

**TOWSON UNIVERSITY  
OFFICE OF GRADUATE STUDIES**

**MULTI-CORE IMPLEMENTATION FOR BARE MACHINE COMPUTING**

**by**

**Hojin Chang**

**A Dissertation**

**Presented to the faculty of**

**Towson University**

**in partial fulfillment**

**of the requirements for the degree**

**Doctor of Science**

**Department of Computer & Information Sciences**

**Towson University**

**Towson, Maryland 21252**

**August, 2017**

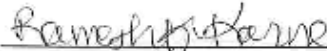
© 2017 by Hojin Chang

All Rights Reserved

TOWSON UNIVERSITY  
OFFICE OF GRADUATE STUDIES


DISSERTATION APPROVAL PAGE

This is to certify that the dissertation prepared by Hoiin Chang, entitled "MULTI-CORE IMPLEMENTATION FOR HARD MACHINE COMPUTING," has been approved by the dissertation committee as satisfactorily completing the dissertation requirements for the degree Doctor of Science in Information Technology.

  
Chair, Dissertation Committee, Dr. Ramesh K. Karne  
7-27-17  
Date

  
Committee member, Dr. Alexander L. Wijesinha  
7/27/17  
Date

  
Committee member, Dr. Sungchul Hong  
7/27/17  
Date

  
Committee member, Dr. Ziyang Tang  
7/27/2017  
Date

  
Doctoral Program Director, Dr. Chan Lu  
7/27/2017  
Date

  
Chair, Department of Computer and Information Sciences,  
Dr. Siddharth Kaza  
7/27/17  
Date

  
Dean of Graduate Studies  
8-2-17  
Date

## Acknowledgements

I would like to express my appreciation to all those who have supported my efforts to complete this dissertation. I am greatly appreciative of my research committee Dr. Ramesh K. Karne (Chair), Dr. Alexander Wijesinha (Co-chair), Dr. Sungchul Hong, and Dr. Ziyang Tang for supporting this research. I am especially thankful to Dr. Karne and Dr. Wijesinha for all the long hours in lab especially in many weekends for their support and advice. Also, I give all glory and honor to my GOD and offer my heartfelt thanks to my beloved parents who are Jaeyoung Chang and JungSoon Yang, whose words have encouraged me to complete my doctoral study.

Many thanks also go to Dr. Chao Lu, Doctoral Program Director of the Department of Computer and Information Sciences at Towson University, for facilitating this work. I am also grateful to the late Frank Anger (National Science Foundation) for his support of the Application Oriented Object Architecture (AOA), which evolved into Bare Machine Computing research and consequently made this dissertation possible.

## Abstract

### MULTI-CORE IMPLEMENTATION FOR BARE MACHINE COMPUTING

Hojin Chang

This dissertation extends on-going Bare Machine Computing (BMC) research at Towson University. BMC applications run on a bare machine without any commercial operating system, kernel, or any other centralized support. This dissertation will serve as a cornerstone for future multi-core systems that run on bare machines.

Multi-core processors are deployed on most desktops, laptops and other devices. Since the inception of BMC, applications were implemented on a 32-bit single core Intel architecture (x86) due to its availability and skill set in the BMC laboratory. The multi-core architecture coupled with 64-bit CPUs poses numerous challenges in implementing bare machine computing applications. This thesis lays a foundation to understand the design issues in implementing and porting existing applications to 64-bit multi-core architecture. First, the existing boot and load programs for BMC are ported to run in the new platform. This posed numerous problems due to architectural differences in 32-bit and 64-bit architectures. Second, some existing applications are ported to run on the new platform. A Web server application written for a 32-bit architecture is made to run on the new platform. This step is involved in a variety of architectural and design issues and incompatibilities among the old and new architectures. Finally, the Web server is made to run on multi-core by scheduling its threads to multiple cores. This posed a daunting challenge due to its complexity. some preliminary work in this area has been completed and the rest is left for future explorations.

This dissertation developed a variety of tools and techniques that are useful for future research in BMC applications. As the current technology is moving towards multi-core CPUs, it is necessary to study this problem for BMC applications. It has been discovered that the 32-bit and 64-bit architectural differences pose major difficulties in porting existing applications to smoothly run 64-bit multi-core architecture, whether they are conventional or bare.

## Table of Contents

List of tables .....	ix
List of figures .....	x
1 MOTIVATION .....	1
2 INTRODUCTION .....	2
3 RELATED WORK .....	4
3.1 Bare Machine Computing Background .....	4
3.2 BMC Applications .....	6
3.3 Other Related Work .....	7
4 INSIGHT INTO x86_64 BARE PC METHODOLOGY .....	9
4.1 Overview .....	9
4.2 Methodology .....	9
4.3 BOOT/LOAD/RUN PROGRAM PROCESS .....	12
4.3.1 Boot Program .....	13
4.3.2 Load Program .....	14
4.3.3 Mass Storage and Memory Layouts .....	14
4.3.4 USB Image Maker .....	18
4.4 BOOT/LOAD/RUN IMPLEMENTATION AND API .....	18
4.5 FUNCTIONAL OPERATION AND TESTING .....	22
4.6 NOVEL FEATURES .....	24
4.7 SUMMARY .....	25
5 Migrating a Web Server to a Multi-core Architecture .....	26
5.1 BASIC ISSUES IN MIGRATION .....	26
5.2 PAGING AND SINGLE CORE .....	27
5.3 MULTICORE MIGRATION .....	30
5.3.1 BSP Flow .....	31
5.3.2 AP Flow .....	39
5.3.3 Task Scheduling Flow .....	45
5.4 FUNCTIONAL OPERATION AND DATA .....	48
5.4.1 First Migration Attempt .....	48
5.4.2 Single Core with Paging .....	48
5.4.3 Multi-Core .....	49
5.5 DISCUSSION .....	51
5.6 SUMMARY .....	52
6 Multi-core BMC Parallelization .....	53
6.1 BMC Tasking .....	53
6.2 Scheduling .....	55
6.3 Existing Inter-Process Communication .....	55
6.4 Multi-core Inter-Process Communication .....	55
7 Conclusion .....	59
Appendix .....	60

A. Directory Structure .....	<b>60</b>
B. Compilation Environment .....	<b>61</b>
C. Debugging Process .....	<b>63</b>
D. User's Guide and Screen Shots .....	<b>64</b>
References .....	<b>69</b>
Curriculum Vitae.....	<b>74</b>



## LIST OF TABLES

Table 1: OS-Based System VS. Bare Machine System.....	<b>5</b>
---	----------

## LIST OF FIGURES

Figure 1: Methodology.....	10
Figure 2: 64-bit Specifics .....	11
Figure 3: USB Layout .....	15
Figure 4: Memory Layout .....	17
Figure 5: EntryPoint32.....	19
Figure 6: Main.c (first Main()).....	19
Figure 7: Entrypoint64.s (64-bit Entrance assembly code).....	20
Figure 8: Main.c (second main()) for 64-bit) .....	21
Figure 9: Selected Interfaces Details.....	22
Figure 10: Menu Interface.....	23
Figure 11: Application Output .....	23
Figure 12: Image Maker Trace.....	24
Figure 13: 4K Paging .....	27
Figure 14: Current Webserver Architecture.....	29
Figure 15: BSP Flow (1E).....	30
Figure 16: GDT, IDT .....	33
Figure 17: MP Configuration Structure .....	34
Figure 18: MP Configuration Data .....	36
Figure 19: Wait Command Flow .....	37
Figure 20: User Menu .....	37
Figure 21: Interrupt Mapping Table.....	38
Figure 22: AP Flow .....	40
Figure 23: Initialize LVT .....	42

Figure 24: AP IDLE Task .....	<b>43</b>
Figure 25: Scheduling (2E) .....	<b>44</b>
Figure 26: Multi-core Bare PC Display .....	<b>46</b>
Figure 27: Multi-core Wireshark Output for a Client Request .....	<b>50</b>
Figure 28: BMC Task Structure .....	<b>54</b>
Figure 29: Multi-core Inter-Process Communication .....	<b>56</b>
Figure 30: Bare PC Structure .....	<b>60</b>
Figure 31: Assembly compiling batch file .....	<b>61</b>
Figure 32: C/C++ Compiling Batch File.....	<b>62</b>
Figure 33: Link Batch File .....	<b>62</b>
Figure 34: Make File .....	<b>63</b>
Figure 35: Memory Dump to debug.....	<b>64</b>
Figure 36: Bare PC Menu .....	<b>65</b>
Figure 37: Load (Option (2)).....	<b>65</b>
Figure 38: Run Application (Option (3)) .....	<b>66</b>
Figure 39: Changing into Multi-Core Mode (Option (6)).....	<b>67</b>
Figure 40: Activate Multi-Core (Option (6)) .....	<b>67</b>

## 1 MOTIVATION

The motivation stems primarily from a need for a 64-bit boot/load/run methodology required for bare PC applications that is part of our mission in the bare machine computing research. Secondly, the current 32-bit bare PC applications developed for x86 architecture can be ported to run on x86-64 architecture in the future. The insight into boot/load/run methodology and its implementation consequently provides a foundation for the development of 64-bit bare PC applications. The methodology presented here also has a broader impact to other pervasive devices and ultimately to the future of bare machine computing paradigm.

## 2 INTRODUCTION

Bare Machine Computing (BMC) [1] [2] [43] [49] [50] is based on running computer applications on a bare machine without any resident operating system (OS). The BMC programming paradigm differs from a conventional programming paradigm in that the programmer manages all the necessary system resources. This paradigm enables full control of the system via the application software, which directly communicates with and controls the underlying hardware. BMC applications are written in mostly in C/C++ with some use of assembly language (MASM, NASM or TASM) as needed. As described in [59], one or more end-user applications can be written as a single entity referred to as an application object (AO) that runs on a bare machine. Existing BMC applications include Web servers [3] [44], mail servers [4], SIP Servers [5] and VoIP systems [6]. BMC applications can run as multi-threaded programs with thousands of threads and yield high performance with high security as there is no centralized OS or kernel, no external dependencies and no resident mass storage.

Many low-overhead operating systems and kernels have been developed for a variety of environments. For example, Tiny OS is an OS for low-power wireless devices [7]. On the other hand, Kitten is a lightweight high-performance OS that can be used together with Palacios, a virtual machine monitor for applications running in a virtualized environment [8]. The elimination of OS abstractions was originally proposed in [9]. In contrast, the BMC paradigm eliminates all intermediary software providing the combined benefits of improved performance and security.

As newer machines are based on a 64-bit processor and multi-core, 32-bit bare PC applications need to be migrated to the new environment. Many techniques have been proposed for migrating applications to multi-core in an OS environment. In [10], tools for migrating applications to multi-core based on Windows and Linux are surveyed. The use of components and partitions for multi-core migration of legacy real-time systems is described in [11]. In [12], the difficulty of migrating real-time software used in the automotive industry to multi-core is discussed and an approach for developing an automated tool based on static program analysis is suggested.

Migrating bare PC code to run on the 64-bit architecture [47] is not the same as migrating OS-based applications. The primary difference is the absence of any OS or other intermediary software between bare PC applications and the hardware. This means that a bare PC application, regardless of whether it is 32-bit or 64-bit, has to manage its own system resources. We describe the technical details underlying the migration process for a 32-bit bare PC Web server. In particular, we consider design and implementation differences in 32-bit versus 64-bit bare PC applications with a view towards building a tool for automating the migration process in the future.

### 3 RELATED WORK

#### 3.1 Bare Machine Computing Background

The Bare Machine Computing (BMC) paradigm was invented by Dr. Ramesh Karne at Towson University, which was also referred to earlier as a dispersed operating system computing (DOSC) system [13]. The key concepts of the BMC paradigm are as follows:

- 1) Computing applications run in a bare machine without Operating System, centralized kernel, or any system software pre-loaded into the machine.
- 2) The applications can be loaded and carried in a portable device such as flash memory, and run in a bare machine anywhere.

The BMC approach is a novel approach for developing computing applications. Unlike traditional approaches, it is application-centric, and completely differs from conventional computing approaches that are environment and platform-centric. There is little need to upgrade or patch the computing environment often; the focus is on the applications themselves. Once a computing box is made bare, the expense to protect it will be minimized since it only has memory, CPU, a basic user interface (input/output), and a network interface. All persistent data is either stored in a removable device such as USB flash memory or on the network.

