that the shell UVM would request a binary, the network UVM would load the binary in its own context, and then it would reply with the address the binary had been loaded at. The shell UVM, if it attempts to access this address would crash, because the binary is still stuck within the network UVM. The reason for this is the hypervisor only transfers the message header and the message buffer, which between the two contains only a pointer to the binary (figure 2, line 8).

To resolve this issue further introspection is needed to transfer the binary between the VMs. This introspection is enabled by adding the guest physical address of the binary to the fields of the *kvmcall* function, which allows the host physical address of the binary to be found in the EPT. Recall, the virtual address for alignment and size of the binary are already provided in the reply buffer. All of this information is then used to copy the binary from the guest into a *malloced* space inside the hypervisor. The pointer to the binary is then stored in the message it loads in its messaging queue in place of the address that was provided from the network UVM. The hypervisor then notifies the shell UVM through IPI that the reply message is ready for it.

The shell UVM of course knows nothing about the binary and the hypervisor has no way of knowing where to copy the binary into the shell UVM. To work around this, the shell UVM has a block of blank memory that it zeros and passes as the address in the reply message that the hypervisor will use. Again, this address must have its guest physical address passed in the *kvmcall* function in order for the hypervisor to use it. To ensure there is enough space for the binary, a block that is larger than known binary sizes is used.

Once the shell UVM core is inside the hypervisor it then pulls the message from the queue. It introspects the address provided by the shell UVM so as to map and then copy the binary to that memory. The shell UVM then resumes function with the same virtual address, but with the actual binary located behind it. More importantly this eliminates the need for the system call in the shell UVM, as the hypervisor copies the binary directly into the micro-kernel. The system call must remain in the network UVM as *VMCALL*s can only be executed from the micro-kernel and the binary is initially loaded in user space.

5.3 Benchmarking and Analysis

Prior to testing, it was believed that the presence of the Network coupled with the network UVM would cause a slow down, because of it size and scope. However, this was not the case as seen in the test results of the memory and AIM9 tests in Table 5:

	Memory Cycles	AIM9 Cycles	Total Cycles	Memory Time (s)	AIM9 Time (s)	Total Time (s)
Bear Micro-Kernel	2.9574E+11	1.4455E+11	4.4029E+11	86.98	42.52	129.45
Bear Micro-Kernel & Hypervisor	2.9956E+11	1.5442E+11	4.5398E+11	88.11	45.42	133.52
Keyboard /VGA UVM (no network)	3.0288E+11	1.4533E+11	4.4821E+11	89.08	42.74	131.83
Network UVM	3.0362E+11	1.4682E+11	4.5043E+11	89.30	43.18	132.48

Table 5 - Network UVM Benchmarks

Instead, the network UVM performed on par with the keyboard/VGA UVM configuration. The only noticeable effect was a slight slow down of the AIM9 tests, which is most likely attributed to a hypervisor bottleneck. As the core i7 processer that is in use does not support end of interrupt virtualization, which requires every core to drop into the hypervisor to clear the APIC flag that corresponds to it on the issuance of every interrupt. The network card, which generates a large amount of interrupts needs this flag cleared often, which means the core handling the network UVM may be holding the hypervisor lock when a shell UVM core also needs to clear the flag.

Furthermore, the network UVM still performed ~1 second faster than the hypervisor and micro-kernel only configuration. While this is only a .78% difference and cannot be fully claimed as a performance improvement, it can nonetheless be claimed that performance wasn't decreased due to UVMs. The equality of performance can be surmised to be from the separation of the interrupt heavy network card from the main user component. In this way while less cores are present in the shell UVM, the user processes are interrupted less due to the absence of the network card, which is most noticeable in AIM9 testing.

In regards to attack surface the network UVM micro-kernel had a reduction from 2,904 lines of code to 2,492 resulting in a 15.3% reduction. The user space code decreased by 7.97% from 50,592 to 46,715 lines, 103 of which are the NUVMHD. The hypervisor saw an increase of 52 to support the additional introspection and copies. The Shell UVM micro-kernel saw a small increase by 4 to allocate the block of memory needed for binary transfers. The user space portion needed 20 lines to manage messaging between it and the network UVM.

5.4 Summary

Networking, the final component of a complete operating system, has been successfully added to the UVM architecture. Any user now has access to the same services that a monolithic kernel would provide, but with the added benefit of enhanced hardware protections and split UVM architecture, proving that it is possible to securely fragment a kernel and still maintain normal operation.

Additionally, buggy device drivers used as a springboard for a ROP attack may still be able to infect their UVM, but they have lost their ability to extend that compromise to other parts of the system. In a similar vein, malicious users have lost the ability to access and break trusted device drivers. This is all provided on top of the fact that each individual component, user or driver has a significantly reduced micro-kernel and user space *attack surface*, which increases *attacker workload* through the restriction of available gadgets. Also, in contrast to most security techniques, performance of UVMs was either on par or better than a standard hypervisor, which proves that security can be added to the system that has little or no performance overhead.

Chapter 6 – Heat Diffusion Scheduling

With the advent of the UVM architecture there is an every increasing number of independent tasks and processes being executed. While initial UVM performance with static core assignment is on par or better than a standard hypervisor. This will not always be the case, as UVM technology transitions deeper into the cloud and UVMs begin floating between cores. This presents a challenge of identifying a new scheduling technique for the execution of a multiplicity of tasks across cores.

What to choose is a complex task, as a wide selection of scheduling algorithms can be utilized in operating system design and no one size fits all. For example, real time computing systems [128] must respond to high priority jobs as soon as they occur, because not doing so could result in system failure. For these types of environments, preemptive schedulers [129] are used to give preference to highest priority jobs first and the lowest priority jobs last. In contrast, larger operating systems often use multilevel feedback queues, which partition the ready queue into two or more queues [130]. For each new process that is scheduled the system determines which queue to place the process in. Each queue may have its own unique scheduling algorithm based on the processes it serves. Additionally, this allows an under served process to be rescheduled in a higher priority queue and likewise an over served process to a lower priority queue. However, The main goal of all of these algorithms is to minimize resource starvation [93], which is when a process is denied access to a resource it needs to finish execution.

In the context of most operating systems, the critical resource is CPU cycles needed to execute user, kernel, or hypervisor code. The first version of Bear was a uniprocessor system that ran a handful of user process drivers and user programs. Thus, fairness of scheduling was provided to each through the round-robin algorithm [123]. As it provides a starvation free solution by offering every processes the same length time slice to run on the processor core before the next process is scheduled. The enforcement of time slices was provided through the Programmable Interrupt Timer (PIT), which fired at a constant time interval.

As Bear matured however, new hardware architectures documented in chapter 3 were added to replace legacy systems. The most impactful changes to scheduling were the replacement of the PIT with the higher resolution APIC timer and the transition to SMP. The APIC Timer allows scheduling of processes to occur at a faster rate than allowable with the PIT. Additionally, the APIC architecture allows for the scheduling of processes across all of the cores available. This was not possible in the early versions of Bear that utilized the PIT. Nonetheless the round-robin scheme can still be used with SMP and a discussion of its software components follows below.

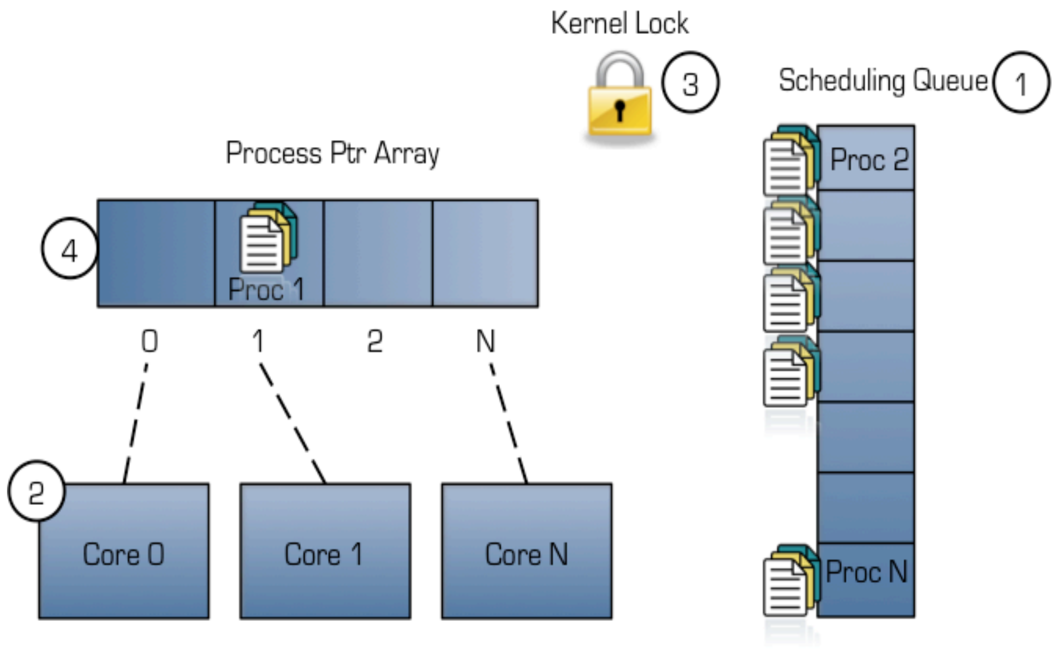


Figure 48 - Scheduling Software Components

First, all processes that are able to run are stored in ready to run queue (1). When any of the cores (2) generate a timer interrupt they grab a kernel lock (3) and pull a process from the ready queue (1) in first-in first-out fashion. The process that was previously running on that core is stored in the process pointer array (4), which each core has its own entry in the array based on core number. The previously running process is then put into the end of the ready queue (1). The next process to run is then stored in that core's process pointer array (4) entry. Lastly, the kernel lock (3) is released and the new process executes. This method of scheduling is repeated for every core each time they receive a timer interrupt.

In addition to architectural changes, the user land component also received several new complex drivers such as: Network File Sharing Daemon (12,850 lines of code), the e1000 network card driver (939 lines of code), and the associated *LWIP* network stack [131] (33,762 lines of code). From the size of these three components alone it can be seen that they require additional computing resources as they form the backbone for network connectivity.

Finally, ever more challenging operating system concepts in diversity [31], memory security [30], and utility virtual machines have been explored. Through all of this change, the round-robin scheduler remained in place. Thus, all performance improvements over this time came from architectural changes that resided below the round-robin code.

Therefore, a new scheduler was sought to better make use of these new realities and improve scheduling performance. This would not be without its own set of challenges, specifically, process affinity [132]. This is a uniquely multiprocessor problem based on the principle of cache coherency [112]. As in the Intel i7 architecture [133] used in this thesis, the cache is laid out so that each core has its own L1 and L2 cache and all cores share an L3 cache. When a process moves from one core to another, information about it is often shared through the L3 cache. This is a process known as snooping [133] by the other core through the L3 cache prior to any transfer. However, when the data is not propagated fast enough between cores, a cache miss can occur, which introduces a significant time penalty on execution. The processor often has to go out to main memory to find the needed data, which is a slower process than when it is available in the cache. Another issue was the introduction of the e1000 network card and its driver to the system, as the card itself by default will generate a hardware interrupt once every 256 nanoseconds or every 3.9 million cycles of core execution. Recall, from chapter 3 that the APIC timer is set to fire once every 34 million cycles. This means that the core that receives the network interrupts will have each of its time slices interrupted on average ~8.2 times. Resulting in any process running on that core receiving less execution time than had it been on another. Thus, breaking the principle of equal time slice fairness in round-robin scheduling.

To address the above issues and improve overall system performance a method of scheduling based on heat diffusion [34,134] was implemented. Several beneficial criteria exist for its selection: it uses a simple, fast, scalable algorithm involving only nearest neighbor communication [135], and global progress and convergence are guaranteed through well-established mathematical analysis. The algorithm has been shown, through simulation [136], to balance multiple independent load distributions over large-scale architectures [137], even with huge random load injections. Vector based extensions to the algorithm allow multiple resources (including process priority, interrupt routing, and CPU load) to be balanced concurrently [138].

6.1 Implementing diffusion

Previously, much of the supporting research in heat diffusion scheduling had been done on large interconnected computing systems [34,138]. In these studies, one or more nodes would quickly become burdened with very large workloads, which then diffused to other nodes via nearest neighbor communication. The same principles largely hold true for an individual SMP system, because a single compute node can now be considered a single processor core.

However, some adjustments are made to the load calculation to account for some traditional measures that cannot be used to determine heat in a localized system. Bandwidth can be eliminated as the cores have near instantaneous communication between each other via a crossbar [133]. All cores share the main memory of the system, which removes the need to account for memory usage, because no additional memory is available. Fortunately, new measures for load can be attributed to process priority, interrupt routing, and individual core load. Driver processes can be given priority by weighting them at different heat levels than those of a standard user process as they often times perform more complex tasks. In terms of routing interrupts, the core receiving them will by default run hotter than one that is not. Lastly, each process itself carries its own heat that adds load to a core. These three variables can be stored and summed to calculate the heat of any core running at any given time. A core can use this heat value to dynamically offload a process to another core with a lower workload.