In the BMC paradigm, an Application Object (AO) [50] is a self-contained, self-controllable and self-executable unit [2], when an AO is developed, it can be run in any bare hardware such as desktop, laptop, and hand-held, or other electronic device.

OS-Based System	Bare Machine System
OS or embedded, Kernel	No OS, No embedding, No Kernel
Centralized Control	Application Controlled
Open Systems	Closed Systems
DLLs, Open Ports	No DLLs, No Open Parts
Some Mass Storage on-board	No mass Storage on-board, External Storage
Multiple Modules, Software, other vendor entities	Single application suite or module
Dynamic Binding	Static Binding
Vulnerable to OS and other commercial Software	No OS or Commercial Software vulnerabilities
General purpose, Multi-user Access	Application Centric and Single User Controlled (User Controlled)
Application depend upon other entities during exception	Self-Controlled, Self-managed application
Many resources to be exploited by intruders In the box	No valuable resources In the Bare box
Virus, worm, ..., etc. can get in	None can get in
Leaves execution trace or valuable resource	None after execution
Open communication to external users	Closed communication to un-intended users
Box has to be secured	Box can be used by any one
Priv. and user mode	User mode only
Other users can access the system while running	Only intended users can access while running
Intruders can damage resources	Intruder may get the system go down but cannot damage any resources
Needs posting to new environments	No posting, runs on any x86(or target) architecture
Frequent updates, Dumping and Waste	No frequent updates, no dumping and no waste
Applications & System Programs	Only application programs

Table 1: OS-Based System VS. Bare Machine System

The BMC approach makes computing application simpler and more secure. An AO is developed in a single programming language and run a bare machine, so the AO developer only needs to know one computer language and AO domain knowledge. The AO has no particular ownership and is self-contained and self-executed so that it can run in any bare machine. As the AO controls both application and execution aspects, in addition to



avoiding all the system and kernel related vulnerabilities by making the device bare, it will be more secure. The BMC paradigm changes the way applications are developed today. A summary of comparing conventional applications to BMC applications in Table 1 illustrates the similarities and differences.

### 3.2 BMC Applications

Several complex bare applications have been developed in the bare machine computing laboratory at Towson University, and were the most renowned outcome of doctoral research. They clearly show the feasibility of the BMC paradigm. Long He [3] developed the first bare PC Web server and demonstrated the feasibility of building complex software that runs on a bare PC with thousands of threads and outperforms other compatible commercial Web servers. Gholam Khaksari [6] developed the first VoIP soft-phone that runs on a bare PC and provides secure communication [51] on an end-to-end basis. Andre Alexander [5] built a SIP server and a bare SIP user agent to demonstrate the feasibility of running high performance SIP servers with secure communication using the SRTP protocol. George H. Ford built the first Email server that runs on a bare PC and provides compatible performance to related commercial email servers [4, 14, 15]. Ali Emdadi [16] implemented the complex TLS protocol for a bare Web server. Roman Yasinovskyy [17] [40] implemented the IPv6 protocol for a bare PC VoIP softphone client. Tsetse [55] [57] implemented the Bare PC NAT Box and 6to4 Gateway [56]. IPSec on Bar PC was implemented by Kazemi [58]. Bharat Rawal [18] developed a unique split protocol concept and applied it to Web servers that run on a bare PC. He also developed mini-cluster configurations for Web servers based on the split protocol concept that offer high

performance [19] and run on a bare PC. Patrick Appiah-Kubi developed a secure Webmail server using TLS [20]. Karne [33] developed a USB mass storage device driver for BMC applications. Loukili [53] [54] studied TCP permeance in BMC and Augusto-Padilla [52] studied the possibility of transforming a Linux wireless driver to run on a Bare PC. Okafor [21] demonstrated the feasibility of transforming the SQLite database to run on a bare PC. Alexander [22] [23] showed the applicability of BMC paradigm to handheld devices. Liang [24] and Thompson developed FAT32 file systems [25] [26] that run on bare PCs. These previous doctoral research projects made possible the discovery of many novel characteristics that are unique to BMC and that would be applicable to future computing applications that run on bare devices.

### 3.3 Other Related Work

Since the invention of computers over 50 years ago, software complexity has been growing rapidly. The massive growth in computing hardware and software has created unmanageable electronic waste [34]. This is partly because new software cannot work with legacy hardware. Software applications, operating systems, tools and gadgets become obsolete in years-sometimes in months. The operating system size has also increased dramatically. For example, for the Microsoft Windows XP professional version with SP3, the size of OS is over one GB.

Much research has focused on developing small kernels and lean OSs or dedicated applications. Previous and recent work include exterminating OS abstractions [9], Exokernel Applications [28, 29], IO-Lite [35], Palacio [8], Libra [27], bare-metal Linux [38], OS Kit [37], Fluke kernel [30], Sandboxing [31], Factored OS [39], Tiny OS [36],

and fast and flexible networking [32]. However, there are significant differences between the BMC approach and those mentioned above. The BMC approach has abandoned the OS approach completely; instead, the AO, a self-contained unit, manages the CPU and memory. The bare machine computing paradigm is at the extreme end of the spectrum compared to OS based systems and other intermediate lean kernel systems.

*As per our knowledge, no work on bare machine multi-core and 64-bit applications has been done before.*

## 4 INSIGHT INTO x86\_64 BARE PC METHODOLOGY

### 4.1 *Overview*

Development of bare PC boot/load/run program methodology is not a trivial task. Most of the tools and codes available on the Web require some sort of operating system, kernel, or embedded system. Consequently, the boot process in particular is simpler as the kernel handles complex load and run processes as needed during the execution. The traditional OS and kernel programs that are designed to work with x86 and x86-64 architectures are mature, complex and huge. They hide the architectural intricacies and complexity inside the kernel; thus they are, often overlooked by the application programmer. For a system programmer to work with a complex kernel is a frustrating and a daunting job. In a bare PC application, the application itself manages boot, load and execution of its own program. This poses different challenges and requires the handling of internals of CPU and direct communication with hardware at run time.

### 4.2 *Methodology*

There are ample amounts of resources and knowledge scattered throughout the Web to construct boot and load programs [42] for conventional platforms that are based on some sort of OS, kernel or embedded systems. As per our knowledge, there are no relevant, useful links, resources, or tools available for bare PC development other than our own research in bare machine computing in the past decade. This section outlines a general overview of this process and the subsequent sections deal with more specific information.

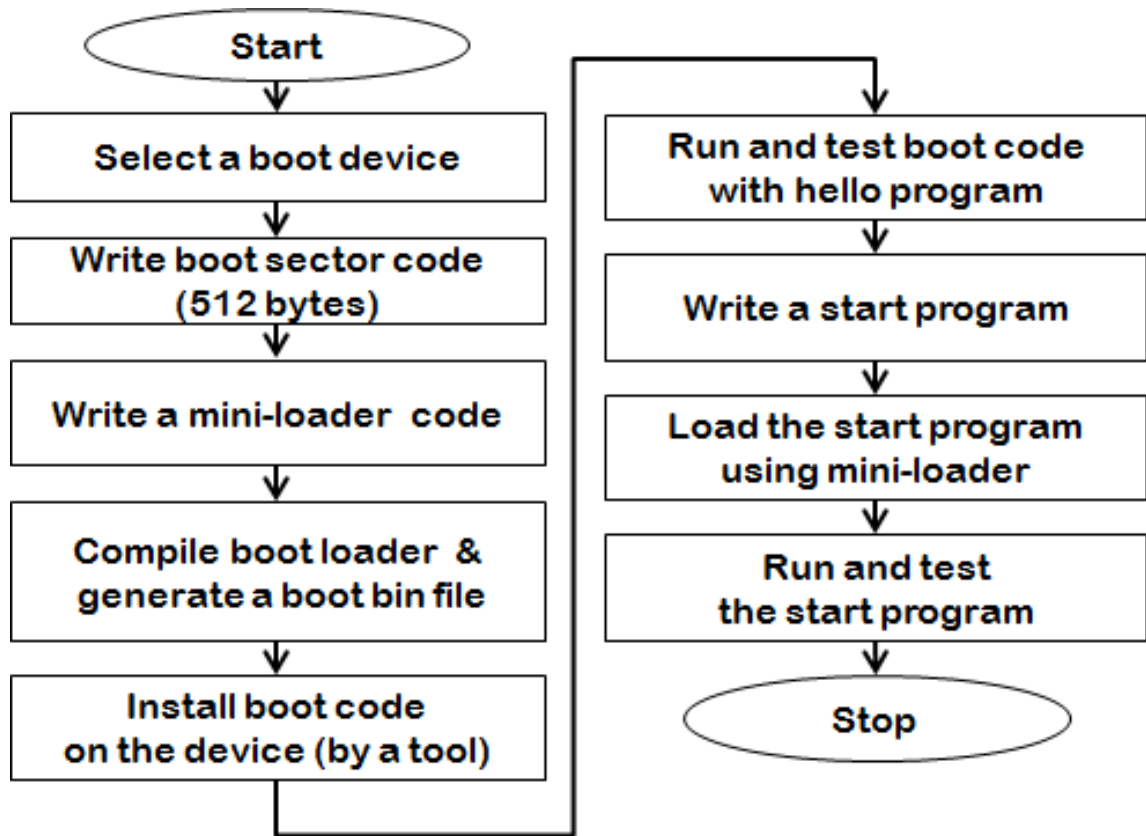


Figure 1: Methodology

Fig.1 shows a generic methodology to illustrate a boot/load/run program process. For a bare PC application, one needs to select a bootable device, which is a detachable mass storage device such as a USB flash drive. A boot sector code and mini-loader that work with a bare PC are needed. The chosen assembler can write this code and compile it to generate a binary file. This bin file is transferred to a bootable USB using a boot install tool. An initial test of boot code can be done by simply printing a “Hello” message after the boot. Once this simple boot test is done, one can write a start program in assembly that can be loaded by the mini-loader (which is part of the boot code). The start program is the beginning of a first program after the boot process.

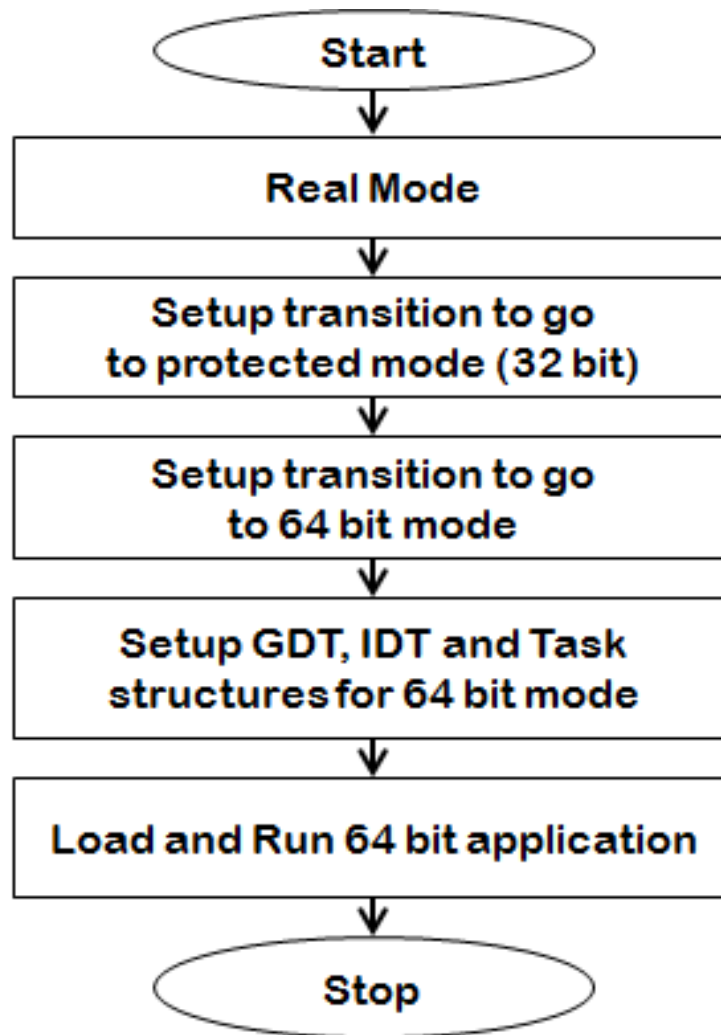


Figure 2: 64-bit Specifics

Fig.2 shows some more details needed to run a 64-bit application. When a PC is booted, it starts in a real mode, which is limited to a 1MB address and has access to all BIOS (Basic Input Output System) interrupts. These interrupts can be used to accomplish simple I/O tasks that are needed during the boot and load process. The start program is used to move from real to protected (32-bit) mode where you have full access to a 4GB address space. In order to transition to 32-bit protected mode, some control registers, GDT and IDT need to be appropriately initialized. Once the 32-bit protected mode is established, you can transition to 64-bit mode which requires a different setting. The 64-bit mode requires setup

for paging, TLB (Translation Look-Aside Buffer), GDT (Global Descriptor Table) and IDT (Interrupt Descriptor Table) entries. Paging is mandatory in 64-bit mode, but optional in 32-bit mode. One can also go directly to 64-bit mode from real mode using the same start program with appropriate setup. A 64-bit application can be loaded, run and tested for its operation once it is in 64-bit mode.