Two components exist for mapping heat to a core and then diffusing work between them. The first is the static component, which is the initialization and assumptions made for interrupts, process priority, and individual process heat. The second component is the dynamic load-balancing component that moves processes between cores. These two pieces were built on top of the round-robin scheduler to provide the diffusion scheduler. The ready queue can continue to store all of the runnable processes that are present on the system by making two minor changes. The first is the addition an identifier in the process structure for each process that maps it to the core it is bound to. This allows for individual processes to be tracked across cores for heat calculations and scheduled by their assigned core. The second modification is the replacement of the *qget()* function with the *qremove()* function for scheduling the next process. Where *qget()* returns values from the queue in first-in first-out fashion, the *qremove()* function allows the ready queue to be searched by each core via their core ID, which maps to the new identifier in the process structure. The code for this process can be seen below.

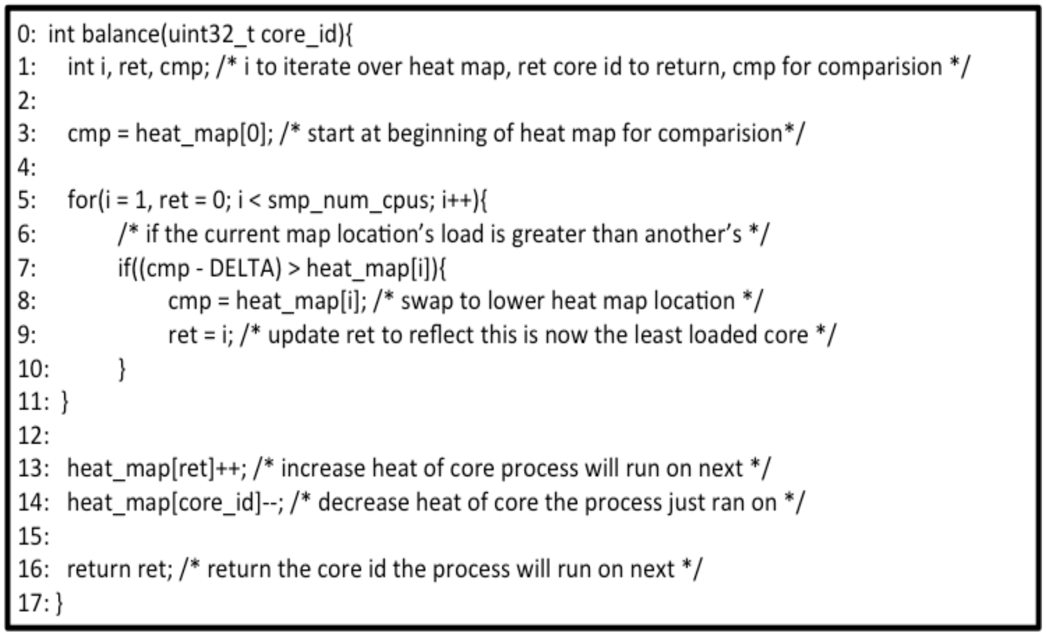
```
static int assigned_core(void* proc, const void* core_id){

    /* return first found process assigned to this core */
    return ((Proc_t*)proc)->core == (*(uint32_t*)core_id);
}

Proc_t *ksched_schedule() {

    Proc_t *next;  /* Next process to run   */
    uint32_t core = this_cpu(); /* Core requesting next process */

    next = (Proc_t*)qremove(readyq, &assigned_core, &core); /* Get next process to run */

    - - - - - - - - - - - - - - - - - -
    - - - - - - - - - - - - - - - - - -
    /* Other non-diffusion scheduling tasks */
    - - - - - - - - - - - - - - - - - -
    - - - - - - - - - - - - - - - - - -

    return next;
}
```

Figure 49 - Code to Schedule Next Process

ksched_schedule() is called for every timer interrupt to retrieve the next process to run for a specific core. It relies on reading the core's local APIC ID (line 9) to pass to *qremove()* (line 11) along with the global pointer to the ready queue, and the helper function *assigned_core()*. The sole purpose of the helper function is to return the pointer to the first found process in the ready queue that has been mapped to that core. The process is then removed from the ready queue by the *qremove()* function. Lastly, not seen here is the previously running process is added back to the end of the ready queue.

To initialize the heat map an array of integers that is of length corresponding to the number of cores present is created (8 cores on Dell 9010). All of the cores start with an initial heat value of zero. Cores that handle hardware interrupts can then be assigned heat values of 0, 10, 100, or 1000. These heat values move with the interrupt they are assigned to. Driver processes, like the e1000 driver are assigned heat values of 1 or 10 and move with them as well. All other user space processes are assigned a heat value of 1. Lastly, when a new process is created it is always assigned to the core that created it through the fork system call.

The movement of processes to a new core occurs through the dynamic load balancing code, which is called during a timer interrupt, but before the next process is retrieved through *ksched_schedule()*. To ease explanation of how this code works, the base case of all processes having a heat of 1 and no interrupt heat assignment is given below.



```
0:  int balance(uint32_t core_id){
1:    int i, ret, cmp; /* i to iterate over heat map, ret core id to return, cmp for comparision */
2:
3:    cmp = heat_map[0]; /* start at beginning of heat map for comparision*/
4:
5:    for(i = 1, ret = 0; i < smp_num_cpus; i++){
6:          /* if the current map location's load is greater than another's */
7:          if((cmp - DELTA) > heat_map[i]){
8:                cmp = heat_map[i]; /* swap to lower heat map location */
9:                ret = i; /* update ret to reflect this is now the least loaded core */
10:         }
11: }
12:
13:   heat_map[ret]++; /* increase heat of core process will run on next */
14:   heat_map[core_id]--; /* decrease heat of core the process just ran on */
15:
16:   return ret; /* return the core id the process will run on next */
17: }
```

Figure 50 - Dynamic Load-Balancing Code

The *balance()* function returns the ID of the core the process will run on the next time it is scheduled. The ID returned by it is stored in the process identifier that was added to the process structure. This is accomplished by assigning the heat value of core zero to a comparator (line 3). Next, the for loop (line 5) iterates over the remaining values stored in the heat map. Along the way, if the current comparator's heat is greater than another core's heat, it will then swap the lower heat into the comparator (lines 7-8). Furthermore, the ID of the core with the lower heat is then stored in the variable *ret* (line 9). Upon completion of the loop the core with the least heat is increased by 1 (line 13). The core the process just ran on has its heat decreased by 1 (line 14).

There are a number of unique ways that the base case can be expanded upon. For one, the process structure can also be passed into the *balance()* function. This allows the routine to check an individual processes heat for comparison and swapping. So if a driver process with a heat of 10 was being considered for movement, the left half of the *if* statement (line 7) is modified to subtract that processes heat from the comparator (cmp – DELTA – *process_assigned_heat*). This also means that a similar change is made to the final addition and subtractions (lines 13 – 14) such that the process heat is accounted for correctly (heat_map[ret] += *process_assigned_heat*, etc). This is just one type of modification that can be made, but other possibilities exist to find the optimal load-balancing solution.

Now one facet that has not been discussed is the *DELTA* (line 7) value used in the comparator portion of the balancing routine. This value exists due to the process affinity problem and only was discovered through testing. The primary purpose is to eliminate cache thrashing across cores in situations when low loads exist. A good explanation of what happens without a delta variable is when there are 10 processes and 8 cores. In this situation the first 8 processes will be scheduled on one of the 8 cores. The last two processes after each scheduling round will be swapped dynamically to one of the other six. Every time one of these swaps occurs, the next run of that process will result in cache misses and large performance penalties. Testing with low values for delta was performed to find the optimal number, which is 2.

6.2 Benchmarks and Analysis

To eliminate as many external factors that could impact performance, experimentation was completed on the kernel only version of the system. This removes the slow-down generated by the presence of the hypervisor and the virtual APIC settings. The memory benchmark is well suited for the evaluation of the diffusion scheduler as a single process spawns 100 additional processes. This results in one core having a high initial load that it then transfers to the other cores.

The initial run of the diffusion scheduler used the code seen in Figure 50 minus the *DELTA* variable. The AIM9 test suite, which runs as a single process, illustrates the problem of process affinity as cache thrashing occurs and results in high overheads. Once, the issue was noticed, a *DELTA* of one and two are used in all further testing. Additional configurations of the scheduler include drivers with heat values of 10, all drivers pegged to a core, and hardware interrupts of heat 10, 100, or 1000. The results of the varying methods and the round-robin scheduler performance are seen in Table 6.

Scheduler Configurations	Cycles Memory	Cycles AIM9	Cycles Total	Memory Time (s)	AIM9 Time (s)	Total Time (s)
Round-Robin Scheduler	2.9574E+11	1.4455E+11	4.4029E+11	86.98	42.54	129.50
Diffusion – All Processes 1	2.9185E+11	2.2333E+11	5.1519E+11	85.84	65.69	151.53
Diffusion – All Processes 1, Delta 1	2.8910E+11	1.7981E+11	4.6891E+11	85.03	52.88	137.91
Diffusion – All Processes 1, Delta 2	2.9378E+11	1.4500E+11	4.3878E+11	86.41	42.65	129.05
Diffusion – All Processes 1, Delta 1, Peg Drivers	2.9111E+11	1.7725E+11	4.6836E+11	85.62	52.13	137.75
Diffusion – All Processes 1, Delta 2, Peg Drivers	2.9316E+11	1.4593E+11	4.3909E+11	86.22	42.92	129.14
Diffusion – User Processes 1, Delta 1, Peg Drivers 10	2.9391E+11	2.0721E+11	5.0112E+11	86.44	60.94	147.39
Diffusion – User Processes 1, Delta 2, Peg Drivers 10	2.9420E+11	1.4323E+11	4.3743E+11	86.53	42.13	128.66
Diffusion - User Processes 1, Delta 2, Interrupts 10, Peg Drivers 1	2.8634E+11	1.4769E+11	4.3402E+11	84.22	43.44	127.65
Diffusion – User Processes 1, Delta 2, Interrupts 100, Peg Drivers 1	2.7283E+11	1.5956E+11	4.3239E+11	80.24	46.93	127.17
Diffusion – User Processes 1, Delta 2, Interrupts 1000, Peg Drivers 1	2.8835E+11	1.6225E+11	4.5060E+11	84.81	47.72	132.53

Table 6 - Scheduler Performance Characterization

When reviewing Table 6, it is important to examine the individual tests first when evaluating the new scheduler, as the memory test benefits from improvements to multi-process execution. Whereas the AIM9 test suite sees performance gains when single process execution is sped up through an enhancement. Adding the times it takes to complete both tests together provides an overall measure of performance, but may miss potential impacts to either form of execution.

For example, on the surface the diffusive scheduler without the *DELTA* variable overall performs 15.68% worse than the round-robin scheduler. This is solely because of a 42.78% performance penalty taken during single process execution of the AIM9 suite due to cache thrashing. In fact, multi-process execution during memory testing is improved by 1.32%, which almost certainly is impacted by cache thrashing to some degree. Thus, all testing is performed with a *DELTA* present. As noticeable performances gains were only shown using a *DELTA* of two, the following discussions will only be in regards to that setting.

The diffusive scheduler performs equivalently to the round-robin scheduler once process affinity has been accounted for. Further exploration of increasing process affinity was explored by pegging drivers to a single core. This alone did not result in any performance gains. In attempt to isolate drivers further from user tasks, their heat value was increased from 1 to 10. As this resulted in a .52% speedup for memory, a .97% speedup for AIM9, and an overall speedup of .70%.

Unfortunately, these results provide less than a 1% margin for scheduling improvement.

The last variable that impacts normal core execution is hardware interrupts. In this approach the core receiving hardware interrupts from the I/O APIC will be assigned a heat of 10, 100, or 1,000. Drivers will continued to be pegged to cores to improve their individual process affinity as testing showed a marginal benefit in doing so. The graphed results of increasing heat can be seen in Figure 51 below.

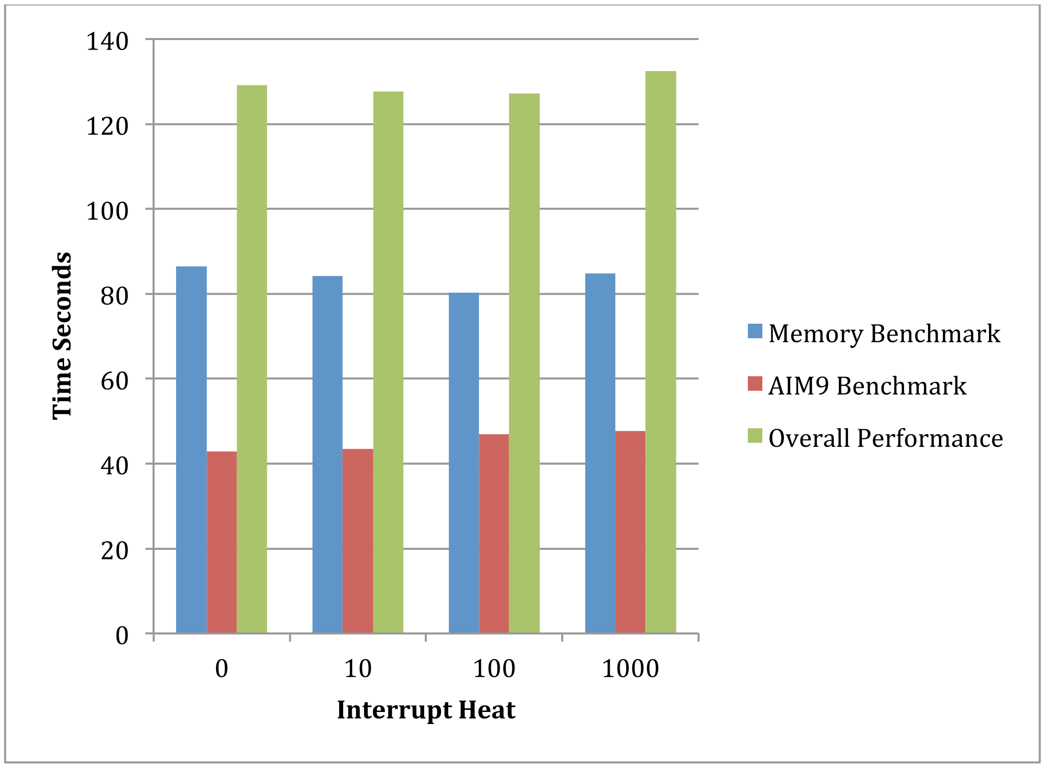


Figure 51 - Diffusion Performance as Interrupt Heat Rises

Looking at the graph it can be seen that from 0 to 100, memory and overall performance improve, while AIM9 performance decreases as interrupt heat rises.

Going from 100 to 1000 heat causes a decrease in all three categories. However, the setting of this variable at 100 has an effect of improving multi-process performance by 8.06%, but decreasing single process performance by 9.81%, which amounts to an overall performance increase of 1.82%. These results are noteworthy as they clearly demonstrate that the heat diffusion algorithm has a marked scheduling improvement in environments with heavy workloads.

With all of the changes to the scheduler, it could be expected that there was a large addition of lines of code, which would increase the attack surface. However, this was not the case, as repurposing pieces of the round-robin scheduler kept the need for additional lines of code low. To add the minimal amount of support for diffusion, 29 lines of code were needed for the configuration presented in Figure 50. To go to the full interrupt and driver pegging setup requires only an additional 23 lines of code, which brings the total to 52 lines.

6.3 Summary

This chapter has shown that methods like heat diffusion can provide performance improvements for multi-process environments, which will be critical for the expansion of the UVM work to the cloud. As year after year has shown that cloud services continue to expand and core count on processors continue to rise. However, deployment of diffusive schedulers should initially be limited to multi-process environments, because initial testing has shown a decrease in performance for single process tasks.

Chapter 7 – Conclusions and Future Work

This thesis examined the possibility of increasing *attacker workload* and mitigating zero-day threats through the redesign of the standard virtualization architecture. An approach based on shrinking the kernel and user components to be encapsulated into lightweight Utility Virtual Machines was implemented in conjunction with scheduling technologies to balance load. Exceeding the expectations of its original design goals, the UVM technology provides the desired virtualization sandboxing and performed on par or better than monolithic virtualization performance, all while achieving its goal of reducing the overarching *attack surface*.

7.1 Conclusions

Observations about *attack surface* for the various UVMs were made in chapters 4 and 5. They have been summarized here in Table 7 for discussion purposes, with reductions marked with ↓ and gains marked with ↑.

	Lines of Code	Net Line Addition/ Reduction	Percent Addition/ Reduction	Estimated Number of Potential Vulnerabilities [27]	Net Potential Vulnerability Increase/ Decrease
Hypervisor	2,489	X	0.0%	0.2240	X
Micro-Kernel	2,904	X	0.0%	0.2614	X
Micro-Kernel user space	50,592	X	0.0%	4.5533	X
UVM Hypervisor	2,706	217 ↑	8.4% ↑	0.2435	0.0195 ↑
Keyboard/VGA UVM Micro-Kernel	2,072	832 ↓	33.4% ↓	0.1865	0.0749↓
Keyboard/VGA UVM User Space	1,210	49382 ↓	190.7% ↓	0.1089	4.4444↓
Shell UVM Micro-Kernel	2,311	593 ↓	22.7% ↓	0.2080	0.0534↓
Shell UVM User Space	2,600	47992 ↓	180.4% ↓	0.2340	4.3193↓
Network UVM Micro-Kernel	2,492	412 ↓	15.3% ↓	0.2243	0.0371 ↓
Network UVM User Space	46,715	3877 ↓	8.0% ↓	4.2044	0.3489 ↓

Table 7 - Summary of Attack Surface Reduction

The biggest source of concern is user space code, which using Pandey et al. approach of estimating .09 defects per 1,000 lines of code, shows that it contains 4.5533 defects. By segmenting this code across UVMs, an attacker has a significantly reduced *attack surface*, which reduces the number of bugs and ROP gadgets in all cases. Furthermore, the micro-kernels residing below in all of these instances have also seen significant *attack surface* reductions. Lastly, one-way communication channels are tightly enforced and can only initiated by select UVM processes. Thus, if an attacker did gain a foothold in either the network or Keyboard/VGA space they would have no means to compromise any other driver UVM present. The shell UVM would only be impacted if it were active and no compromise could occur if it was dormant.

Working on network time scales the driver UVMs could be refreshed prior to the user logging into the shell UVM, which would mitigate much of this risk.

However, the hypervisor did have to grow a small amount to support these code size reductions. This was deemed an acceptable risk, as the hypervisor is located below the EPT created semantic gap, which offers it considerable protection. Moreover, the hypervisor operates solely as an intermediary between UVM guests. Nothing the hypervisor introspects is ever executed and mappings only subsist while a core is within the hypervisor. In following this standard the hypervisor is protected in the event a guest is compromised.

Lastly, beyond the reductions the commonly accepted thought of *attack surface* has completely changed from the standard monolithic approach. All that remains is the hypervisor; the low hanging fruits, which are the guests, have been completely sandboxed through hardware enforced isolation. An attacker can no longer compromise the network and have immediate access to the shell. Nor could a malicious user compromise the shell and directly inject code into a driver. This by itself greatly increases the workload of any attacker.

7.2 Future Work

This work provides a stepping off point for multiple continued research efforts. Foremost being the transitioning of UVM concepts to larger scale industry projects, such as Intel supporting EPT switching for Xen hypervisor guests [139]. This technology is designed to increase the performance of inter-VM communication, which is provided through VMCALLs in Xen and is a cornerstone of the UVM work. Coupling a network UVM with this technology would allow for far greater transfer speeds while maintaining network isolation.

Another corporation that is also pursuing similar technology is Docker, which acquired Unikernel Systems in January 2016 [140]. Their express intention of this acquisition is to leverage unikernels to build VM containers that perform small roles. This planned technology is almost a one for one recreation of UVMs, but with a unikernel replacing the role of the micro-kernel.

One area that needs more attention across type 1, type 2, and UVM architectures is the creation of guests. This is and largely remains a manually intensive process of either programming or compiling what is desired into the guest. Any future work in this field should focus on user-friendly alternatives that can be used to quickly create a new containerized VM.

In terms of diffusive scheduling the surface has only been scratched in terms of its performance benefits. As this technology can be combined, with the already existing faculties to run across multiple instances in a cloud [25] and the resilient inter-process communication mechanisms [141]. This will also reintroduce a greater number of variables to the scheduling algorithm, such as bandwidth, memory usage, and other related distributed computing measures. Combining with greater computing resources will only improve multiple process performance demonstrated in this thesis further. Since it has already been proven that diffusion can be extended *"to operate across multiple processors using only local nearest neighbor communication with well understood global convergence and termination properties"* [142].

This also opens the possibility of porting the diffusive scheduler to the hypervisor. Which would allow the ability to balance multiple user UVMs across many servers. In turn this creates new and interesting challenges in determining what setups should be used across this distributed network for driver and user UVMs.