Notice that, as the 64-bit mode supports its previous real and protected modes, the boot/load/run program process for this is more complicated than its predecessors. The transition from one mode to another can be made using interrupt gates or long jump instruction. If we need to use any BIOS calls, we need to go back to real mode from 64-bit mode. Thus, one can design a cycle going from real-protected-64, 64-real, or 64-protected-real modes. In some cases, in a 64-bit mode, if you need to run a 32-bit mode application, you may also choose to go back to 32-bit mode or run it in a compatibility mode. The compatibility mode also requires manipulation of control registers and other structures. The implementation details of this process will be described in later sections. The 64-bit mode and its usage depend upon the need of a user to run strictly 64-bit applications or a mixed set of applications.

### ***4.3 BOOT/LOAD/RUN PROGRAM PROCESS***

This section describes boot, load and run program process in detail. It also shows the intricacies involved in creating such programs for a bare PC application.

#### 4.3.1 *Boot Program*

A boot program [42] usually consists of a 512-byte (could be more) block of a binary file. It is written in an assembly language and it has a single “text” segment. No data or stack segment is needed to create a working boot program. A boot code needs to be compiled as a binary executable. A variety of compilers such as NASM, MASM, TASM or GAS can be used to create this binary. Note that these assemblers are not compatible in their syntax and structures. A bootable mass storage device such as a hard disk, USB or a CD is used to contain the boot binary in sector 0. For an IBM compatible PC, the boot code on the bootable device is at sector 0, which is loaded into memory at 0x7c00 by its BIOS interrupt upon a power on state. The CPU starts executing this boot code starting at this address. The boot code must contain a 2-byte signature at the end of its 512-byte block (0x55AA). A simple “hello” boot does not have any loader. The “hello” text can be printed using video memory, indicating the success of a boot process.

In IBM compatible PCs, initially video memory can be used to display text on the screen. The video memory start location is at 0xB8000, which can be loaded in the ES segment register to address this memory. To display each character, it requires two bytes to be stored in the video memory, one byte for data and the other byte for the properties. Sophisticated graphics and color patterns require a graphics driver to handle the display and visualization. Simple text and graphics can be done using video memory.



#### 4.3.2 *Load Program*

A detachable mass storage device (USB flash drive) is used to store boot/load application programs and data. There is a mini-loader in the boot program that is simple and limited in its usage due to its size constraints in the boot code. When a boot code is running, the system is in real mode and it has access to BIOS interrupts. The BIOS interrupt 0x13 is used to read all needed sectors from the USB into main memory. The load program requires the starting sector number and the total number of sectors to read from the device. Initially, it reads all sectors needed into memory during initialization and in real mode. If more data is needed to read or write to the USB in protected or 64-bit mode, a device driver and a sophisticated loader are needed for USB . It is also possible to fall back to real mode to read more data using interrupt 0x13 again.

#### 4.3.3 *Mass Storage and Memory Layouts*

Insight into the boot/load/run program process is illustrated using USB map and memory map figures as shown in Fig. 3 and Fig. 4 respectively. The USB map shows the layout of files that are installed on it in chronological order using a “USB Image Maker” tool, which creates a bootable USB image for bare PC applications. The boot loader (BootLoader.bin) is always installed at sector 0. Sector 1 consists of an entry point code written in assembly (EntryPoint32.bin), which provides a transition from real to protected mode (16 to 32-bit). Entry point code is always loaded in sector 1.

There are three other types of binary files in the USB map as shown in Fig. 3: Mode32.bin, Mode64.bin and Hello.elf. The Mode32.bin file consists of Main.o and

ModeSwitch.o objects including the 32-bit main. The Main.o (first main) object is a 32-bit C program, which demonstrates the transition to 32-bit mode. In addition, it checks available memory and obtains information using get CPUID. The ModeSwitch.o written in assembly provides a transition from 32-bit mode to 64-bit mode. The Mode64.bin consists of EntryPoint64.o and Main.o (second main) objects including the 64-bit main. The EntryPoint64.o is an assembly program that provides an entry point to 64-bit mode. The Main.o (second main) object is a 64-bit C program that demonstrates the transition to 64-bit mode. The Hello.elf is the user application program for 64-bit mode that is installed at the end of all other files. The total number of sectors illustrated in this example include: 106 sectors (1 boot sector, 1 entrypt sector, 5 Mode32.bin sectors, 65 Mode64.bin sectors and 34 Hello.elf sectors).

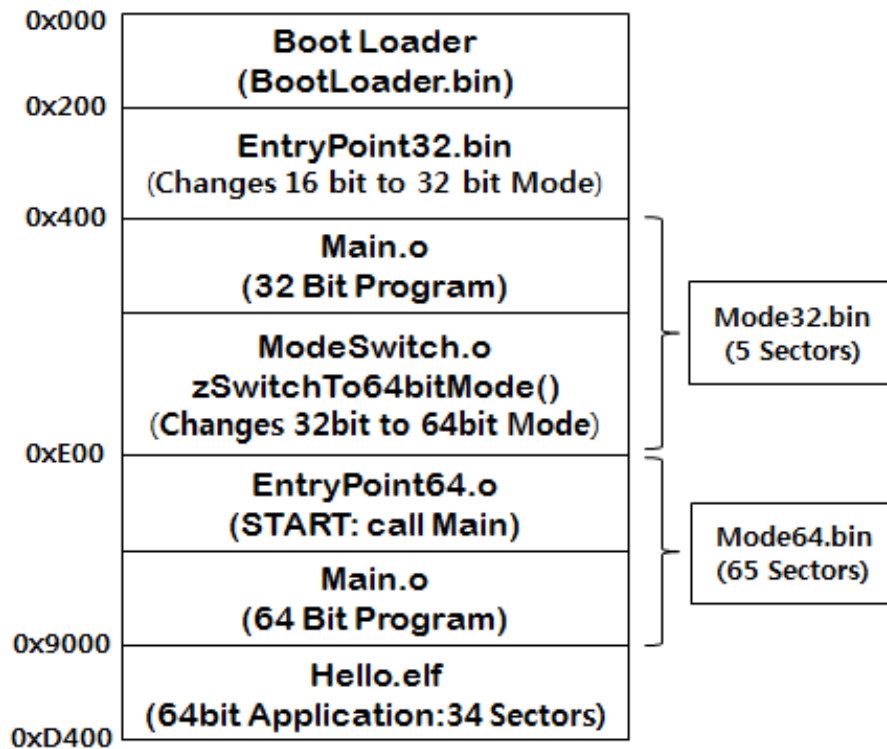


Figure 3: USB Layout

In this given example, 105 sectors are loaded into memory during boot code execution, excluding the boot sector (i.e. 0xD200 bytes). The USB map shows the individual file sizes and their locations. The mini-loader loads all these sectors initially using interrupt 0x13 in real mode.

In this example, the 105 sectors are loaded into memory as shown in Fig. 4. We load these sectors starting above the 1MB address as they can be accessed in protected and 64-bit modes. The starting address for EntryPoint32.bin is at 0x00010000. Thus, the boot loader jumps to the START label in this entry point code using a long JMP instruction (**jmp 0x1000:0x0000**). The ES register is loaded with 0x1000, which translate to 0x00010000, that is the 1MB starting address. At this point, all segment registers have zero values. The entry point code setup transition from real to protected mode and then jumps to a C program main() using another long JMP instruction (**jmp dword 0x18:0x10200**). The GDT entry setup for this Main.o code is 0x18 ( $0x18/8 = 3$ ), which is a 3<sup>rd</sup> entry in the table. The Main.o object invokes ModeSwitch.o object, which sets up the 64-bit mode using zSwitchTo64bitMode() function. After the 64-bit mode, setup it jumps to EntryPoint64.o START label using a long JMP instruction (**jmp 0x08:0x10c00**). The GDT entry setup for Mod64.bin code is 0x08 ( $0x08/8 = 1$ ), which is a 1<sup>st</sup> entry in the table. The 0<sup>th</sup> entry in the GDT table is defined as a null GDT. This entry point code will call a 64-bit program Main.o (second main) which provides a user menu to load and run a 64-bit application. This is part of Mode64.bin as shown in the memory map.

The user menu helps to load and run the “**Hello.elf**” application. Initially, the “Hello.elf” application was loaded in memory at location 0x00018e00 during the boot time. When this application is actually loaded by user in a 64-bit mode, it may be relocated to other part of the memory such as 0x500000, as illustrated in the memory layout. In a bare PC setup, usually memory map is designed and controlled by the AO programmer and usually it is a fixed location to load a given AO. However, a given AO can have a set of applications constituted as a single user AO.

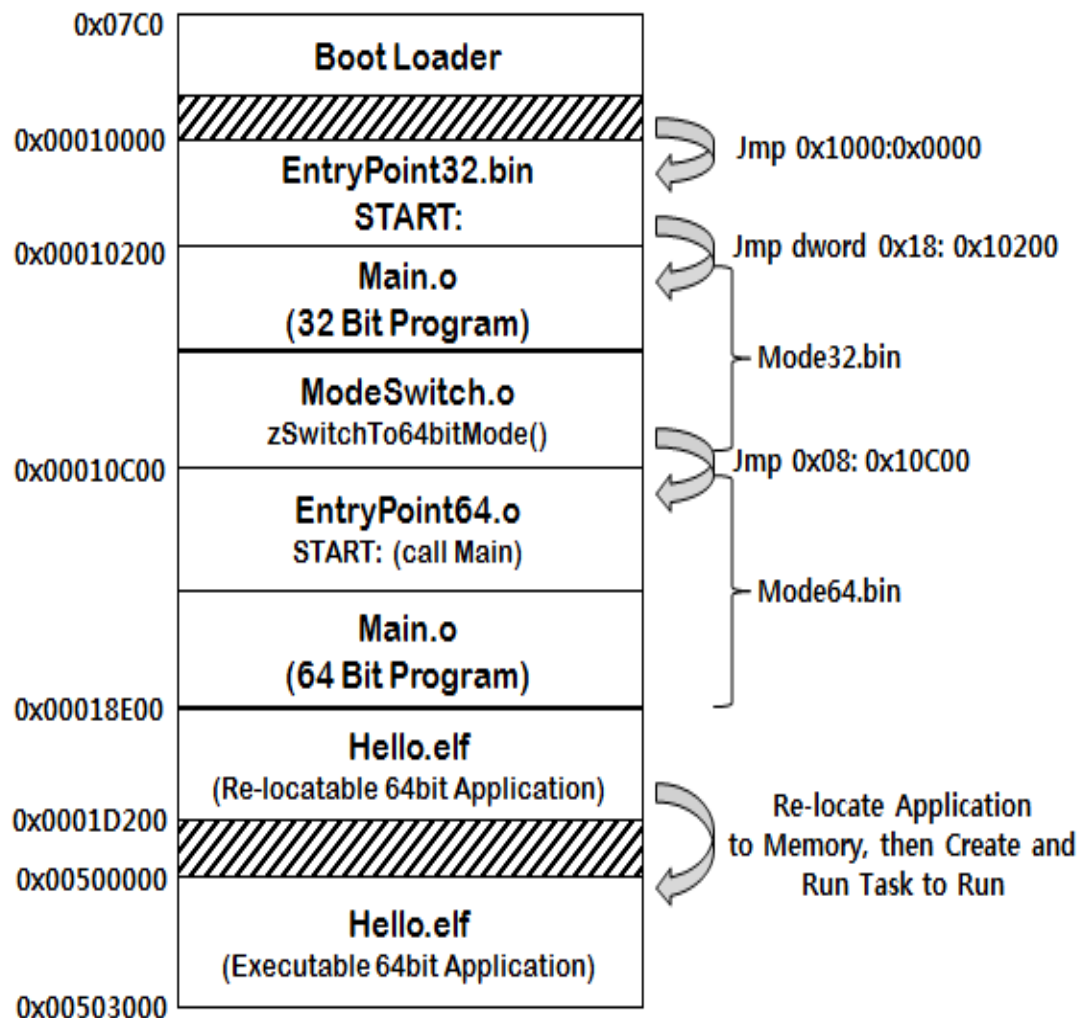


Figure 4: Memory Layout

#### 4.3.4 *USB Image Maker*

We built a “USB Image Maker” [41] tool using a C program that runs on Linux. This tool uses a “makefile” to compile its code using some input options. The tool uses all the binary and boot images as shown in Fig. 3. To install them on the USB, we use Win32DiskImager which is a free development software. The USBImageMaker makes the USB bootable image, but the Win32DiskImager creates a bootable USB. The USBImageMaker tool is modified so that the boot record bytes 5, 7 and 9 to pass information to the bare PC programs. Byte 5 shows the total number of sectors including the boot sector (106 sectors), byte 7 indicates the size of Mod32.bin (5 sectors) and byte 9 shows the size of Module64.bin (65 sectors). The size of Hell.elf can be derived with the above parameters. This tool can also be used to pass many other parameters to the bare PC application. Notice that the bare PC boot image is not same as other OS boot images and the unused bytes in the boot image can be used as parameter space. The bare PC boot image is also not compatible with other OS boot images. Most of the code used for this tool is off-the-shelf [41] and some modifications are made to suit our bare PC development environment. More details of this tool are not covered in this paper due to space constraints.