Appendix A – Stage One Bootloader Code

```
### boot1.S
### This file is placed at the start of a *slice*, and is called by
### the MBR's 512-byte block. For compatibility reasons, this file is
### also exactly 512 bytes.
###
### Copyright (c) 2011 Morgon Kanter
### All rights reserved.
### Redistribution and use in source and binary forms are freely
### permitted provided that the above copyright notice and this
### paragraph and the following disclaimer are duplicated in all
### such forms.
###
### This software is provided "AS IS" and without any express or
### implied warranties, including, without limitation, the implied
### warranties of merchantability and fitness for a particular
### purpose.

### Make sure this is linked with -Ttext=0x7c00, which is the address
### that both the MBR gets loaded in from the BIOS, and the address
### where the actual MBR loads this in.
###
### Just as when we're loaded up by the MBR, the drive entry will be
### placed in %dl and our slice from the main table will be placed in
### %si.
###
```

```
### Note if you go to disassemble this with objdump, the output will
### almost certainly be wrong for some sections. This is because we
### mix 16-, 32-, and 64-bit code in the same file. This is all very
### clearly delimited by the .code16, .code32, and .code64 designators
### given at those points in this file.

    ## Global symbols
    .globl gdt_long

    ## Memory locations
    .set BOOT_STACK,0x7bf8      # location of the stack "bottom"... the stack grows
                        # down so this is the highest addr of the stack. It
                        # would hit the partition table if it overran.
    .set MEM_LOADED,0x7c00      # Where we're loaded
    .set MEM_DLBL,0x7e00        # Start of disklabel
    .set DLBL_BBASE,0x7e28      # "bbase" in disklabel
    .set PART_TBL,0x7be         # Partition table for the disk
    .set NUM_PART,0x88          # Offset to number of partitions
    ## Place to put the available memory. Be sure not to use the
    ## first low 0x500 bytes of this (it's the IVT etc). This is a
    ## many-entry table, with 24-byte entries, each with the
    ## following format:
    ## First qword = Base address
    ## Second qword = Length of "region" (if 0, ignore the entry).
    ## Next dword = Region "type"
    ##   * Type 1: Usable (normal) RAM
    ##   * Type 2: Reserved - unusable
    ##   * Type 3: ACPI reclaimable memory
    ##   * Type 4: ACPI NVS memory
    ##   * Type 5: Area containing bad memory
    ## Next dword = ACPI 3.0 extended attributes bitfield (if 24
    ##           bytes are returned, but we increment 24 bytes
    ##           regardless).
    ##   * Bit 0 of the extended attributes field indicates if the
    ##     entire entry should be ignored.
    ##   * Bit 1 of the extended attributes field indicates if the
    ##     entry is non-volatile (whatever that means).
    ## This information was taken from:
    ## http://wiki.osdev.org/Detecting_Memory_%28x86%29
    .set MEM_ENTRIES,0x802        # Number of mem table entries
    .set MEMTABLE,0x804           # Mem table

    ## Basic page table locations for baby's first page tables
    .set PML4T,0x1000             # Page map level 4 table
    .set PDPT,0x2000        # Page directory pointer table
    .set PDT,0x3000         # Page directory table
    .set PT,0x4000          # Page table

    ## Constants
    .set PARTSIZE,0x10            # Size of partition entry in label
    .set MAGIC,0xaa55             # Magic: bootable
    .set EMAGIC,0x534D4150        # e820 Magic
    .set SEL_CODE,0x08            # Code selector
    .set SEL_LCODE,0x08           # Code selector, long mode
```

```
	.globl start			# Entry point
	.code16				# Real mode, for now
start:
	xorw %ax,%ax		# Zero
	movw %ax,%es		# Address
	movw %ax,%ds		#  data
	movw %ax,%ss		# Set up
	movw $MEM_LOADED,%sp  #  stack


mmap.0:
	movw $MEMTABLE-24,%di	# Memory table
	xorl %ebx,%ebx		# Zero
	pushw %si		# Save %si
	movw %ax,%si		# Zero
mmap:
	addw $24,%di		# Set next map entry
	movl $0xe820,%eax		# Get memory map
	movl $24,%ecx		# Buffer size
	movl $EMAGIC,%edx		# Signature
	int $0x15			# Get memory map
	jc mmap.1		# Method #1 of marking finished
	cmpl $EMAGIC,%eax		# Error?
	jne mem_err		# Yes
	incw %si			# Total number of entries
	cmpl $0x0,%ebx		# Method #2 of marking finished
	jne mmap		# Not finished
```

```
        ## Now that we have the memory map set up, let's see about

        ## entering protected mode (in the 32-bit part of file).

mmap.1:

        movw %si,MEM_ENTRIES        # Number of memtable entries

        popw %si            # Restore %si

        jmp postmem


mem_err:

        movw $msg_mem,%si           # "Memory detection error"

        jmp putstr          # Error out


### Output an ASCIZ string to the console via the BIOS.

putstr.0:

        movw $0x7,%bx               # Page:attribute

        movb $0xe,%ah       # BIOS: Display

        int $0x10                   #  character

putstr:

        lodsb                       # Get character

        testb %al,%al       # End of string?

        jnz putstr.0        # No

putstr.1:

        jmp putstr.1        # Await reset.


msg_mem:.asciz "Memory detection error"


### The GDT. This table looks retarded:

###  -----------------------------------------------------------------------
### |0                 15|16                      31|
### -------------------------------------------------------------------------
### |       Limit 0:15       |            Base 0:15            |
### -------------------------------------------------------------------------
### |32      39|40      47|48      51|52      55|56      63|
### -------------------------------------------------------------------------
### |Base 16:23|Access Byte|Limit 16:19|Flags|Base 24:31|
### -------------------------------------------------------------------------
###
### The bit arithmetic here is horrible. The access byte and flags go
### in backwards from what you'd expect (they are defined bitwise
### starting from 7 and going to 0, so when you specify it you have to
### reverse so it starts at 0 and goes to 7...).
gdt:
    .word 0x0, 0x0, 0x0, 0x0                # Null entry
    .word 0xffff, 0x0, 0x9a00, 0x00cf       # Code entry (SEL_CODE)
    .word 0xffff, 0x0, 0x9200, 0x00cf       # Data entry
    .word 0xffff, 0x0, 0xfa00, 0x00cf       # User code entry
    .word 0xffff, 0x0, 0xf200, 0x00cf       # User data entry
gdt_48:
    .word .-gdt-1
    .long gdt
### If we're short on space, we can do the following by modifying the
### memory at gdt instead of duplicating it all and save about 20
### bytes in the process.
### NOTE: If this expands/contracts, update GDT_SIZE in constants.h
gdt_long:
    .word 0x0, 0x0, 0x0, 0x0                # Null entry
    .word 0xffff, 0x0, 0x9a00, 0x00af       # Code entry (SEL_LCODE)
    .word 0xffff, 0x0, 0x9200, 0x00cf       # Data entry
    .word 0xffff, 0x0, 0xfa00, 0x00af       # User code entry
    .word 0xffff, 0x0, 0xf200, 0x00cf       # User data entry
    .word 0x0067, 0x6400, 0x8900, 0x0010 # TSS, depends on MEM_TSS_BASE
    .word 0x0, 0x0, 0x0, 0x0                # TSS Entry Part 2
gdtlong_48:
    .word .-gdt_long-1
    .long gdt_long


### Do things that we need to enter protected mode.
postmem:
    ## Set video mode to 80x25 for basic console.
    mov $3,%ax              # Video mode 80x25x16
    int $0x10                       # Interrupt (set video mode)
    mov $0x1003,%ax                 # Toggle blinking
    mov $0,%bx              # Blinking disabled
    int $0x10                       # Interrupt (set video mode)
loadgdt:
    lgdt gdt_48             # Load gdt
    ## Don't set up the selectors yet, they push every data
    ## reference 16 bytes higher because, since we aren't yet in
    ## protected mode, they aren't "selectors", they are still
    ## just real-mode segment offsets!
loadidt:
    ## We don't have nearly the amount of space here to set up a
```

```
	## desirable interrupt table! Defer it for when we go to the
	## C boot code.
loadtss:
	## Not really interesting to set up the TSS here either,
	## because we have no interrupts. Let the C boot code do this,
	## when it handles interrupts.
protected.0:
	## Now we can set up what will become the selectors, since we
	## have no more data references to make.
	movw $0x10,%ax			# Set segment selectors
	movw %ax,%ds		#  data
	movw %ax,%ss		#  stack
	movw %ax,%es		#  es
	movw %ax,%fs		#  fs
	movw %ax,%gs		#  gs
	cli				# Disable interrupts
	movl %cr0,%eax		# Control Register 0 to %eax
	orb $0x1,%al		# Set the lowest bit
	movl %eax,%cr0		# %eax to Control Register 0
	ljmp $SEL_CODE,$protected	# Protected mode
	.code32
protected:
	## Now we're in protected mode. Set up long mode.
	## Have to re-set the segment selectors here, so they are
	## considered 32 bit (otherwise, any time we set data it will
	## fail). I'm not sure if this is a bug in QEMU or if it works
	## this way on bare hardware as well.
```

```
    movw $0x10,%ax              # Set segment selectors
    movw %ax,%ds        #  data
    movw %ax,%ss        #  stack
    movw %ax,%es        #  es
    movw %ax,%fs        #  fs
    movw %ax,%gs        #  gs
    movl %cr4,%eax      # Control Register 4 to %eax
    bts $5,%eax         # Set PAE
    bts $7,%eax         # Set pages golbal enable
    movl %eax,%cr4      # %eax to Control Register 4
    movl $0xc0000080,%ecx       # EFER register
    rdmsr                       # EFER to %eax
    bts $8,%eax         # Set IA-32e (long mode)
    bts $11,%eax        # Allow No-EXecute bit
    wrmsr                       # %eax to EFER
paging.0:
    ## Set up 64-bit paging -- required before we're actually in
    ## long mode. The first 1 MB will be identity-mapped.
    movl $PML4T,%edi            # Page-map level 4 table
    movl %edi,%cr3      #  in base table
    movl $4096,%ecx             # 4 kb*4 count
    xorl %eax,%eax      # Zero
    rep                         # Clear
    stosl                       #  PML4T
paging.1:
    ## Set the first entry of each page table level to point to
    ## the next level. This handles the pointing of PML4T to PDPT,
    ## PDPT to PDT, and PDT to PT.
    movl $3,%ecx            # Count
    movl %cr3,%edi          # Destination (Page tables)
    movl $0x1000,%eax               # Increment
    movl $0x2003,%ebx               # Page present, read/writable
paging.2:
    movl %ebx,(%edi)                # Point to next paging level
    addl %eax,%ebx          # Next level
    addl %eax,%edi          # Next level
    loop paging.2           # For each page level
paging.3:
    ## Set each page of the PT level to be read/writable. Note
    ## that %edi now points to the PT level thanks to the loop.
    movl $0x00000103,%ebx           # Page present, read/writable
    movl $512,%ecx          # Number of pages in PT level
paging.4:
    movl %ebx,(%edi)                # Set present, read/writable
    addl $0x1000,%ebx               # Next 4096 bytes
    addl $8,%edi            # Next page entry
    loop paging.4           # For each page entry
paging.5:


Recursive.Paging:                   #Recursive paging is set up here
    movl $PML4T, %eax               #Move phys address of PML4T into eax
    addl $0x1000, %eax              #Add Page Size to move to end of PML4T
    subl $0x8, %eax                 #Subtract to move to last entry of PML4T
```

```
        movl $PML4T, %ebx               #Move Phys address of PML4T into ebx

        addl $0x3, %ebx                 #Add page permission bits Page Present, read/write


        mov %ebx, (%eax)                #Do the mapping into last entry of PML4T


        ##Turn on paging

        movl %cr0,%eax          # Control Register 4 to %eax

        orl $0x80000000,%eax            # Enable paging

        movl %eax,%cr0          # %eax to Control Register 4

        lgdt gdtlong_48         # Load long-mode gdt

        ljmp $SEL_LCODE,$long           # Enter long mode

        .code64

long:

        ##  Welcome to long mode.

        ## As above, reload the segment registers.

        movw $0x10,%ax                  # Set segment selectors

        movw %ax,%ds            #  data

        movw %ax,%ss            #  stack

        movw %ax,%es            #  es

        movw %ax,%fs            #  fs

        movw %ax,%gs            #  gs

        ## Enable SSE (required for stuff like clang varargs), and other

        ## things that should be turned on.

        movq %cr0,%rax                  # Control register 4 to %eax

        bts $1,%rax             # Set MP bit

        btr $2,%rax             # Clear EM bit

        bts $5,%rax             # Allow native (new) FPU error reporting
```

```
    movq %rax,%cr0              # %eax to control register 0
    movq %cr4,%rax              # Control register 4 to %eax
    bts $9,%eax         # Set OSFXSR bit
    bts $10,%eax        # Set OSXMMEXCPT bit
    movq %rax,%cr4              # %eax to control register 4
    ## Now let's return to C and be done with this.
    ## The boot block base is loaded in as part of the disklabel,
    ## and we know the disklabel starts right after us in the code.
    ## So, find the boot block base and jump there.
    movq DLBL_BBASE,%rbx        # Boot block base (bytes)
    addq $MEM_DLBL,%rbx         # Add start of disk label
    subq $0x200,%rbx            # Subtract the first 512 bytes (included in
                                #  disklabel for some reason)
    movq $BOOT_STACK,%rsp       # Reset the stack pointer to a value clang likes
    jmp *%rbx           # Into the C entry point

### Fill the rest of the 512 bytes with NOP and make bootable.
    .org 0x1FE,0x90     # Fill the rest with NOPs
    .word MAGIC         # Bootable magic
```

Appendix B – Register Layouts

Control Register 0	
Bit	Label & Description
0	PE – If 1, enables 32-bit protected mode
1	MP – Controls the behavior of *wait* and *fwait* instructions
2	EM – Allows saving
3	TS – Allows saving floating point context on hardware switches
4	ET – Reserved for us in older Intel processors
5	NE – If 1, enables internal x87 floating point error reporting
16	WP – If 1, ring 0 code can write to pages marked read only
18	AM – If 1, processor checks alignment on certain operations
29	NW – If 1, disables write-back caching
30	CD – If 1, disables memory caches
31	PG – If 1, enables paging

Control Register 3	
Bit	Label & Description
0-2	Ignored
3	PWT –
4	PCD –
5-11	Ignored
12-48	Address of PML4 Table
49-63	Reserved for future use

Control Register 4	
Bit	Label & Description
0	VME – If 1, enables virtual interrupts in virtual-8086 mode
1	PVI – If 1, enables virtual interrupts in protected mode
2	TSD – If 1, only kernel mode can read the hardware timestamp
3	DE – Controls the usage of the debug registers
4	PSE – If 1, enables 4MB pages --- If 0, enables 4KB pages
5	PAE – If 1, enables paging to produce physical addresses greater than 32 bits
6	MCE – If 1, enables machine check exceptions
7	PGE – If 1, allows global pages with special caching properties
8	PCE – If 1, allows user-land code to use performance counters
9	OSFXSR – If 1, enables *fxsave* and *fxstor* instructions
10	OSXMMEXCPT – Controls operation of SSE instructions
13	VMXE – If 1, enables VMX instructions
14	SMXE – If 1, enables supervisor mode
16	FSGSBASE – Controls *rdsfbase*, *rdgsbase*, *wrfsbase*, and *rfsgsbase* instructions
17	PCIDE – If 1, enables process-context identifiers
18	OSXSAVE - Controls operation of *xsave*, *xstor*, and *xgetbv* instructions
20	SMEP – If 1, prevents kernel mode from executing user-mode code
21	SMAP – If 1, prevents kernel mode from accessing user-mode data
22	PKE – IF 1, enables x86-64 paging to associate linear addresses with protection keys

Extended Feature Enable Register MSR	
Bit	Label & Description
0	SCE – If 1, enable fast system call instructions
8	LME – If 1, enables 64-bit long mode
10	LMA – Indicates if long mode is active
11	NXE – IF 1, enables the use of the no-execute bit in page tables

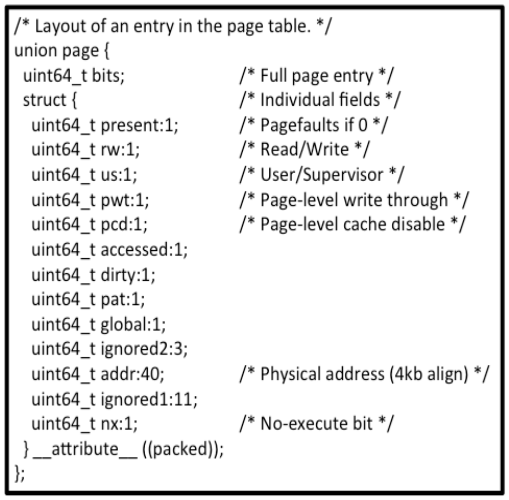
APPENDIX C – Macros for paging

```
/* These four functions make use of the self-referencing page table trick
   to build a mapping to a given paging structure entry - either a pml4t entry,
   a pdpt entry, a pd entry, or a pt entry. */
#define PTE2vaddr(pml4te, pdpte, pde, pte) ((uint64_t*)(0xffffff8000000000 | (((uint64_t)((pml4te) & 0x1ff)) << 30) \
                                                     | (((uint64_t)((pdpte) & 0x1ff)) << 21) \
                                                     | (((uint64_t)((pde) & 0x1ff)) << 12) \
                                                     | (((uint64_t)((pte) & 0xfff))*8)))

#define PDE2vaddr(pml4te, pdpte, pde)      ((uint64_t*)(0xffffffffc0000000 | (((uint64_t)((pml4te) & 0x1ff)) << 21) \
                                                     | (((uint64_t)((pdpte) & 0x1ff)) << 12) \
                                                     | (((uint64_t)((pde) & 0xfff)) * 8)))

#define PDPTE2vaddr(pml4te,pdpte)          ((uint64_t*)(0xffffffffffe00000 | (((uint64_t)((pml4te) & 0x1ff)) << 12) \
                                                     | (((uint64_t)((pdpte) & 0xfff)) * 8)))

#define PML4TE2vaddr(pml4te)               ((uint64_t*)(0xfffffffffffff000 | (((uint64_t)((pml4te) & 0xfff)) * 8)))
```

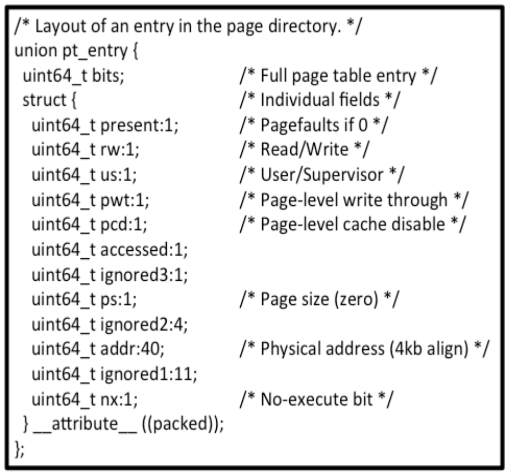
```
/* These macros take a virtual address, and return the parts used for indexing
 * into the various page table levels.
 */
#define virt2pml4t(addr)      ((((uint64_t)(addr)) >> 39) & 0x1FF)
#define virt2pdpt(addr)       ((((uint64_t)(addr)) >> 30) & 0x1FF)
#define virt2pd(addr)         ((((uint64_t)(addr)) >> 21) & 0x1FF)
#define virt2pt(addr)         ((((uint64_t)(addr)) >> 12) & 0x1FF)
#define virt2pg(addr)         ((((uint64_t)(addr))      ) & 0xFFF)
```

```
/* Put a physical address in the "addr" slot of the table.
 * This is equivalent to setting table->bits to the actual address, by the way,
 * so long as you set table->bits first.
 */
#define ADDR2TABLE(addr) ((uint64_t)(addr) >> 12)
/* And vice versa */
#define TABLE2ADDR(addr) (((uint64_t)(addr)) << 12)
```

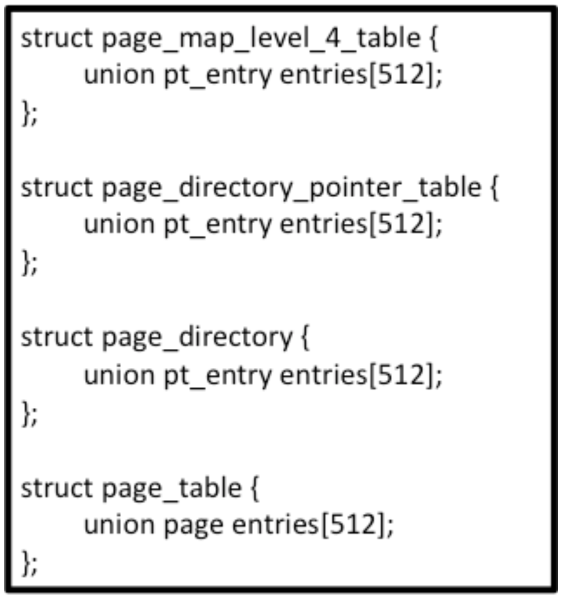
Appendix D – Paging Structures & Frame Array Initialization C Code



```
/* Layout of an entry in the page table. */
union page {
 uint64_t bits;                /* Full page entry */
 struct {                      /* Individual fields */
  uint64_t present:1;          /* Pagefaults if 0 */
  uint64_t rw:1;               /* Read/Write */
  uint64_t us:1;               /* User/Supervisor */
  uint64_t pwt:1;              /* Page-level write through */
  uint64_t pcd:1;              /* Page-level cache disable */
  uint64_t accessed:1;
  uint64_t dirty:1;
  uint64_t pat:1;
  uint64_t global:1;
  uint64_t ignored2:3;
  uint64_t addr:40;            /* Physical address (4kb align) */
  uint64_t ignored1:11;
  uint64_t nx:1;               /* No-execute bit */
 } __attribute__ ((packed));
};
```



```
/* Layout of an entry in the page directory. */
union pt_entry {
 uint64_t bits;                /* Full page table entry */
 struct {                      /* Individual fields */
  uint64_t present:1;          /* Pagefaults if 0 */
  uint64_t rw:1;               /* Read/Write */
  uint64_t us:1;               /* User/Supervisor */
  uint64_t pwt:1;              /* Page-level write through */
  uint64_t pcd:1;              /* Page-level cache disable */
  uint64_t accessed:1;
  uint64_t ignored3:1;
  uint64_t ps:1;               /* Page size (zero) */
  uint64_t ignored2:4;
  uint64_t addr:40;            /* Physical address (4kb align) */
  uint64_t ignored1:11;
  uint64_t nx:1;               /* No-execute bit */
 } __attribute__ ((packed));
};
```



```
struct page_map_level_4_table {
        union pt_entry entries[512];
};

struct page_directory_pointer_table {
        union pt_entry entries[512];
};

struct page_directory {
        union pt_entry entries[512];
};

struct page_table {
        union page entries[512];
};
```

```
void setup_table( void *phys, void *vaddr, uint64_t flags) {

 union pt_entry *entry = (union pt_entry*)vaddr;

 kmemset(entry, 0, sizeof(union pt_entry));
 entry->present = 1;

 entry->rw = (flags & PG_RW) ? 1 : 0;

 entry->nx = (flags & PG_NX) ? 1 : 0;

 entry->us = (flags & PG_USER) ? 1 : 0;

 ((union page*)entry)->global = ( flags & PG_GLOBAL ) ? 1 : 0;

 entry->addr = ADDR2TABLE((uint64_t)phys);

 return;

}




void frame_array_init(void) {


 struct memmap *block = (struct memmap *)0x804;

 struct memmap *oldblock;

 uint16_t *numblocks = (uint16_t *)x802;

 uint64_t memory_present, frame_position;

 int framearray_pages;

 int i, j;


 struct page_map_level_4_table *pml4t;

 struct page_directory_pointer_table *pdpt;

 struct page_directory *pd;

 struct page_table *pt;

 union page *page;


 int pml4t_idx;
```

```
  int pdpt_idx;

  int pd_idx;

  int pt_idx;

  uint64_t framearray_idx;


  int pts, pds, pdpts;


  int hole_length;


  /** This calculates the total number of bytes available in ram on the  system by taking the last
block base address and adding its
        length to it. The blocks of memory on the sytem given to us by the bios from boot1.s **/
  memory_present = block[(*numblocks)-1].base + block[(*numblocks)-1].length;


  /** set the number of entries that will be in the array */

  frame_array_len = memory_present / PAGE_SIZE; /*Note PAGE_SIZE = 0x1000*/


  /** find the number of pages needed to store the array */

  framearray_pages  =  ((memory_present/PAGE_SIZE)  *  sizeof(struct  frame_array_entry_t))  /
PAGE_SIZE;


  if((memory_present/PAGE_SIZE) % framearray_pages)

    framearray_pages += 1;


  /** add the number of frames that will be needed for paging structs  to address the frame array
*/

  pts = framearray_pages % 512 ? (framearray_pages / 512) + 1 : framearray_pages / 512;
  pts += 1; /* in case it crosses a pt boundary virtually */

  pds = pts % 512 ? (pts / 512) + 1 : pts / 512;
  pds += 1; /* in case it crosses a pd boundary virtually */

  pdpts = pds % 512 ? (pds / 512) + 1 : pds / 512;
  pdpts += 1; /* in case it crosses a pdpt boundary virtually */

  framearray_pages += pts + pds + pdpts;

/** first, find a block of physical memory suitable for storing the frame array. the first block is not
suitable bc we've used stuff       there already. **/
  block += 1;

  for(i = 1; i < *numblocks; i++, block++ ) {

    if(block->length == 0)
      continue;

    /* if the block is useable & big enough*/
    if ( (block->type != 1) ||  (block->length <= (framearray_pages * PAGE_SIZE)) )
      continue;

    frame_position = block->base;

    /** find out where in the tables we will store the array based on the fixed virtual address given
**/
```

```
  pml4t_idx = virt2pml4t(frame_array_vaddr);

  pdpt_idx = virt2pdpt(frame_array_vaddr);

  pd_idx = virt2pd(frame_array_vaddr);

  pt_idx = virt2pt(frame_array_vaddr);


  /** NOTE: For setup tables in this function, we are marking them as executable because this
table might be used for some

      executable stuff later on (we don't know) and we don't trust later allocators to check this
top level permissions. marking the

      page as no-execute is sufficient to ensure that the frame array is not executable **/


  /* initialize the virtual address */

  pml4t = (struct page_map_level_4_table *)read_cr3();

  if(!(pml4t->entries[pml4t_idx]).present){

   setup_table( (void *)frame_position, PML4TE2vaddr(pml4t_idx), 1/* executable */);

   /**then memset the frame for the pdpt ITSELF to zero */

   kmemset(PDPTE2vaddr(pml4t_idx,0), 0, PAGE_SIZE);

   frame_position += PAGE_SIZE;

  }


  pdpt = (struct page_directory_pointer_table *)PDPTE2vaddr(pml4t_idx,0);

  if(!(pdpt->entries[pdpt_idx]).present){

   setup_table( (void *)frame_position, PDPTE2vaddr(pml4t_idx,pdpt_idx), 1 /* executable */);

   kmemset(PDE2vaddr(pml4t_idx,pdpt_idx,0), 0, PAGE_SIZE);

   frame_position += PAGE_SIZE;

  }
```

```
    pd = (struct page_directory *)PDE2vaddr(pml4t_idx,pdpt_idx,0);

    if(!(pd->entries[pd_idx]).present){

      setup_table((void *)frame_position, PDE2vaddr(pml4t_idx,pdpt_idx,pd_idx), 1 /* executable

*/);

      kmemset(PTE2vaddr(pml4t_idx,pdpt_idx,pd_idx,0), 0, PAGE_SIZE);

      frame_position += PAGE_SIZE;

    }


    /** this loop maps in the frames to store enough pages to contain the frame array **/

    for ( i = 0; i < framearray_pages; i++ ) {


      if(pt_idx == 512) {

        pt_idx = 0;

        pd_idx++;


        if(pd_idx == 512) {

          pd_idx = 0;

          pdpt_idx++;


          if(pdpt_idx == 512) {

            pdpt_idx = 0;

            pml4t_idx++;


            if(pml4t_idx == 511) {

              printf("Not enough virtual memory to cover the largest block!");

              panic();

            }
          /**add a new pdpt to the pml4t **/
          setup_table( (void *)frame_position, PML4TE2vaddr(pml4t_idx), PG_RW);
          /**then memset the frame for the pdpt ITSELF to zero */
          kmemset(PDPTE2vaddr(pml4t_idx,0), 0, PAGE_SIZE);
          frame_position += PAGE_SIZE;
        }//pdpt's

        /** add a new pdt to the pdpt */
        setup_table( (void *)frame_position, PDPTE2vaddr(pml4t_idx,pdpt_idx), PG_RW);
        kmemset(PDE2vaddr(pml4t_idx,pdpt_idx,0), 0, PAGE_SIZE);
        frame_position += PAGE_SIZE;
      }//pd's

      /** add a new pt to the pdt */
      setup_table((void *)frame_position, PDE2vaddr(pml4t_idx,pdpt_idx,pd_idx), PG_RW);
      kmemset(PTE2vaddr(pml4t_idx,pdpt_idx,pd_idx,0), 0, PAGE_SIZE);
      frame_position += PAGE_SIZE;
    }//pt's

    /** finally attach the frames to actual pt's */
    setup_table((void *)frame_position, PTE2vaddr(pml4t_idx,pdpt_idx,pd_idx, pt_idx), PG_RW |
PG_NX | PG_GLOBAL );
    frame_position += PAGE_SIZE;
    pt_idx++;
  }//for loop on framearray_pages
```

```
 /* we found our good block and finished initialization: break! */
 break;

} // blocks for loop

/** now actually populate the array **/
framearray = (struct frame_array_entry_t*)frame_array_vaddr;

framearray_idx = 0;
block = (struct memmap *)0x804;
for( i = 0; i < *numblocks; i++ ) {

 for ( j = 0 ; j < block->length/PAGE_SIZE; j++ ) {
  if ( block->type == 0x1 )
   framearray[framearray_idx].free = 0x1;
  else
   framearray[framearray_idx].free = 0x0;

  framearray[framearray_idx++].type = block->type;
 }

 /** increment to next block */
 oldblock = block++;

 /** check to see if there is a hole */
 if ( (oldblock->length + oldblock->base) == block->base )
  continue;
```

```
    if ( i == (*numblocks - 1) ) /** if this is the last block */
      break;

    /** there is a hole... */
    hole_length = (block->base-(oldblock->length + oldblock->base))/PAGE_SIZE;
    for ( j = 0; j < hole_length; j++ ) {

      framearray[framearray_idx].free = 0x0; /* no hole is free (TWSS) */
      framearray[framearray_idx++].type = 0x06;
    }
  } /* end the loop iterating over blocks to populate framearray */

  /** Now we want to actually fill the frame array entries with the vaddr that is mapped to the
frame, so we'll walk the page tables to
        find currently present mappings **/
  pml4t = (struct page_map_level_4_table*)0x1000;
  for ( pml4t_idx = 0; pml4t_idx < 512; pml4t_idx++ ) {

    if ( !pml4t->entries[pml4t_idx].present )
      continue;

    for ( pdpt_idx = 0; pdpt_idx < 512; pdpt_idx++ ) {

      pdpt = (struct page_directory_pointer_table*)PDPTE2vaddr(pml4t_idx,0);
      if ( !pdpt->entries[pdpt_idx].present )
        continue;
  }

      for ( pd_idx = 0; pd_idx < 512; pd_idx++ ) {

        pd = (struct page_directory*)PDE2vaddr(pml4t_idx,pdpt_idx,0);
        if ( !pd->entries[pd_idx].present )
          continue;

        for ( pt_idx = 0; pt_idx < 512; pt_idx++ ) {

          pt = (struct page_table*)PTE2vaddr(pml4t_idx, pdpt_idx, pd_idx, 0);
          if ( pt->entries[pt_idx].present ) {

            page = (union page *)PTE2vaddr(pml4t_idx,pdpt_idx,pd_idx, pt_idx);

            /* store virtual address and mark as taken */
            framearray[TABLE2ADDR(page->addr)/PAGE_SIZE].vaddr                =
idx2vaddr(pml4t_idx,pdpt_idx,pd_idx,pt_idx);
            framearray[TABLE2ADDR(page->addr)/PAGE_SIZE].free = 0x0;
          }

        } /* end pt level loop */
      } /* end pd level loop */
    } /* end pdpt level loop */
  } /* end pml4t level loop */

  return;
}
```