#### 4.4 *BOOT/LOAD/RUN IMPLEMENTATION AND API*

This section shows some internal details of implementation and some generic API that can be used to build your own 64-bit bare PC applications [48]. Fig. 5 shows the entry point code (EntryPoint32.s), which is invoked from the boot loader as shown in Fig. 4. The snippets of assembly code are shown in Fig. 5 to illustrate the steps implemented in this program. It uses BIOS interrupt to set A20 bit, which is needed to transition to higher

memory above 1MB. It initializes GDT register and loads control register 0 to go to protected mode. Finally, it uses a long jump to go to Main.o (first main) located at 0x10200 and GDT entry 0x18 (3rd entry).

```

<EntryPoint32.s>
; Activate A20 by BIOS service
mov ax, 0x2401      ; set A20 activation service
int 0x15            ; call BIOS interrupt service

lgdt [ GDTR ]      ; set GDTR structure

mov eax, 0x4000003B
; PG=0, CD=1, NW=0, AM=0, WP=0, NE=1, ET=1, TS=1, EM=0, MP=1, PE=1
mov cr0, eax        ; write them to CR0, then switch to the protected mode

[BITS 32]           ; 32 bit mode(Protected Mode)
PROTECTEDMODE:

```

Figure 5: EntryPoint32

```

<Main.c (first main())>
//print a string on display
void zStringLinePrint(int cX, int cY, const char* cptrString);
//check memory size if it is more than 64Mbytes or not
BOOL zCheckEnoughMemory(void);
void zInitializePageTables(void); // Create page table for 64bit(IA-32e) mode
// get CPU-ID, vendor string
zGetCPUID( 0x00, &dwEAX, &dwEBX, &dwECX, &dwEDX);
// check up supporting 64 bit or not
zGetCPUID( 0x80000001, &dwEAX, &dwEBX, &dwECX, &dwEDX);
// switch mode 32 bit to 64 bit, then jump to 64bit(IA-32e) mode

```

Figure 6: Main.c (first Main())

Written in C, the Main.c (first main) in Fig. 6 lists the main steps taken in this program. It creates a page table for 64-bit mode and uses CPUID instruction to obtain the vendor-id

and 64-bit mode supportability feature. Finally, it calls zSwitchTo64bitMode() assembly call that is located in EntryPoint64.s.

```
<Entrypoint64.s>  
mov ax, 0x10 ; set 64 bit  
mov ds, ax ; DS  
mov es, ax ; ES  
mov fs, ax ; FS  
mov gs, ax ; GS  
; stack for 64bit mode  
; with 1MB size  
mov ss, ax ; SS  
mov rsp, 0x6FFF8  
; set RSP  
mov rbp, 0x6FFF8  
; set RBP  
call Main  
; 64 bit main  
; program
```

Figure 7: Entrypoint64.s (64-bit Entrance assembly code)

The EntryPoint64.s program written in assembly is shown in Fig. 7. It initializes data segments with GDT selector 2 ( $0x10 = 16/2 = 2$ ). It initializes the stack and calls Main program (this is the second main) in Fig. 8.

The EntryPoint64.s program written in assembly is shown in Fig. 7. It initializes data segments with GDT selector 2 ( $0x10 = 16/2 = 2$ ). It initializes the stack and calls Main program (this is the second main).

```

<Main.c(second main())>
void zInitializeConsole( int iX, int iY );
void zGetCursor( int *piX, int *piY );
void zSetCursor( int iX, int iY );
void zInitializeGDTableAndTSS( void );
void zLoadGDTR( QWORD qwGDTRAddress );
void zLoadTR( WORD wTSSSegmentOffset );
void zInitializeIDTTables( void );
void zLoadIDTR( QWORD qwIDTRAddress);
zCheckTotalRAMSize();
zInitializePIC();
zMaskPICInterrupt( 0 );
zEnableInterrupt();
TCB* zCreateTask(...);
void zInitializeKeyboard(void);
BYTE zGetCh( void );
void zClearScreen( void );
void zMenu(void);
void zWaitCommand(void);
void zChoiceToStopTask(void);
int zMemDump(void);

```

Figure 8: Main.c (second main() for 64-bit)

Most of the API shown in this figure is self-explanatory. Some important interfaces are shown in Fig. 9 with some key details. The `zInitializeGDTableAndTSS()` function updates 64-bit GDT with new parameters required in 64-bit mode. It initializes TSS for 64-bit, which is different from 32-bit mode. The `zLoadGDTR()` function loads the GDT register with new GDT table. The `zLoadTR()` function loads the task register. The `zInitializeIDTTable()` initializes the IDT table. A sample IDT entry for keyboard is shown to illustrate this API. Notice that the keyboard IDT entry is pointing to 0x08 which is a 64-bit mode code segment descriptor. The `zLoadIDTR()` function initializes IDT table and loads IDT register. Only one IST (interrupt stack table) is used in our demonstration, which is used to create and run one 64-bit mode application. In 64-bit mode, task switching is



done in software, whereas in 32-bit mode, it is done in hardware. The details of task management are not shown in this paper due to space limitations. All the functions described in this section illustrate a common API that can be used to construct boot/load/run programs for any x86-64 based bare PC applications.

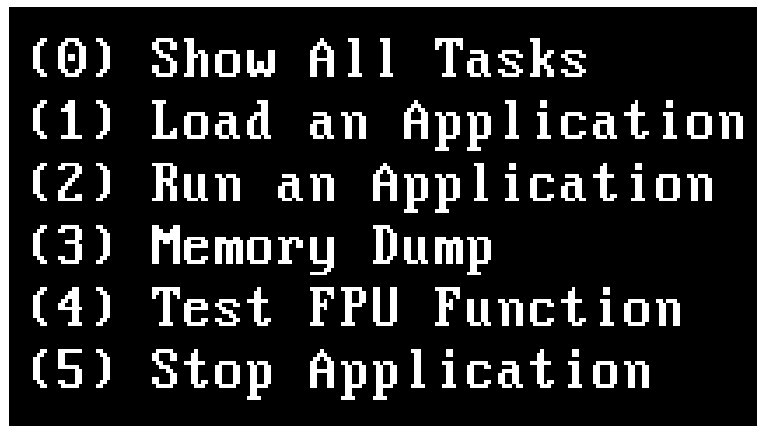
<more details of selected interfaces>	
<b>void zInitializeGDTTableAndTSS( void );</b>	
-	Update 64 bit GDT Code and Data (0x08, 0x10)
-	The new entries reflect 64 bit needs
-	Initialize TSS for 64 bit (it's different from 32 bit)
<b>void zLoadGDTR( QWORD qwGDTRAddress );</b>	
-	lgdt [ rdi ]
-	rdi contains GDT table address
<b>void zLoadTR( WORD wTSSSegmentOffset );</b>	
-	ltr di
-	di contains TSS segment(16 bytes) descriptor offset
<b>void zInitializeIDTTables( void );</b>	
-	create IDT table
-	add IDT entries
	(e.g. Keyboard: zSetIDTGate( &(amp; ptrIDT[ 33 ] ), kISRKeyboard, 0x08, IDT_FLAGS_IST1, IDT_FLAGS_PGR, IDT_TYPE_INTERRUPT ); )
<b>void zLoadIDTR( QWORD qwIDTRAddress);</b>	
-	0x08 is for entry of GDT code
-	lidt [ rdi ]
-	rdi contains IDT table address
-	only one IST table used(IST1)

Figure 9: Selected Interfaces Details

#### 4.5 FUNCTIONAL OPERATION AND TESTING

The application program, boot and loader is installed in a USB (mass storage device). This USB can be used to boot/load/run on any x86-64 compatible PC without any hard disk or OS. We used a 2GB Verbatim USB and a 64-bit Laptop (ASUS N43S Intel Core

i5, 3rd generation CPU) to test this boot/load/run program process. When the power is turned on, it boots, loads its 64-bit application and runs the application. When it is booted, it starts in real mode and moves into protected and subsequently to 64-bit mode to run the application. When the system transitions to 64-bit mode, it provides a user interface (menu) to load a 64-bit application and runs. It also provides a menu option to dump memory contents for debugging purposes. We have tested a small 64-bit application program using this boot/load/run program process. This system can be expanded to run any set of applications and other features. The snap shots of the running process for the 64-bit “Hello” program are shown in Fig. 10 thru Fig. 12. The Fig. 10 shows the menu interface, Fig. 11 shows the hello program display, and Fig. 12 shows the trace produced by the USB Image Maker tool.



```
(0) Show All Tasks
(1) Load an Application
(2) Run an Application
(3) Memory Dump
(4) Test FPU Function
(5) Stop Application
```

Figure 10: Menu Interface



```
Hello, Welcome 64-bits world!!
```

Figure 11: Application Output

```

./UsbImageMaker.exe 00.BootLoader/BootLoader.bin 01.Mode32
/Mode32.bin 02.Mode64/Mode64.bin 04.Application/hello.elf

[INFO] Copy boot loader to image file
[INFO] File size is aligned 512 byte
[INFO] 00.BootLoader/BootLoader.bin size = [512]
      and sector count = [1]
[INFO] Copy protected mode program to image file
[INFO] File size [2690] and fill [382] byte
[INFO] 01.Mode32/Mode32.bin size = [2690]
      and sector Count = [6]
[INFO] Copy IA-32e mode program to image file
[INFO] File size [32948] and fill [332] byte
[INFO] 02.Mode64/Mode64.bin size = [32948]
      and sector count = [65]
[INFO] Copy 64bit Application to image file
[INFO] File size [17256] and fill [152] byte
[INFO] 04.Application/hello.elf size = [17256]
      and sector count = [34]
[INFO] Start to write all information
[INFO] Total sector count except boot loader [105]
[INFO] Total sector count of
      protected mode program [6]
[INFO] Total sector count of 64-bit application [34]
[INFO] Image file create complete

```

Figure 12: Image Maker Trace

#### 4.6 NOVEL FEATURES

This work presents a complete methodology of creating a boot program, loading application into memory and running a program without any need for an operating system, kernel, or embedded system. There is no middleware required other than the application itself. The CPU structures GDT, IDT, TSS, Video Memory, Keyboard and Display are directly controlled by the application object programmer. This programmer has a complete knowledge of application at static and run time. There is no commercial or vendor software involved in this software engineering paradigm. An end user application suite can be carried on a mass storage device and run anywhere on a bare PC. The direct hardware

control and interfaces (API) shown in this paper provide a complete overview to construct a standalone 64-bit application. Eventually, this API implementation can be moved into the processor chip thus making it intelligent and providing direct access to the programmer. We can eventually make this API standard across many pervasive devices to create portable applications that run on many pervasive devices. When hardware devices are made bare, they can be placed anywhere without concern for computer protection other than physical damage or vandalism.