APPENDIX E – ACPI Structures and APIC Handling Code

```
struct ACPIHeader {
    char Signature[4]; /* Table signature RSDT, APIC, FADT, etc */
    uint32_t Length; /* Length of the table */
    uint8_t Revision; /* ACPI version number */
    uint8_t Checksum; /* Checksum of table */
    char OEMID[6]; /* OEM identifier */
    uint64_t OEMTableID; /*Optional OEM Table */
    uint32_t OEMRevision; /* Optional OEM revision number */
    uint32_t CreatorID;  /* BIOS vendor that created the table */
    uint32_t RevisionID; /* BIOS revision */
}__attribute__((packed));

struct ACPIMADT { /*ACPI MADT */
    char Signature[4]; /* Signature MADT */
    uint32_t Length; /* Length of MADT */
    uint8_t Revision; /* ACPI Revision number */
    uint8_t Checksum; /* Checksum of the table */
    char OEMID[6]; /* OEM ID */
    uint64_t OEMTableID; /* Optional OEM table */
    uint32_t OEMRevision; /* Optional OEM revision number */
    uint32_t CreatorID; /* BIOS vendor that created the table */
    uint32_t RevisionID; /* BIOS revision */
    uint32_t LocalAPICAddress; /* MMIO address of the local APIC */
    uint32_t Flags; /* Flags for system capabilities */
}__attribute__((packed));

struct APICLocalAPICEntry{
    uint8_t DeviceType; /* set to 0 for APIC */
    uint8_t Length; /* Length of entry */
    uint8_t ProcessorID; /* Proccesor ID */
    uint8_t APICID; /*APIC ID */
    uint32_t flags; /* set to 0 when APIC is usable */
};

struct APICIOAPICEntry{
    uint8_t DeviceType; /* set to 1 for I/O APIC */
    uint8_t Length; /* Length of entry */
    uint8_t IOAPICId; /* I/O APIC ID */
    uint8_t Reserved; /* Reserved */
    uint32_t IOAPICAddress; /* MMIO address for this I/O APIC */
    uint32_t GlobalSystemInterruptBase; /*What interrupt line the I/O APIC starts at */
};

struct APICInterruptOverrideEntry{
    uint8_t DeviceType; /* set to 2 for interrupt override */
    uint8_t Length; /* Length of entry */
    uint8_t Bus; /* 0 for ISA */
    uint8_t Source; /* The interrupt line number */
    uint32_t Interrupt; /* The interrupt pin on the I/O APIC associated with this entry */
    uint16_t Flags; /* Flags to determine how to configure this entry */
};
```

```
void ACPI_Parse_MADT(struct ACPIMADT *madt)

{

 uint8_t *entry, *end_dt;

 struct APICHeader *header;

 struct APICIOAPICEntry *ioapic;

 struct APICLocalAPICEntry *apic;

 struct APICInterruptOverrideEntry *interruptoverride;


 entry = (uint8_t *)madt + sizeof(struct ACPIMADT);

 end_dt = (uint8_t *)madt+madt->Length;


 while(entry < end_dt){

  header = (struct APICHeader *)entry;


  if((header->DeviceType == 0) && (apic->flags & 0x1) == 1){

   apic = (struct APICLocalAPICEntry*)header;

   /* Found an APIC entry */

  }

  else if(header->DeviceType == 1){

   ioapic = (struct APICIOAPICEntry*)header;

   /* Found an I/O APIC entry */

  }

  else if(header->DeviceType ==  2){

   interruptoverride = (struct APICInterruptOverrideEntry*)header;

   /* Found an Interrupt Override entry */

  }
```

```
    entry+=header->Length;  /* move to next entry */

  }

  return;

}


void ACPI_Parse_Entry(struct ACPIHeader *header)

{

  const char *apic = "APIC";

  const char *dmar = "DMAR";

  const char *hpet = "HPET";

  const char *mcfg = "MCFG";


  if(strncmp(header->Signature,apic,strlen(apic))==0)

      ACPI_Parse_MADT((struct ACPIMADT *)header);

  else if(strncmp(header->Signature,dmar,strlen(dmar))==0)

      ACPI_Parse_DMAR((struct ACPIDMAR *)header);

  else if(strncmp(header->Signature,hpet,strlen(hpet))==0)

      ACPI_Parse_HPET((struct ACPIHeader *)header);


  return;

}


void Scan_ACPI(struct RSDPDescriptor20 rsdpdesc)

{

  struct ACPIHeader *rsdt;

  uint32_t *entry_ptr, *end_rsdt;
```

```
 uint32_t address, address_end;

 /* identity page the address given by the RSDT, remember page alignment */
 attach_page(rsdpdesc.RsdtAddress & 0xFFFFFFF000,rsdpdesc.RsdtAddress & 0xFFFFFFF000);

 rsdt = (struct ACPIHeader *)(uintptr_t)(rsdpdesc.RsdtAddress);

 entry_ptr = (uint32_t*)(rsdt+1);
 end_rsdt = (uint32_t*)(((uint8_t*)rsdt)+rsdt->Length);
 address = *entry_ptr; /*Start of ACPI tables*/

 address_end = *(end_rsdt-1)+0x2000; /* The BIOS has a nasty habit of putting the end off a
page boundary */

 /*   First loop we iterate by page size over the entire ACPI table */
 while(address < address_end){
   /*Remember things need to be page aligned to work correctly*/
   /*The BIOS writers won't do it for you*/
   attach_page(address & 0xFFFFFFF000,address & 0xFFFFFFF000);
   address+=0x1000;
  }

 /* Second loop we parse the now mapped in ACPI table */
 while(entry_ptr < end_rsdt)
  {
   address = *entry_ptr;
   ACPI_Parse_Entry((struct ACPIHeader *)(uintptr_t)address);
```

```
      entry_ptr++;

    }



    return;

}
```

APPENDIX F – Fields and Values for APIC MMIO

```
/* Notable APIC MMIO Fields */
enum {
  APIC_APICID =          0x20,          /* When read provides local APIC ID */
  APIC_APICVER =         0x30,          /* Version Register */
  APIC_TASKPRIOR =       0x80,          /* Task Prioirty Register */
  APIC_EOI =             0x0B0,         /* End of Interrupt Register */
  APIC_LDR =             0x0D0,         /* Logical Destination Register */
  APIC_DFR =             0x0E0,         /* Destination Format Register */
  APIC_SPURIOUS =        0x0F0,         /* Spurious Interrupt Vector */
  APIC_ESR =             0x280,         /* Error Status Register */
  APIC_ICRL =            0x300,         /* Interrupt Command Register Low */
  APIC_ICRH =            0x310,         /* Interrupt Command Register High */
  APIC_LVT_TMR =         0x320,         /* Local Vector Table Timer Register */
  APIC_LVT_PERF =        0x340,         /* Local Vector Table Performance Register */
  APIC_LVT_LINT0 =       0x350,         /* Local Vector Table Logical Interrupt 0 */
  APIC_LVT_LINT1 =       0x360,         /* Local Vector Table Logical Interrupt 1 */
  APIC_LVT_ERR =         0x370,         /* Local vector Table Error Register
  APIC_TMRINITCNT =      0x380,         /* Initial Timer Count */
  APIC_TMRCURRCNT =      0x390,         /* Current Timer Count */
  APIC_TMRDIV =          0x3E0,         /* Divide Configuration for Timer */
};

/* Notable values that can be wrote to APIC MMIO */
enum{
  APIC_DISABLE =         0x10000,       /* Disable Value */
  APIC_SW_ENABLE =       0x100,         /* Enable Software Interrupts */
  APIC_NMI =             0x0400,        /* Set Non Maskable */
  TMR_PERIODIC =         0x20000,       /* Perodic Timer */
  TMR_BASEDIV =          0x100000,
  APIC_DEASSERT =        0x00000000,    /* Deassert Level-Sensitive Interrupt */
  APIC_ASSERT =          0x00004000,    /* Assert Level-Sensitive Interrupt */
  APIC_ALLINC =          0x00080000,    /* Message All Including Self */
  APIC_LEVEL =           0x00008000,    /* Level Triggered Interrupt */
  APIC_INIT =            0x00000500,    /* INIT/RESET IPI */
  APIC_EDGE =            0x00000000,    /* Edge Triggered Interrupt */
  APIC_DELIVS =          0x00001000,    /* Delivery Status  */
  APIC_STARTUP =         0x00000600,    /* Startup IPI */
  APIC_TDIV_1 =          0xB,           /* APIC timer dividers 1,2,3,8,16,32,64,128*/
  APIC_TDIV_2 =          0x0,
  APIC_TDIV_4 =          0x1,
  APIC_TDIV_8 =          0x2,
  APIC_TDIV_16 =         0x3,
  APIC_TDIV_32 =         0x8,
  APIC_TDIV_64 =         0x9,
  APIC_TDIV_128 =        0xA,
};
```

Appendix G – Time Stamp Counter (TSC) Handling Code

```
inline uint64_t readtscp() {

  uint32_t lo, hi;

  asm volatile("rdtscp" : "=a"(lo), "=d"(hi) :: "rcx" );

  asm volatile("cpuid");

  return (uint64_t)(lo) | ((uint64_t)(hi) << 32);

}


inline uint64_t readtsc() {

  uint32_t lo, hi;

  asm volatile("cpuid");

  asm volatile("rdtsc" : "=a"(lo), "=d"(hi) :: "rcx" );

  return (uint64_t)(lo) | ((uint64_t)(hi) << 32);

}


uint64_t get_tsc_freq(){


  uint64_t perf_stat_msr, platform_msr, flex_msr;

  uint64_t ratio, flex_ratio_max, flex_ratio_min, flex_ratio_cur;

  uint64_t tsc_freq;


  perf_stat_msr = read_msr(0x198);

  platform_msr  = read_msr(0xCE);

  flex_msr      = read_msr(0x194);


  /*per the Intel manuals and various online sites. Bit 31 of the performance
```

```
 *stats register indicates xe is running if so read the ratio from the perf
 *stat msr if it's not running the value is in the platform msr
 */
if(((perf_stat_msr >> 31) & 1)){
  ratio = ((perf_stat_msr & 0x1F0000000000) >> 40);
}
else{
  ratio = (( platform_msr  & 0xFF00) >> 8);
}

/*the platform also has the min and max ratios */
flex_ratio_min = ((platform_msr & 0x3F0000000000) >> 40);
flex_ratio_max = ((platform_msr & 0xFF00) >> 8);
/*the flex msr is theory has the current ratio */
flex_ratio_cur = ((flex_msr & 0xFF00) >> 8);

/*bit 16 of the flex msr tells us the ratio's can vary!!
 *however if the lower bits are zero we default to the max
 *the frequency multiplier is 100 unless the bios is overclocked
 */
if(( flex_msr >> 16) & 1){
  if(flex_ratio_cur == 0){
    tsc_freq = flex_ratio_max * 100 * 1000 * 1000;
  }
  else{
    tsc_freq = flex_ratio_cur * 100 * 1000 * 1000;
  }
  }
  /*if they're not flexing then just use the correct ratio from above*/
  else{
    tsc_freq = ratio * 1000 * 1000 * 100;
  }

  return tsc_freq;
}
```