#### **4.7 SUMMARY**

This presents a novel methodology to write bare PC applications that are independent of any operating system, kernel, or embedded system. It described internal details of implementation for boot, load and run process for a 64-bit application. These detailed code snippets and prototypes can be used to implement your own system to create bare PC applications. It also showed its functional operation and testing of a boot/load/run program process. Some significant contributions of this paper, which have a broader impact in developing future bare PC or machine applications, have been identified.

## 5 MIGRATING A WEB SERVER TO A MULTI-CORE ARCHITECTURE

### 5.1 BASIC ISSUES IN MIGRATION

The original bare PC Web server application [3] was written in Visual Studio 10 using C/C++ (32-bit) code and some MASM (Microsoft assembler) code. It also used TASM (Turbo assembler) and NASM (Netwide assembler) code for startup and boot code. The server ran on a single CPU x86 desktop (Dell Optiplex 520) with no paging and no hard disk. A USB flash drive was used to store the entire application (boot, startup and application code including data). In order to migrate the 32-bit code to run on an x86\_64-bit multicore desktop (Dell Optiplex 9010) [47], the first attempt was to run the original server code without changes. Several issues were identified, some of which are discussed below.

#### (1) Static Variables

Static variables had incorrect values in them (stored in BSS(Basic Stack Segment) although the BSS segment was not used in the 32-bit code). This did not appear to cause any problems on the 32-bit machine. Bare PC applications use batch files to compile and link i.e. Visual Studio 10 “/bin” files are used for compilation without any use of header files (or libraries). The problem was fixed by adding a “/MERGE:.bss=.data” option in the link for running on a 64-bit machine. This option merges all data into a single data section.

#### (2) Dummy Blocks

The 64-bit machine was found to hang if dummy block braces “{....}” were used (without any conditions preceding). This was not an issue in 32-bit; the braces were simply removed to fix this problem.

#### (3) C++ Classes

The same C++ class was compiled in two directories and linked, which should not be the case. Simply renaming the class in one directory solves this problem.

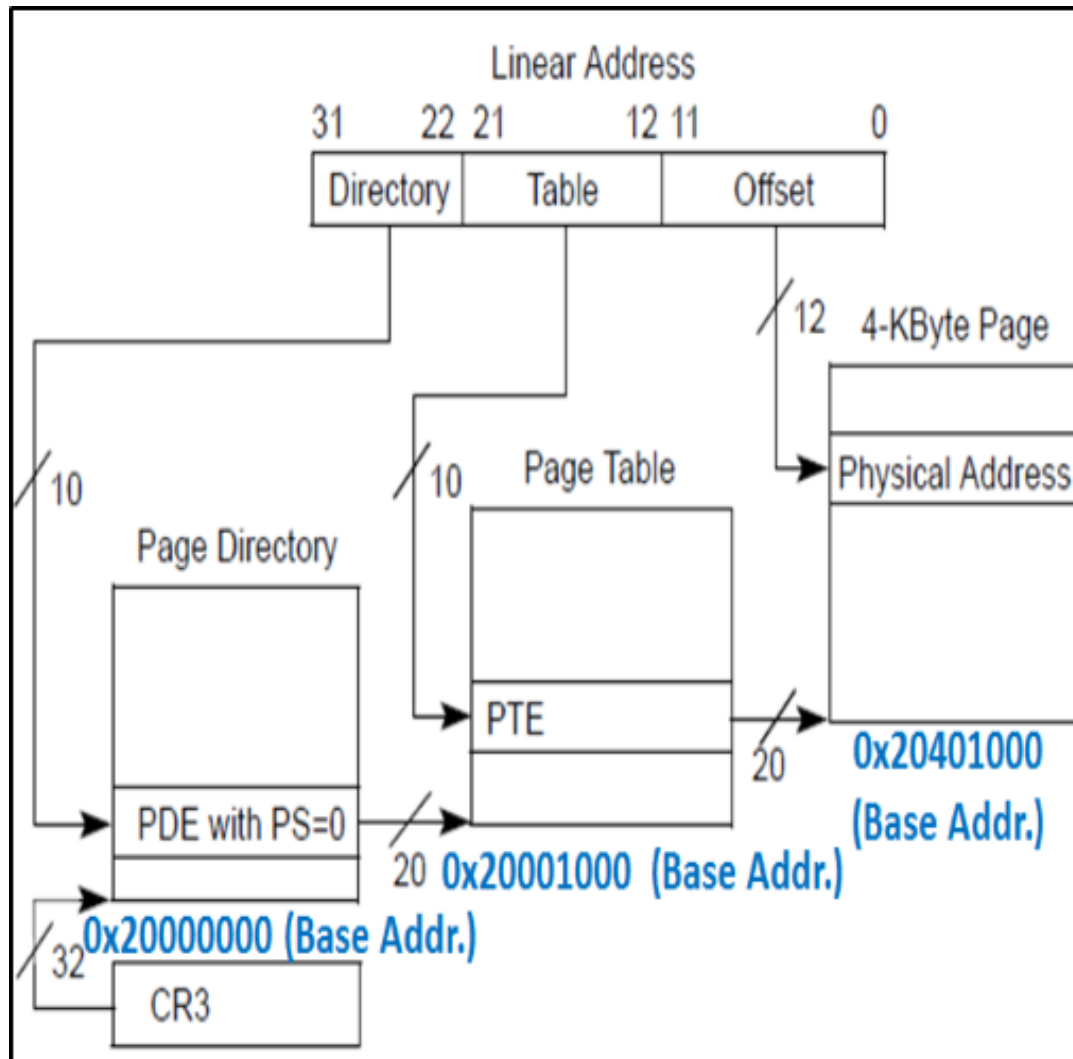


Figure 13: 4K Paging

## 5.2 PAGING AND SINGLE CORE

The 32-bit bare PC code has no paging i.e. the CR0 (control register 0) PG(page) bit and CR4 PAE(page enable) bits are zeros [46] (p.1961). As the Web server is a 32-bit application, paging mode was enabled by setting the CR0 PG bit to 1. This needed to be done before the Web server tasks are created. We therefore created a paging data structure

and initialized the structure with the necessary memory before paging was turned on. The entire Web server executable is 325KB and 1GB of address space is used for the application including code, data and stack. We used a 4K paging scheme and created a data structure as shown in Figure 13. It also shows the actual values used in the PDE and PTE entries. The CR3 is initialized with 0x20000000 (512MB), which is the base address of page directory (PD). This initialization is done in the TSS (task state segment) for the Web server application. There are 1024 entries in PD (there is only one PD in the system). Each PD entry (PDE) points to a separate page table (PT), and there are 1024 PTs. Each page table entry (PTE) points to a page where the offset is used to address the exact location in memory. The paging is managed by the CPU, and software only initializes the data structures. All entries were provided in order to address up to 4GB of memory.

The 32-bit server uses physical memory up to 1GB. This memory is contiguously mapped in our paging structure, thus avoiding page replacement overhead i.e. each logical page is mapped to one physical page. The 32-bit server also uses LDT (Local Descriptor Table) in addition to GDT (Global Descriptor Table). This causes a problem in paging mode as it needs separate paging structures for LDT. We addressed this problem by eliminating LDT and creating new GDTs for paging. As the Web server uses a single address space and there is a single AO running in the bare PC, this paging model is efficient. However, adding paging to the 32-bit code poses some implementation and testing difficulties especially when the entries have wrong values or are not initialized.

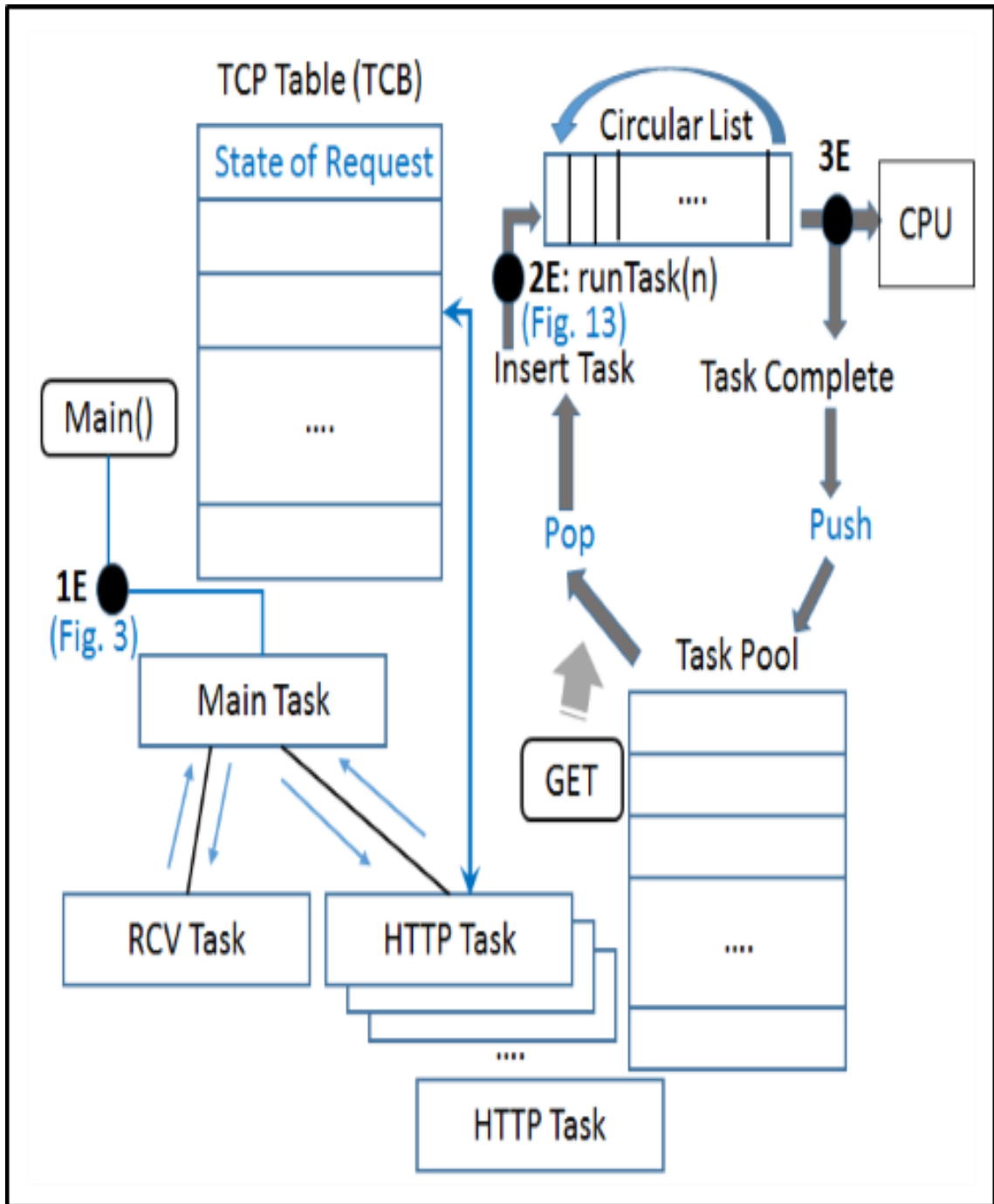


Figure 14: Current Webserver Architecture



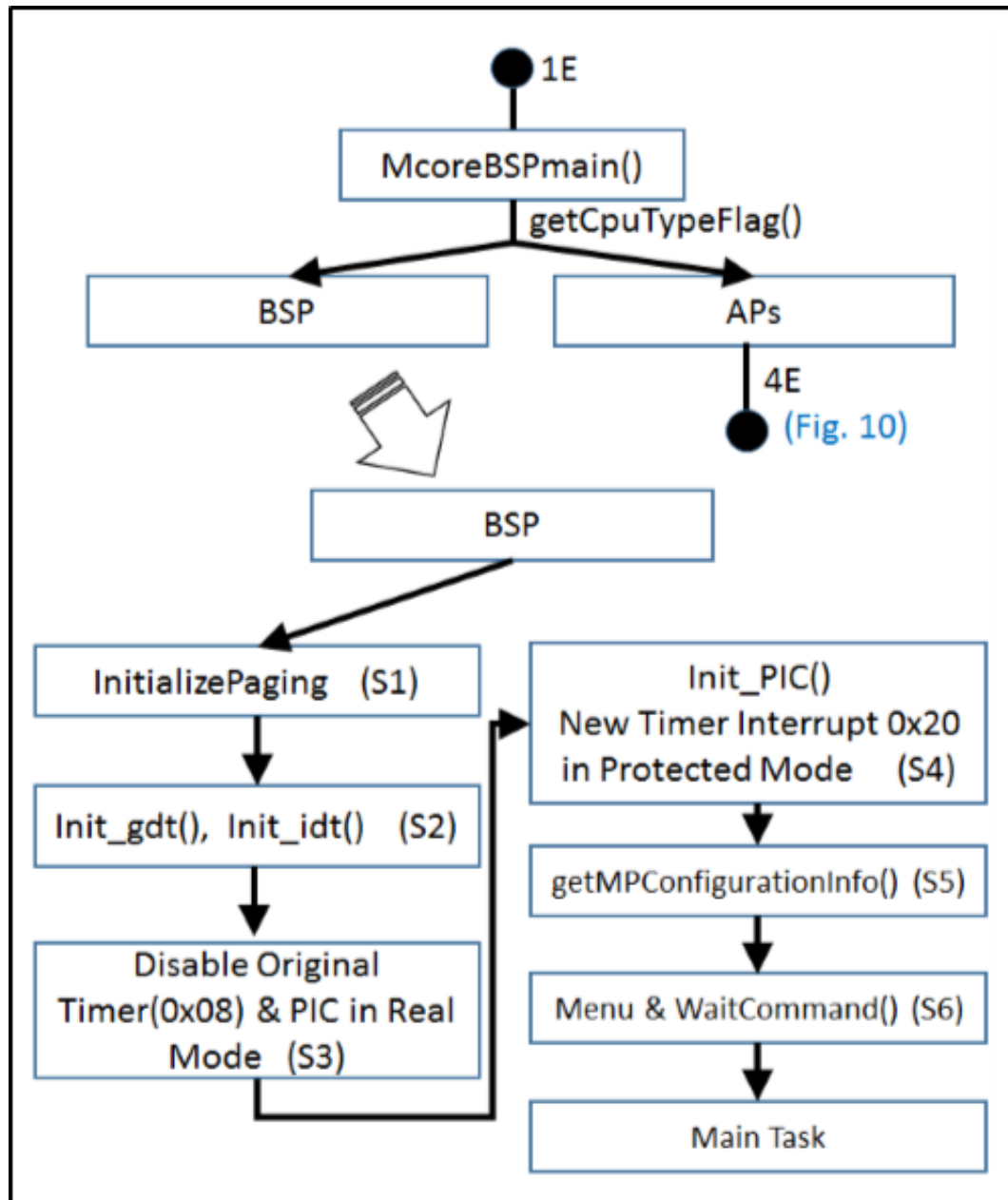


Figure 15: BSP Flow (1E)

### 5.3 MULTICORE MIGRATION

Multicore migration poses many challenges in that a bare PC application has to deal with architectural, design and implementation issues in addition to functional operation, testing, and the unique bare PC programming and computing paradigm. A bare PC Web server is

a complex system that differs significantly from its OS-based counterparts. Figure 14 shows a high-level view of the bare PC Web server architecture. There is a main task (MTSK), receive task (RTSK) and several http tasks (HTSK). The MTSK is always running in the system. When a packet arrives, the RTSK runs as a single thread of execution. The RTSK processes packets as they arrive and updates a table that keeps track of TCP parameters and state (known as the TCB). Each entry in the TCB maintains the status of its corresponding client request from start to finish. A task pool is created and stored in advance in a stack to serve client requests. When a client's HTTP Get request arrives, an existing task from the task pool is popped and inserted into a circular list. When this task is complete, it is pushed back onto the stack for reuse. The 32-bit server can run over 6000 concurrent tasks on a single core Dell Optiplex 260 [19]. The code size is about 636 sectors, which is the entire AO needed for the application. The server is designed strictly as a Web server and only performs HTTP requests, which provides protection against conventional attacks that require an OS and other functionality.

#### 5.3.1 *BSP Flow*

The Web server executable has two separate executables. The first is prcycle.exe (PEX), which has the boot, load and startup code. PEX loads and runs the second executable, test.exe (TEX), which is the Web server application. The display menu enables a user to load, run and debug programs. The Optiplex 9010 uses a quadcore 64-bit processor model. These cores play different roles in the bare PC Web server. The first core is referred to as a boot strap processor (BSP); the other three cores are known as application processors (AP1, AP2, and AP3). Initially, BSP runs the boot code and loads the Web server

application executable. The BSP is automatically configured to run by BIOS (basic input/output system) and APs are not enabled at boot time. In the boot loader code, we set a CPU flag (0x01) at location 0x7c00 (boot address) to identify BSP as a boot processor for the Web server system. When BSP recognizes its role, it resets the CPU flag (0x00) at location 0x7c00 before activating APs.

BSP begins execution at an entry point `main()`, which is same in the 32-bit Web server code. Figure 14 shows this entry point and also an extension point “1E.” This extension point shows the additional code needed for migration (shown in Figure 15). The entry point in BSP starts with the `McoreBSPmain()` function. This reads the CPU type flag (using `getCpuTypeFlag()`) as stored in the boot sector, and identifies itself as a BSP processor as described above. The BSP calls `InitializePaging()` to create and initialize paging data structures (S1). The 32-bit Web server used GDT, LDT and IDT (interrupt descriptor table) entries that were implemented in PEX using TASM code. These entries work only in real mode and the Web server code has interrupt gates to transfer between real and protected modes (real<>protected). Thus, TEX runs in protected mode and PEX runs in real mode. In order to reach hardware interfaces, it uses (real<>protected) mode cycle which serves as a bridge between these two modes.

* TSS DESCRIPTOR in GDT (8 Bytes):							
63							0
Base (H) (1 byte)	Limit (H) (1 byte)	D DPL S (4 bits)	Type (4 bits)	Base (M) 1byte	Base (L) 2bytes	Limit (H) 2bytes	
0x00	0x00	0x8	0x9	0x00	0x0a40	0x006f	Old
0x19	0x00	0x8	0x9	0x20	0x0000	0x0068	BSP
0x19	0x00	0x8	0x9	0x20	0x0068	0x0068	AP1
0x19	0x00	0x8	0x9	0x20	0x00d0	0x0068	AP2
0x19	0x00	0x8	0x9	0x20	0x0138	0x0068	AP3

* Timer & Keyboard Descriptor in IDT (8 Bytes):						
63						0
Handler Offset (H) (2 bytes)	P DPL Type (1 bytes)	Reserved (1 bytes)	Handler Code Selector (2 bytes)	Handler Offset (L) (2 bytes)	INT	
0x0000	0x8f	0x00	0x0098	0x0167	Timer(0x08)	Old
0x0000	0x8f	0x00	0x0138	0x0000	KeyBoard (0x09)	
0x0000	0x8f	0x00	0x0038	0x0b45	NIC (0x73)	
0x0002	0x8e	0x00	0x0030	0x4763	Timer(0x20)	New
0x0002	0x8e	0x00	0x0030	0x4732	KeyBoard (0x21)	
0x0000	0x8e	0x00	0x0038	0x0b45	NIC (0x73)	

Figure 16: GDT, IDT

IDT entries are used to address interrupt service routing and GDT entries are used to address the application's code, data and stack. In order to migrate the 32-bit code, we need to implement new GDT and IDT entries and eliminate LDT (new processors use protected mode GDT and IDT structures). Also, PEX is written in TASM (64-bit compilers do not support this assembler). Thus, new GDT and IDT entries were created and initialized using the `init_gdt()` and `init_idt()` functions (S2). Figure 16 shows sample GDT and IDT entries used in the system to illustrate migration. The new GDT and IDT entries can be used in protected mode, which is required for migration. For example, 32-bit Web server code

used a real mode timer interrupt at 0x08, whereas the new code uses 0x20 interrupt in protected mode. The PIC (programmable interrupt controller) chips have to be disabled (S3) and new APIC (advanced PIC) [45] (p.25) have to be enabled for migration (S4).

```
typedef struct _MPFLOATINGPOINTER
{
    char vcSignature[ 4 ];
    DWORD dwMPConfigurationTableAddress;
    BYTE bLength;
    BYTE bRevision;
    BYTE bChecksum;
    BYTE vbMPFeatureByte[ 5 ];
} MPFLOATINGPOINTER;

typedef struct _MPCONFIGURATIONTABLEHEADER
{
    char vcSignature[ 4 ];
    WORD wBaseTableLength;
    BYTE bRevision;
    BYTE bChecksum;
    char vcOEMIDString[ 8 ];
    char vcProductIDString[ 12 ];
    DWORD dwOEMTablePointerAddress;
    WORD wOEMTableSize;
    WORD wEntryCount;
    DWORD dwMemoryMapIOAddressOfLocalAPIC;
    WORD wExtendedTableLength;
    BYTE bExtendedTableChecksum;
    BYTE bReserved;
} MPMCONFIGURATIONTABLEHEADER;

typedef struct _PROCESSORENTRY
{
    BYTE bEntryType;
    BYTE bLocalAPICID;
    BYTE bLocalAPICVersion;
    BYTE bCPUFlags;
    BYTE vbCPUSignature[ 4 ];
    DWORD dwFeatureFlags;
    DWORD vdWReserved[ 2 ];
} PROCESSORENTRY;
```

Figure 17: MP Configuration Structure

The next step in the migration process is to obtain multiprocessor (MP) configuration information [45] (p.37) (S5). This is implemented in the `getMPConfigurationInfo()` function. This information gathering can be done in many ways [45] (p.38). The MP configuration information is used to manage APs and interrupts. The basic information

stored in the MP configuration includes PIC mode, virtual wire mode via local APIC, virtual wire mode via I/O APIC, and I/O symmetric mode. The information is stored in the Extended BIOS memory area in many places as it is accessed by a floating pointer structure. It is necessary to obtain the MP Floating Pointer Structure Address (e.g. 0x000fda10, \_MPFLOATINGPOINTER), which can be found in one of the three places in memory: (1) first KB of extended BIOS data area (EBDA), (2) within the last KB of system memory (639K-640K for 640K systems and 511K-512K for 512K systems), or (3) BIOS ROM address between 0f0000h and 0ffffh. As the information is “floating” in the above areas, we need to search those areas with a signature \_MP\_ to obtain the information.

In the system used for migration, we found this address in the EBIOS memory area. Using this address, we accessed the MP Configuration Table address (for example, the 0x000fd6d0, \_MPCNFIGURATIONTABLEHEADER). The MP floating pointer structure in this system requires use of the MP configuration table instead of the default MP configuration, and virtual wire mode support instead of PIC mode. The first processor entry structure address can be obtained by adding 0x2c to the MP Configuration Table address (0x000fd6d0), i.e. 00fd6fc. The subsequent processor entries can be found by adding another 20 bytes for each entry. The processor parameters listed in the \_PROCESSORENTRY are used to identify a given processor complex and configuration. Figure 17 shows the MP configuration table data structures, and Figure 18 shows the snapshot for the output of accessing these data structures in the current implementation. The processor parameters obtained in this step are used to manage multi-cores in the system including interrupt vectors and control. The MP configuration structures are complex and suited for

conventional OS-based systems as they use system calls and other libraries. Managing these structures in a bare PC, which uses the bare machine computing paradigm, is non-trivial and requires thorough understanding of the intricacies of 64-bit architecture and multi-core architecture to migrate 32-bit applications.

```
===== MP Configuration Table Summary =====
MP Configuration Table Analysis Success
MP Floating Pointer Address : 0x000FDA10
PIC Mode Support : NO
MP Configuration Table Header Address : 0x000FD6D0
Base MP Configuration Table Entry Start Address : 0x000FD6FC
Total Number of Cores/Processors: 4
ISA Bus ID : 0x3
===== MP Floating Pointer =====
Signature : _MP_
MP Configuration Table Address : 0x000FD6D0
Length : 1 * 16 Bytes
Version : 4
CheckSum : 0xEB
Feature Byte 1 : 0x0 (Use MP Configuration Table)
Feature Byte 2 : 0x0 (Virtual Wire Mode Support Instead of PIC Mode)
===== MP Configuration Table Header (1/2)=====
Signature : PCMP
Length : 700 Bytes
Version : 4
CheckSum : 0xDD
Press any key to continue... ('q' is exit) :

===== MP Configuration Table Header Continue...(2/2)=====
OEM ID String : CBX3
Product ID String : DELL
OEM Table Pointer : 0x0
OEM Table Size : 0 Bytes
Entry Count : 76
Memory Mapped I/O Address Of Local APIC : FEE00000
Extended Table Length : 124 Byte
Extended Table Checksum : 0xB2
Press any key to continue... ('q' is exit) :
```