Appendix H – VMEXIT APIC Access Handling Code

```
static void apic_access_handler( vproc_t *vp ){

 uint64_t qualification, ins_len, vp_RIP;


 /* Take care of the fact that we need move past the instruction that

    caused a vmexit in the first place. In this case it is the size of

    an apic access */

 vmread(VM_EXIT_INSTRUCTION_LEN, &ins_len);

 vmread(GUEST_RIP, &vp_RIP);

 vmwrite(GUEST_RIP, vp_RIP + ins_len);


 /*Table 27-6 in the vmexit section chapter 27 of the intel manual describes*/

 /*The qualification field is where this information is populated          */

 /*the access type that caused a vmexit. 0x0000 = read, 0x1000 = write,     */

 /*others are not currently handled       ^ (read bit)   ^ (write bit)     */

 vmread(EXIT_QUALIFICATION, &qualification);


 /* For now we assume the guest has complete control of the core it is      */

 /* configuring. Thus, while we are catching what the guest would like to do*/

 /* we do not restrict what the guest is doing. Some checks should be added */

 /* to prevent potential out of bounds behaviour, but the main push for this*/

 /* is motivated by the drive to give the guest complete control of the core*/

 /* it is using. We are not enabling external interrupt exiting and we are  */

 /* leaving the exception bitmap zeroed out. This allows the guest IDT to   */

 /* handle all of the interrupts it is receiving. This is also a performance*/

 /* improvment as we skip coming into the hypervisor for timer interrupts.  */
/* The only real emulation we should see here is the aknowledgment of EOI, */
/* but we should be able to elimintate that with incoming intel updates.   */
if((qualification & 0x1000) == 0x1000){

 if((qualification & 0xfff) == APIC_ICRL){

   /* If it is an init signal start the count for joining a core*/
   if( vp->reg_storage.rsi == APIC_INIT){
    count_startup_ipis++;
   }/* We received a SIPI count the first*/
   else if( (vp->reg_storage.rsi & 0xf00) == APIC_STARTUP){
    count_startup_ipis++;

    /*Second SIPI received time to join a core to the guest */
    if(count_startup_ipis == 3){
      count_startup_ipis = 0;

      /*IPI's don't have a lot of room for data. So, we use a global  */
      /*vproc pointer to keep track of who we are joining to          */
      SIPI_vp = vp;
      /*send the ipi to the guest the core wants to start             */
      /*This is neat as we can read the previous ICRH write to find */
      /*out who we are trying to message                              */
      send_ipi((lapic_read(APIC_ICRH) >> 24), HYPV_PSEUDO_SIPI);

      /*wait for the core we are starting to acknowledge the IPI      */
      while(lapic_read(APIC_ICRL) & APIC_DELIVS);
```

```
        }

      }

    }

    else{/*write everything else to the APIC*/

      lapic_write((qualification & 0xfff), vp->reg_storage.rsi);

    }

  }  /*else if reads are harmless*/

  else if((qualification & 0x1000) == 0x0000){

    vp->reg_storage.rax = lapic_read((qualification & 0xfff));

  }

  else{/* else disaster strikes */

    printf("[HYPV APIC] unknown APIC access caught\n");

    panic();

  }

  restore_gpregs(vp);

  launch_vproc(vp);


  return;

}
```

Appendix I – Utility Virtual Machine Queue Code

```
static void *util_msgq; /* the utility vm message queue */


/* initialized the utility vm message queue */

void init_util_msg_queue(void){

  util_msgq = qopen();

  return;

}


/* This just adds a utility vm core  message to the utility message queue */

void add_util_msg(Util_msg_t* msg){

  qput(util_msgq, msg);

  return;

}




/* The helper function for now will only be called when another core sends  */

/* an interrupt to a core that lets the other core know it has a message in  */

/* the hypervisor. Thus, if the util_message queue or core number are NULL  */

/* a catastrophic error has occurred. This function just pulls the first    */

/* found message belonging to a specific core from the util message queue.  */

static int core_msg_cmp(void* msg, const void* core_number){


  if (msg == NULL){

    kprintf("NULL util_msg_t given to core_msgcmp\n");

    panic();
  }

  if (core_number == NULL){
    kprintf("NULL core_number given to core_msgcmp\n");
    panic();
  }

  return ( (((Util_msg_t*)msg)->core_dst) == (*(uint32_t*)core_number) );
}

/* a core in the hypervisor will call this function to retrieve a message   */
/* from the utility message queue. It will only return messages that belong */
/* to that specific core, because this_cpu returns core specific APIC ID    */
Util_msg_t* remove_util_msg(void){
  uint32_t core_id = this_cpu();
  Util_msg_t* ret;

  ret = (Util_msg_t*)qremove(util_msgq, &core_msg_cmp, &core_id);
  /* Since the other core is sending an ipi to the core that will call this  shouldn't happen*/
if(ret == NULL){
    kprintf("core does not have a message waiting for it PANIC\n");
    panic();
  }

return ret;
}
```

Bibliography

[1] Paul Barham et al., "Xen and the Art of Virtualization," in *ACM Smposium on Operating Systems Principles*, 2003, pp. 164-177.

[2] Robert P. Goldberg, "A Survey of Virtual Machine Research," in *Computer*, vol. 7, 1974, pp. 34-45.

[3] Robert Denz and Stephen Taylor, "A Survey on Securing the Virtual Cloud," *The Jorunal of Cloud Computing*, vol. 2, no. 1, pp. 1-9, 2013.

[4] VMWare, "Virtualization Overview," Vmware, Inc., Palo Alto, White Paper 2006. [Online]. http://www.vmware.com/pdf/virtualization.pdf

[5] Intel, *Intel® 64 and IA-32 Architectures Software Developer's Manual*. U.S.: Intel, 2015.

[6] Roberto Paleari, Lorenzo Martignoni, Roglia Giampaolo, and Danilo Bruschi, "A Firstful of red-pills: how to automatically generate procedures to detect CPU emulators," in *USENIX conference on Offensive Technologies*, vol. 3rd, 2009.

[7] Peter Ferrie, "Attacks on More Virtual Machine Emulators," in *Symantec Technology Exchange*, 2007, pp. 1-17.

[8] Niall Fitzgibbon and Mike Wood, "Conficker.C A Technical Analysis," Sophos Labs Inc., White Paper 2009.

[9] Danny Quist and Val Smith, "Detecting the Prescence of Virtual Machines Using Local Data," in *Offesnive Computing*, 2006.

[10] Joanna Rutowski, "Red Pill. or how to detect VMM using (almost) one CPU instruction," , 2004.

[11] Amani Ibrahim, James Hamlyn-Harris, and John Grundy, "Emerging Security Challenges of Cloud Virtual Infrastructure," in *APSEC Cloud Workshop*, 2010.

[12] Manuel Corregedor and Sebastian Von Solms, "Implementing Rootkits to Address Operating System Vulnerabilities," in *Information Security South Africa*, 2011.

[13] E. Buchanan, R Roemer, S. Savage, and H. Shacham, "Return- Oriented Programming: Exploitation without Code Injection," in *Black Hat*,

2008.

[14] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *ACM Conference on Computer and Communications Security*, New York, 2007, pp. 552-561.

[15] R. Hund, T. Holz, and F.C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *USENIX Security Symposium*, 2009, pp. 383-398.

[16] Serdar Cabuk, Chris Dalton, Aled Edwards, and Anna Fischer, "A Comparative Study on Secure Network Virtualization," HP Laboratories, Technical Report HPL-2008-57 , 2008.

[17] Marco De Vivo, Gabriela De Vivo, and Germinal Isern, "Internet Security Attacls at the Basic Levels," in *ACM SIGOPS Operating System Review*, vol. 32, 1998, pp. 4-15.

[18] Peter Chen and Brian Noble, "When Virtual is Better Than Real," in *Hot Topic in Operating Systems Workshop*, vol. 8th, 2001.

[19] Sina Bahram et al., "DKSM: Subverting Virtual Machine Introspection for Fun and Profit," in *Symposium on Reliable Distributed Systems*, vol. 29th, 2010.

[20] Greg Hoglund and James Butler, *Rootkits: Subverting the Windows Kernel*, 1st ed.: Addison-Wesley Professional, 2005.

[21] Neal Leavitt, "Is Cloud Computing Really Ready for Prime Time?," in *Computer*, vol. 42, 2011, pp. 15-20.

[22] Goordon E. Moore, "Cramming more Component onto Integrated Circuits," *Electronics*, vol. 38, no. 8, April 1965.

[23] Ofri Wechsler, "Inside Intel Core Microarchitecture," Intel, White Paper 2006.

[24] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig, "Intel ® Virtualization Technology: Hardware Support for Efficient Processo r Virtualization," *Intel Technology Journal*, vol. 10, no. 3, pp. 167-178, 2006.

[25] C. Nichols, "Bear - a Resilient Core for Tactical Systems," in *MILCOM*, Nov. 2013, pp. 1416-1421.

[26] Jorrit Herder, Herbert Bos, Ben Gras, Philip Homburd, and Andrew Tanenbaum, "MINIX 3: a highly reliable, self-repairing operating system," in *ACM SIGOPS Operating Systems Review*, vol. 40, New York, 2006, pp. 80-89.

[27] R. K. Pandey and Vinay Tiwari, "Reliability Issues in Open Source Software," in *International Journal of Computer Applications*, vol. 34, 2011.

[28] Martin Osterloh, Robert Denz, and Stephen Taylor, "Diversified-NFS," in *2nd International Conference on Cloud Security Management (ICCSM)*, Reading, 2014.

[29] Fernando J. Corbató and Victor A. Vyssotsky, "Introduction and Overview of the Multics system," in *Join Computer Conference ACM*, vol. part I, 1965.

[30] Scott Brookes, Robert Denz, Martin Osterloh, and Stephen Taylor, "ExOShim: Preventing Memory Disclosure using Execute-Only Kernel Code," in *Submitted to ICCWS*, 2016.

[31] Scott Brookes, Martin Osterloh, Robert Denz, and Stephen Taylor, "The KPLT: The Kernel as a shared Object," in *In Proceedings of MILCOM*, 2015.

[32] Tanenbaum and Woodhull, *Operating Systems: Design and Implementation*.: Prentice Hall, 2006.

[33] The MPI Forum, "MPI: A Message Passing Interface, version 2.2.," University of Tennesse, Knoxville, Specification 2009.

[34] Alan Heirich and Stephen Taylor, "A Parabolic Load Balancing Method," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, pp. 235-248, 1998.

[35] Chuck Lever and Chuck Boreham, "malloc() Performance in a Multithreaded Linux Environment ," in *USENIX Annual Technical Conference*, San Diego, 2000, pp. 55-56.

[36] AIM Independent Resource Benchmark. (2013, February) sourceforge. [Online]. http://sourceforge.net/projects/aimbench/

[37] Al Daniel, "CLOC - Count Lines of Code," Software 2016.

[38] Robert Denz and Stephen Taylor, "Securing the Cloud Through Utility Virtual Machines," in *IMCIC: Complexity, Informatics, and Cybernetics*, vol. 1, Orlando, 2016, pp. 187-192.

[39] Gerwin Klein et al., "seL4: Formal Verication of an OS Kernel," in *ACM Symposium on Operating Systems Principles*, 2009.

[40] Gordon Lyon, "NMAP Network Scanning: The Official NMAP Project Guide to Network Discovery and Security Scanning," , 2009.

[41] D. Kennedy, J. O'Gorman, D. Kearns, and M. Aharoni, *Metasploit: The Penetration Testers Guide*, 1st ed.: No Starch Press, 2011.

[42] L. Davi, A. Dmitrienko, AR. Sadeghi, and M. Winandy, "Privlilage Escalation Attacks on Android," *Information Security Journal, Springer*, 2011.

[43] Brancik Kenneth, *Insider computer fraud: an in-depth framework for detecting and defending against insider IT attacks*, 1st ed.: CRC Press, 2007.

[44] Scott Campbell, "How to think about security failure," *Communications of the ACM*, vol. 49, no. 1, pp. 37-39, 2006.

[45] David Cross and Edmund McGarrel, *Enhancing security throughout the supply chain*, 1st ed. Washington DC: IBM Center for the Business of Government, 2004.

[46] Allen Householder, Kevin Houle, and Chad Dougherty, "Computer Attack Trends Challenge Internet Security," *(Supplement to Computer Magazine)*, vol. 4, pp. 5-7, 2002.

[47] Chris Eagle, *The IDA Pro Book*, 1st ed.: No Starch Press, 2005.

[48] Eldad Eilam, *Reversing*, 1st ed.: Wiley, 2005.

[49] J.E. Forrester and B.P. Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing," in *USENIX Windows Systems Sumposium*, 2000.

[50] Stephen Checkoway et al., "Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage," in *USENIX/ACCURATE/IAVoSS Electronic Voting Technology Workshop*, 2009.

[51] Dorothy Denning, "An Intrusion-Detection Model," in *IEEE Transactions on Software Engineering* , vol. SE-13 Issue 2 , 1987, pp. 222-232.

[52] Aman Bakshi and B Yogesh, "Securing cloud from DDOS Attacks using Intrusion Detection System in Virtual Machine ," in *International Conference on Communication Software and Networks* , 2010.

[53] Richard Lippmann, Joshua Haines, David Fried, Johnathan Korba, and Das Kumar, "The 1999 DARPA Off-Line Intrusion Detection Evaluation," *The International Journal of Computer and Telecommunications Networking - Special issue on recent advances in intrusion detection systems* , vol. 34, no. 4, pp. 579-595, 2000.

[54] Tal Garfunkel and Medel Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection ," in *Network and Distributed Systems Security Symposium*, 2003.

[55] Gene Kim and Eugene Spafford, "The Design and Implementation of Tripwire: A File System Integrity Checker ," in *ACM Conference on Computer and communications security*, 1994, pp. 18-29.

[56] Steven Hofmeyr, Stephanie Forrest, and Anil Somayaji, "Intrusion Detection using Sequences of System Calls," *Journal of Computer Security*, vol. 6, pp. 151-180, 1998.

[57] Abinav Srivastava and Jonathon Giffin, "Tamper-Resistant, Application- Aware Blocking of Malicious Network Connections ," in *International symposium on Recent Advances in Intrusion Detection* , 2008, pp. 39-58.

[58] Jonas Pfoh, Christian Schneider, and Claudia Exkert, "A Formal Model for Virtual Machine Introspection," in *ACM workshop on Virtual machine security* , 2009, pp. 1-10.

[59] Peter Loscocco, Perry Wilson, Aaron Pendergrass, and Durward McDonell, "Linux Kernel Integrity Measurement Using Contextual Inspection ," in *ACM workshop on Scalable trusted computing* , 2007, pp. 21-29.

[60] D Zhu and Erika Chin, "Detection of VM-Aware Malware," University of Berkeley, White Paper 2007.