Figure 18: MP Configuration Data

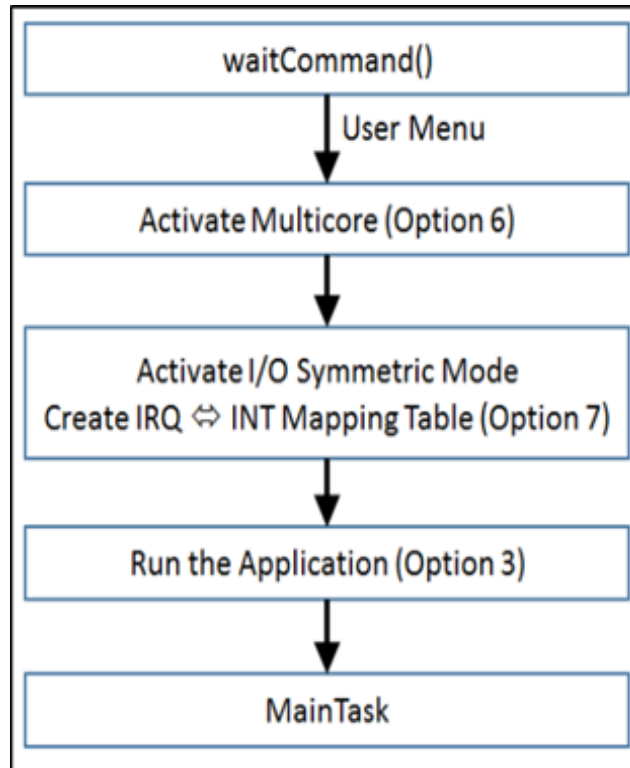


Figure 19: Wait Command Flow

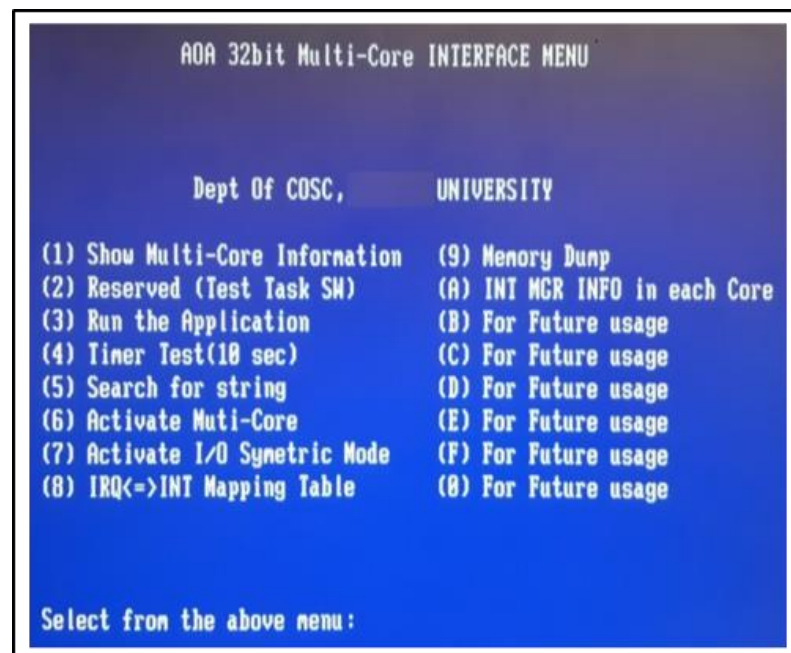


Figure 20: User Menu



===== IRQ To I/O APIC INT IN Mapping Table =====			
PIC Controller		Muliti Core Int Controller	
IRQ[0 ]	-> INTIN [2 ]	:	Timer IRQ to INTIN
IRQ[1 ]	-> INTIN [1 ]	:	PS2 KB IRQ to INTIN
IRQ[2 ]	-> INTIN [255]	:	Slave PIC IRQ to INTIN
IRQ[3 ]	-> INTIN [255]	:	COM2 IRQ to INTIN
IRQ[4 ]	-> INTIN [4 ]	:	COM1 IRQ to INTIN
IRQ[5 ]	-> INTIN [255]	:	LPT2 IRQ to INTIN
IRQ[6 ]	-> INTIN [255]	:	Floppy Disk IRQ to INTIN
IRQ[7 ]	-> INTIN [255]	:	LPT1 IRQ to INTIN
IRQ[8 ]	-> INTIN [8 ]	:	RTC IRQ to INTIN
IRQ[9 ]	-> INTIN [9 ]	:	Reserved IRQ to INTIN
IRQ[10]	-> INTIN [255]	:	Not Used IRQ to INTIN
IRQ[11]	-> INTIN [255]	:	Not Used IRQ to INTIN
IRQ[12]	-> INTIN [12]	:	PS2 Mouse IRQ to INTIN
IRQ[13]	-> INTIN [13]	:	CoProcessor IRQ to INTIN
IRQ[14]	-> INTIN [14]	:	HDD1 IRQ to INTIN
IRQ[15]	-> INTIN [15]	:	HDD2 IRQ to INTIN
Press any key to return to menu...			

Figure 21: Interrupt Mapping Table

The BSP invokes the waitCommand() function (S6), which provides a sequence of commands to run an application as shown in Figure 19. This function provides a user interface to start and debug applications. The actual menu screen is shown in Figure 20. This menu allows a user to activate multicore and I/O symmetric mode (options 6 and 7 respectively), map IRQs to the interrupt table (8), and run the application (option 3). Each processor has its own APIC to control interrupts. The interrupt requests come to an I/O APIC controller, which will be delegated to each processor APIC. Thus, to activate multicore (option 6), the BSP performs two steps. The first step involves reading the MSR register using the “rdmsr” assembly instruction, ORing with the 0x800 value and writing back using the “wrmsr” assembly instruction. In the second step, it activates its own local APIC by changing the spurious interrupt vector register (0xfe00000 (base) + 0xf0 (offset))

with a value 0x100 [45] (p.77). At this point, interrupts are directed to only the BSP. In order to direct interrupts to other APs, we need to activate I/O symmetric mode (option 7). In symmetric mode, it is not possible to use virtual wire mode interrupt routing [45] (p.85).

The MP configuration table must be set up to do appropriate interrupt routing to APICs. This implementation is complex and depends on the APIC architecture and interrupt load balancing. In this system, timer interrupts are directed to APs and all other interrupts are processed by the BSP. The I/O APIC controller IRQ to INTIN map is shown in Figure 21. This map is derived as part of option 7 and also uses the MP configuration table. This map helps to identify interrupt numbers and provides the appropriate interrupt code to process. Finally, when run (option 3) is chosen, the BSP creates MTSK and starts executing that task. At this point, the BSP is in MTSK and all other APs are activated and running their individual idle tasks.

### 5.3.2 *AP Flow*

The BSP takes over the boot process and loading of an application as the boot code had 0x01 in its first byte. When the BSP entered main(), it checked for this flag and took the BSP flow as shown above. When the BSP turns off this flag, all APs are ready to enter main(). Each AP will also go through the McoreBSPMain() function (Figure 15), but they then discover that getCpuTypeFlag() returns a zero value. Thus, they take the AP Flow path as shown in Figure 22. The AP Flow is as complex as the BSP Flow. An AP enters the MainForAP() function and gets its ID by invoking getAPCID(). The ID is stored in the PROCESSORENTRY structure at an offset of 4. The data structures as shown in the MP

configuration [45] (p.37) allow us to compute the address for each core and its related PROCESSENTRY to access the core id. In our system, the IDs for APs are 0, 2, 4 and 6. Figure 18 shows these IDs and the rest of the parameters read from the MP configuration structures.

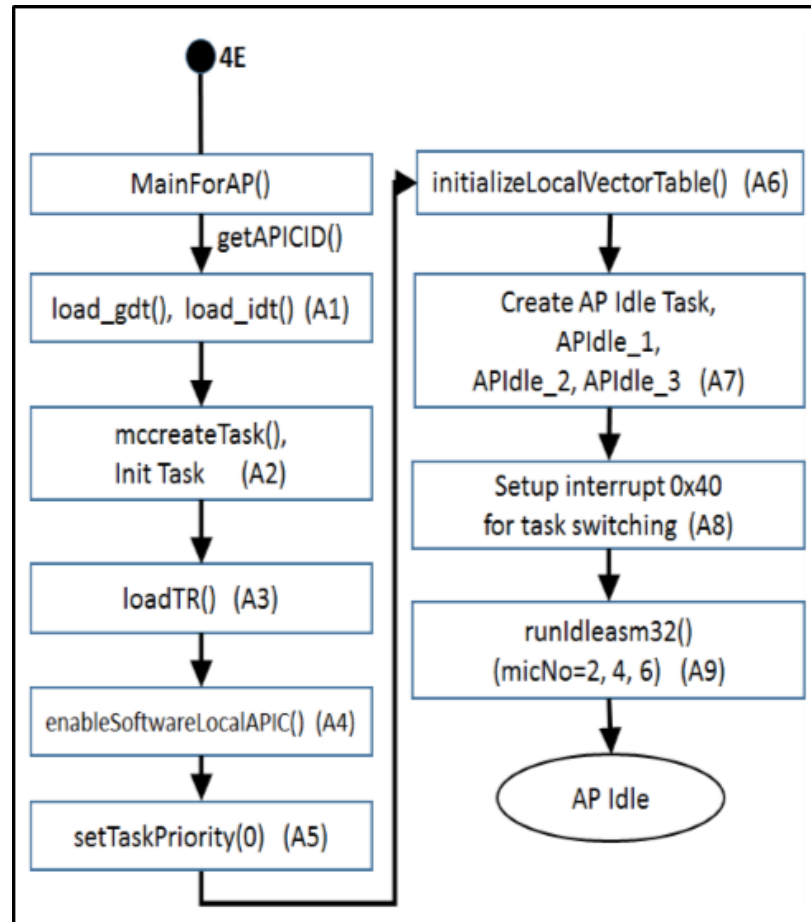


Figure 22: AP Flow

Each AP needs GDT and IDT tables to work with the Web server application. These tables are initialized before by the BSP using the `init_gdt()` and `init_idt()` functions (A1). In this case, we simply load the entries by using the functions `load_gdt()` and `load_idt()`. All processors have access to all resources of the Web server as they are running a shared memory multi-processor model. The AP then creates an initial task by using `mcreateTask()`

(A2). The loadTR() function will load the TR (task register) to prepare to run a task (A3). The enableSoftwareLocalAPIC() function will activate its own local APIC by changing the spurious interrupt vector register (0xf0000000 (base) + 0xf0 (offset)) with a value 0x100 (A4) [45] (p.75~87). The setTaskPriority(0) function will set zero value in task priority register (0xf0000000(base) + 0x80 (offset)) to disable processing interrupts in the AP (A5).