[61] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song, "Dynamic Spyware Analysis ," in *USENIX Annual Technical*

Conference , 2007.

[62] Ilsun You and Kangbin Yim, "Malware Obfuscation Techniques: A Brief Survey ," in *International Conference on Broadband, Wireless Computing, Communication and Application* , 2010.

[63] Benjamin Livshits, "Dynamic Taint Tracking in Managed Runtimes," Microsoft Research, Technical Report MSR-TR-2012-114 , 2012.

[64] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu, "Stealthy Malware Detection Through VMM-Based “Out-of-the-Box” Semantic View Reconstruction," in *ACM conference on Computer and communications security*, 2007, pp. 128-138.

[65] T. Klein. (2003) Scooby Doo - VMware Fingerprint Suite. [Online]. http://www.trapkit.de/research/vmm/scoopydoo/index.html

[66] J. Giffin, "The Next Malware Battleground Recovery after Unknown Infection ," *IEEE Journal on Security and Privacy* , pp. 74-76, 2010.

[67] CERT, "CERT/CC Security Improvement Modules: Analyze all available information to characterize an intrusion ," CERT Coordination Center , Technical Report 2001.

[68] Dave Dittrich, "Report on the Linux Honeypot Compromise," The Honeyney Project, Technical Report 2000.

[69] George Dunlap, Samuel King, Sukru Cinar, Murtaza Basrai, and Peter Chen, "ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay," in *Operating systems design and implementation* , 2002, pp. 211-224.

[70] Kerstin Buchacker, Volkmar Sieh, and Friedrich Alexander, "Framework for testing the fault- tolerance of systems including OS and network aspects ," in *IEEE High- Assurance System Engineering Symposium* , Universität Erlangen-nürnberg , 2001.

[71] Joanna Rutkowska and Alexander Tereshkin, "Bluepilling the Xen Hypervisor," in *Black Hat*, 2008.

[72] Reiner Sailer et al., "sHype: Secure Hypervisor Approach to Trusted Virtualized Systems ," IBM Research, Research Report RC23511, 2005.

[73] Peter Loscocco and Stephen Smalley, "Integrating Flexible Support for

Security Policies into the Linux Operating System," , 2001, pp. 29-42.

[74] Trent Jaeger, Reiner Sailer, and Xiaolin Zhang, "Analyzing integrity protection in the SELinux example policy ," in *USENIX Security Symposium* , 2003.

[75] Sebastian Vogl, "Secure Hypervisors," in *International Conference on Enterprise Information System* , 2010.

[76] Ge Cheng, Hai Jin, Deqing Zou, Alex Ohoussou, and Feng Zhao, "A Prioritized Chinese Wall Model for Managing the Covert Information Flows in Virtual Machine Systems ," in *International Conference for Young Computer Scientists* , 2008, pp. 1481-1487.

[77] Guanhai Wang, Minglu Li, and Chuliang Weng, "Chinese Wall Isolation Mechanism and Its Implementation on VMM ," in *Systems and Virtualization Management. Standards and the Cloud* , vol. 71, 2010, pp. 13-18.

[78] Reiner Sailer et al., "Building a MAC- Based Security Architecture for the Xen Open-Source Hypervisor ," in *Computer Security Applications Conference* , 2005.

[79] Jakub Szefer, Eric Keller, Ruby Lee, and Jennifer Rexford, "Eliminating the Hypervisor Attack Surface for a More Secure Cloud ," in *ACM conference on Computer and communications security, pp 401-412* , pp. 401-412.

[80] Derek Murray, Gregorz Milos, and Steven Hand, "Improving Xen Security through Disaggregation ," in *ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* , 2008, pp. 151-160.

[81] Yaozu Dong, Zhao Yu, and Greg Rose, "SR-IOV networking in Xen: architecture, design and implementation ," in *First conference on I/O virtualization* , 2008.

[82] Christopher Clark et al., "Live Migration of Virtual Machines ," in *Symposium on Networked Systems Design & Implementation* , vol. 2, pp. 273-286.

[83] Mohamed Gouda and Alex Liu, "A Model of Stateful Firewalls and its Properties ," in *IEEE International Conference on Dependable Systems and Networks* , 2005.

[84] Chrstopher Kruegel, Fredrik Valeur, Giovanni Vigna, and Richard Kemmerer, "Stateful Intrusion Detection for High-Speed Networks ," in *IEEE Symposium on Security and Privacy* , 2002.

[85] Chen Xianqin, Wan Han, Wang Sumei, and Long Xiang, "Seamless Virtual Machine Live Migration on Network Security Enhanced Hypervisor ," in *Broadband Network & Multimedia Technology* , 2009, pp. 847-853.

[86] J. Laprie, "Resilience for the scalability of dependability ," in *ISNCA* , 2005, pp. 5-6.

[87] Flavio Lombardi, Roberto Di Pietro, and Claudio Soriente, "ReW: Cloud Resilience forWindows Guests through Monitored Virtualization ," in *IEEE Symposium on Reliable Distributed Systems* , 2010, pp. 338-342.

[88] Flavio Lombardi and Roberto Di Pietro, "Kvmsec: a security extension for linux kernel virtual machines ," in *ACM symposium on Applied Computing* , 2009, pp. 2029-2034.

[89] Flavio Lombardi and Roberto Di Pietro, "Secure virtualization for cloud computing," *Journal of Network and Computer Applications* , pp. 1113-1122, 2011.

[90] Rusty Russel, "lguest: Implementing the little Linux hypervisor," in *Linux Symposium* , vol. 2, 2007, pp. 173-178.

[91] Hans Reiser and Rudiger Kapitza, "Hypervisor-Based Efficient Proactive Recovery ," in *IEEE International Symposium on Reliable Distributed Systems* , 2007, pp. 83-92.

[92] Hans Reisner and Rudiger Kapitza, "VM-FIT: Supporting Intrusion Tolerance with Virtualisation Technology ," in *Workshop on Recent Advances on Intrusion-Tolerant Systems* , 2007, pp. 18-22.

[93] Andrew Tenenbaum, *Modern Operating Systems*.

[94] Rick Lehtinen, Deborah Russell, and G.T. Gangemi Sr., *Computer Security Basics*, 2nd ed.: O'Reilly Media, 2006.

[95] S. Berger et al., "Security for the cloud infrastructure: Trusted virtual data center implementation," *IBM Journal of Research and Development*, vol. 53, pp. 560-571, 2009.

[96] Stefan Berger, Ramon Caceres, Dimitrios Pendarakis, Reiner Sailer, and Enriquillo Valdez, "TVDc: Managing Security in the Trusted Virtual Datacenter ," in *ACM SIGOPS Operating Systems Review* , vol. 42, 2008, pp. 40-47.

[97] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou, "Achieving Secure, Scalable, and Fine-grained Data Access Control in Cloud Computing ," in *Information communications* , 2010, pp. 534-542.

[98] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., Toshiba Corporation, "Advanced Configuration and Power Interface Specification," Specification 2013.

[99] Intel, "MultiProcessor Specification," Intel Corporation, Specification May 1997.

[100] AMD, "AMD64 Architecture Programmer's Manual," Advanced Micro Devices Corporation, Architecture Specification Publication No. 24592, 2013.

[101] Intel, "Intel Platform Innovation Framework for EFI Firmware File System," Intel Corporation, Architecture Specification 2013.

[102] Unified EFI, Inc, "Unified Extensible Firmware Interface," UEFI Forum, Architecture Specifcaton 2015.

[103] Christopher Domas, "The Memory Sinkhole," in *Black Hat*, Las Vegas, 2015.

[104] Love Robert, *Linux Kernel Development*, 2nd ed.: Sams Publishing, 2005.

[105] Rob Pike et al., "Plan 9 from Bell Labs," *Computing Systems*, vol. 8, no. 3, pp. 221-254, Summer 1995.

[106] James Larkby-Lahet, Brian A Madden, Dave Wilkinson, and Daniel Mosse, "XOmB: an Exokernel for Modern 64-bit, Multicore Hardware," University of Pittsburgh, White Paper 2010.

[107] Gary Kildall, "The History of CP/M, The Evolution of an Industry: One Person's Viewpoint," *Dr. Jobb's Journal of Computer Calisthenics & Orthodontia*, vol. 5, no. (1)(41), pp. 6-7, 1980.

[108] Ray Duncan, *The MS-DOS Encyclopedia*. USA: Microsoft Press.

[109] ECMA, "Volume and File Structure of Disk Cartridges for Information Interchange," Technical Specification Standard ECMA-107, 1995.

[110] Irv Englander, *The Architecture of Computer Hardware and System Software: An Information Technology Approach*, 5th ed. USA: Wiley, 2014.

[111] Intel, "8259A Programmable Interrupt Controller (8259A/8259A-2)," Intel Corporation, Datasheet Order Number� 231468-003 , 1988.

[112] David Patterson and John Hennessy, *Computer Organization and Design. Hardware/Software Interface*, 4th ed. Burlington, MA, USA: Morgan Kaufmann Publishers, 2009.

[113] Intel, "8086 16-Bit HMOS Microproocessor 8086/8086-2/8086-1," Intel, Data Sheet Order Number: 231455-005, 1990.

[114] Intel, "High Performance Microprocessor with Memory Managment and Protection," Technical Specification Order Number: 210253-013, 1988.

[115] Masahiko Sakamoto. (2010, May) Why BIOS loads MBR into 0x7C00 in X86. [Online]. http://www.glamenv-septzen.net/en/view/6

[116] Intel, "Dual-Core Intel® Xeon® Processor 2.80 GHz," Intel Corporation, Specification Update Document Number: 309159-008, 2006.

[117] Grigorios Magklis, Fernando Latorre, and Antonio Gonzalez, *Processor Microarchitecture: An Implementation Perspective*.: Morgan & Claypool Publishers, 2011.

[118] Brian Stuart, *Principles of Operating Systems: Design and Applications*, 1st ed.: Cengage Learning EMEA, 2008.

[119] Joseph McGivern, *Interrupt-Driven PC System Design*.: Annabooks, 1998.

[120] Intel, "82489DX Advanced Programmable Interrupt Controller," Intel Corporation, Datasheet Order Number:290446-002, 1993.

[121] Intel, "Using the RDTSC Instruction for Performance Monitoring," Intel

Corporation, Manual 1998.

[122] (2011, June) Kernel Newbies. [Online]. http://kernelnewbies.org/BigKernelLock

[123] Remzi Arpaci-Dusseau and Andrea Apraci-Dusseau, *Operating Systems: Three Easy Pieces*, 1st ed.: Arpaci-Dusseau Books, 2014.

[124] Beth Pariseau. (2011, April) SearchServerVirtualization. [Online]. http://searchservervirtualization.techtarget.com/news/2240034817/KVM-reignites-Type-1-vs-Type-2-hypervisor-debate

[125] American Standards Association, "American Standard Code for Information Interchange," Technical Report ASA X3.4-1963, 1983.

[126] Morgon Kanter, "Enhancing Non-Determinism in Operating Systems," Thayer School of Engineering, Dartmouth College, Hanover, PhD Thesis 2013.

[127] Martin Reddy, *API Design for C++*, 1st ed.: Morgoan Kaufmann, 2011.

[128] Mordechai Ben-Ari, *Principles of Concurrent and Distributed Programming*, 2nd ed.: Pearson, 2006.

[129] Sanjoy Baruah, Louis Rosier, and Rodney Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-time Systems*, vol. 2, no. 4, pp. 301-324, Nov 1990.

[130] Avi Silberschatz and Peter Galvin, *Operating System Concepts*.: Addison-Wesley, 1998, vol. 4.

[131] Atmel, "AT04055: Using the lwIP Network Stack," Atmel Corporation, Technical Report 42233A – SAM – 03/2014, 2014.

[132] TMurgent Technologies, "Processor Affinity," White Paper 2003.

[133] John Hennesy and David Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed.: Morgan Kaufmann, 2011.

[134] Jerrel Watts and Stephen Taylor, "A Practical Approach to Dynamic Load Balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, pp. 235-248, 1998.

[135] Jerrel Watts and Stephen Taylor, "Communications Locality Preservation in Dynamic Load Balancing," *High Performance Computing:Grand Challenges in Computer Simulation*, pp. 186-190, 1998.

[136] Jerrel Watts and Stephen Taylor, "Automatic Granularity control for Load Balancing of Concurrent Particle Simulations," *Grand Challenges in Computer Simulation*, pp. 115-120, April 1998.

[137] Jerrel Watts and Stephen Taylor, "Dynamic Management of Heterogeneous Resources," *High Performance Computing:Grand Challenges in Computer Simulation*, pp. 151-156, April 1998.

[138] J. Watts and S. Taylor, "A Vector-based Strategy for Dynamic Resource Allocation," *Journal of Concurrency: Practice and Experiences*, 1998.

[139] Jun Nakajima, "Xen as High-Performance NFV Platform," in *Xen Project Developer Summit*, Chicago, 2014.

[140] Mano Marks. (2016, January) Docker. [Online]. https://blog.docker.com/2016/01/unikernel/

[141] K. McGill, "Operating System Support for Resilience," Thayer School of Engineering at Darrtmouth College, PhD Thesis 2011.

[142] Alan Heirich and Stephen Taylor, "Load Balancing by Diffusion," in *International Conference on Parallel Programming*, 1995, pp. 192-202.