The initializeLocalVectorTable() function (A6) [45] (p.75~87) is difficult to implement due to its complexity. The actual code is shown in Figure 23. This function has seven intricate steps needed to set up interrupt capabilities for the core. Step 1 sets up the base address, which is derived from the MP configuration table (i.e. 0xf0000000). Step 2 blocks the timer interrupt by setting the mask value 0x010000 into the LVT (local vector table) timer register with an offset of 0x320 (i.e. address 0xf000320). Step 3 is used to block LVT0 interrupt, for which its register address is located at (0xf000350). Step 4 is used to setup this interrupt as NMI (non-maskable) interrupt, for which its register address is located at (0xf000360). Step 5 is used to block the error interrupt for which its register address is located at (0xf000370). Step 6 is used to block the performance monitoring interrupt, for which its register address is located at (0xf000340). Finally, Step 7 is used to block the temperature sensor interrupt, for which its register address is located at (0xf000330). We blocked most of the interrupts in the AP to facilitate and debug multicore processors. In a completely migrated system, these interrupts must be enabled and handled accordingly to capture errors and other issues relevant to a high-performance system.

```

void initializeLocalVectorTable( void )
{
    DWORD dwLocalAPICBaseAddress, dwTempValue, dwSetVal;
    // Step 1.
    dwLocalAPICBaseAddress = zGetLocalAPICBaseAddress();
    // Step 2.
    dwSetVal=getcmpaddrasm32(dwLocalAPICBaseAddress +
        APIC_REGISTER_TIMER);
    setLocalAPICasm32(dwLocalAPICBaseAddress +
        PIC_REGISTER_TIMER, dwSetVal|APIC_INTERRUPT_MASK);
    // Step 3.
    dwSetVal=getcmpaddrasm32(dwLocalAPICBaseAddress +
        APIC_REGISTER_LINT0);
    setLocalAPICasm32(dwLocalAPICBaseAddress +
        APIC_REGISTER_LINT0, dwSetVal|APIC_INTERRUPT_MASK);
    // Step 4.
    dwSetVal=getcmpaddrasm32(dwLocalAPICBaseAddress +
        APIC_REGISTER_LINT1);
    setLocalAPICasm32(dwLocalAPICBaseAddress +
        APIC_REGISTER_LINT1, dwSetVal|APIC_TRIGGERMODE_EDGE |
        APIC_POLARITY_ACTIVEHIGH | APIC_DELIVERYMODE_NMI);
    // Step 5.
    dwSetVal=getcmpaddrasm32(dwLocalAPICBaseAddress +
        APIC_REGISTER_ERROR);
    setLocalAPICasm32(dwLocalAPICBaseAddress +
        APIC_REGISTER_ERROR, dwSetVal|APIC_INTERRUPT_MASK);
    // Step 6.
    dwSetVal=getcmpaddrasm32(dwLocalAPICBaseAddress +
        APIC_REGISTER_PERFORMANCEMONITORINGCOUNTER);
    setLocalAPICasm32(dwLocalAPICBaseAddress +
        APIC_REGISTER_PERFORMANCEMONITORINGCOUNTER,
        dwSetVal|APIC_INTERRUPT_MASK);
    // Step 7.
    dwSetVal=getcmpaddrasm32(dwLocalAPICBaseAddress +
        APIC_REGISTER_THERMALSENSOR);
    setLocalAPICasm32(dwLocalAPICBaseAddress +
        APIC_REGISTER_THERMALSENSOR, dwSetVal|APIC_INTERRUPT_MASK);
}

```

Figure 23: Initialize LVT

Once the interrupts are set properly, we can create an idle task which will run in the AP as a never-ending loop in the processor (A7). We created three idle tasks (APIdle\_1,

APIIdle\_2, APIIdle\_3) with their own GDT indexes (i.e 0x208, 0x210, 0x218) and their corresponding GDT entries. We have also created one IDT entry for each AP; an interrupt vector 0x40 operates as a task gate to do task switching from init task to idle task (A8).

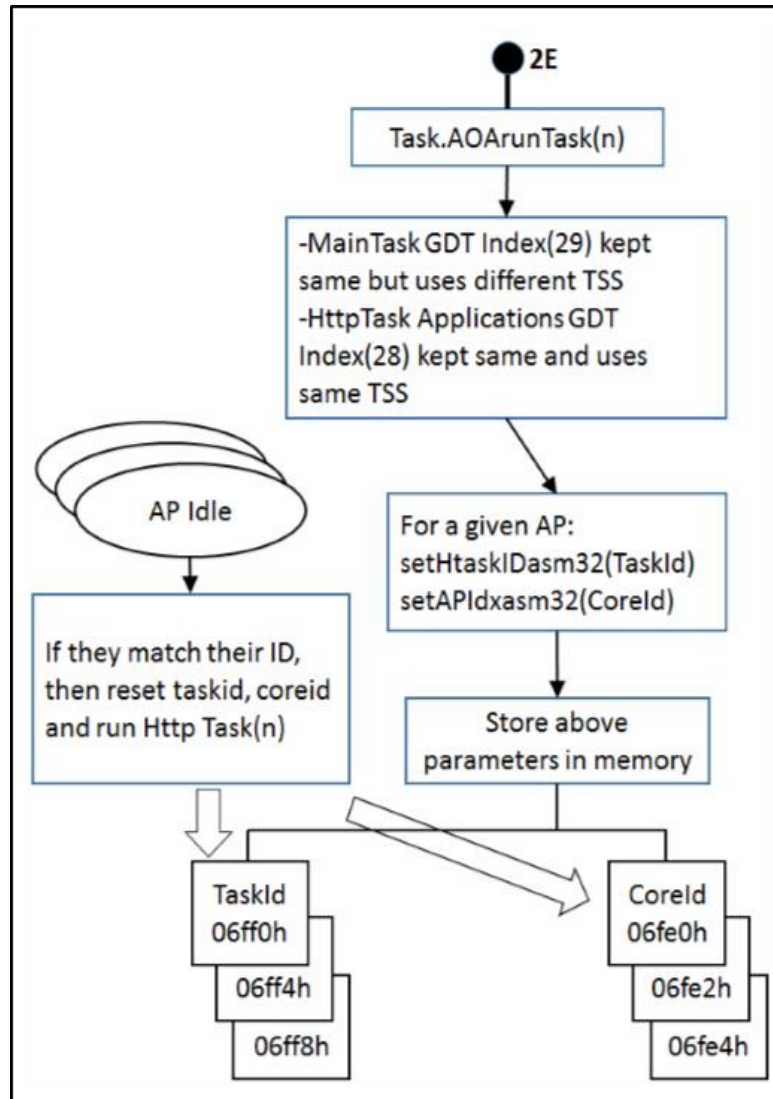


Figure 24: AP IDLE Task

```

// Step 1. -----
void AOATask::APIdle_1(long z1)
{
    int i, iCoreIDX;
    int previous_task;
    i=0;
    iCoreIDX=0;
    previous_task=0;
    while(1) {
        iCoreIDX=io.AOagetAPIdxasm32();
        i=(int)io.AOagetHTakIDasm32();
        if (previous_task==i)
        {
            ;
        }
        else{
            if (iCoreIDX==1) //iCoreIDX = index of the APIC ID
            {
                io.AOasetAPIdxasm32(0x00);
                io.AOasetHTakIDasm32(0x0000);
                runTask(i);
                previous_task=i;
            }
        }
    }
};

// Step 2. -----
void (_thiscall AOATask::*pFunc1)(long) = &AOATask::APIdle_1;
void* p1Ptr = (void*)&pFunc1;
Function_Address_Array[0] = (int)p1Ptr;

// Step 3 -----
if (micNo==2) //micNo = APIC ID
{
    idlesel=(TASK_GDT_IDLE+io.AOagetCoreIDX(io.AOagetAPICID()))*8;

    task.AOAmccreateTask((long)Function_Address_Array[0],
        (int*)(AP_STACK_BASE_ADDR2+io.AOagetAPICID()*0x10000),
        idlesel, TASK_GDT_IDLE+io.AOagetCoreIDX(io.AOagetAPICID()), 0);

    io.AOAmsetidtentry(0x40, 0, idlesel, MCATR_TASKGATE);
}

// Step 4. -----
io.AOArunIdleasm32();

```

Figure 25: Scheduling (2E)

Each AP will run its own idle task and monitor for any tasks assigned to it by the BSP. In order to go to idle task, we need an interrupt 0x40 to do task switching. The idle task will be running and waiting for any future tasks to be assigned by the BSP processor. The `runIdleasm32()` will enable the task switch from init to idle task by invoking interrupt 0x40. Figure 24 illustrates some details of creating the idle task in the AP. Step 1 in the figure shows the `APIIdle_1` function, where there is a never-ending loop to serve BSP requests. The address of this function is captured in Step 2 and stored in an array. Step 3 shows task creation for AP1. In this step, we create a task using the `AOAmccreateTask()` function, which needs function address as captured in the array, stack address, GDT entry no and task id. We also set up an IDT entry by the `AOAmsetidtentry()` function that is needed for task switching from init to idle task. Finally, Step 4 shows the invocation of interrupt 0x40 that will switch from init to idle task. The process described above enables AP to go from start to running idle task (A9). Now, each AP is running in their own idle task and they are ready to process the HTTP task requested by the BSP. The while loop in Step 1 compares its own AP identification with the one waiting to run, then takes that HTTP task and runs it using the `runTask(i)` function.

### 5.3.3 Task Scheduling Flow

This section describes the migratory code added for the Web server at point “2E” in Figure 14. When a Get request arrives from an HTTP client, the server pops an HTTP task (HTSK) from the task pool and inserts it into a circular list. Each request is an independent HTSK that handles data transfer and termination of the task. The connection part of the HTSK is done by RTSK. The MTSK and RTSK run on the BSP only. When there is only



one core, all tasks are in the same processor. Figure 25 shows the task scheduling flow altered from the 32-bit Web server code.

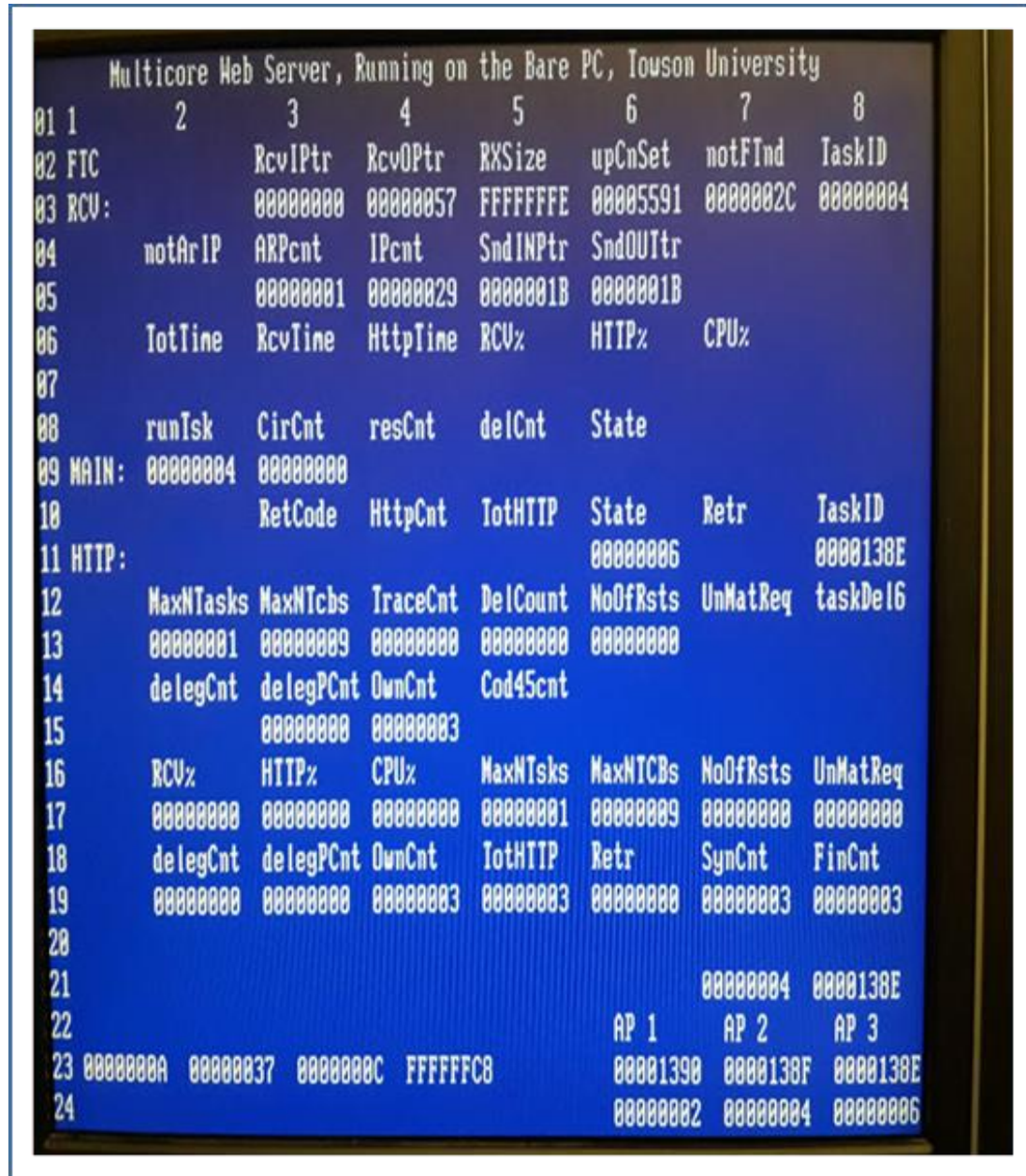


Figure 26: Multi-core Bare PC Display

When a HTSK is ready to be dispatched, it calls the function runTask(n) to run a task with task id n. In order to run task n, we have to do task switching from MTSK to HTSK.

The MTSK GDT index (in the 32-bit server) is 29. The new code uses the same GDT index with a new TSS. In the 32-bit server, HTSK has its own GDT index (28) and its own TSS. The new code uses the same GDT and TSS structures with no change. The task switch logic is not changed for migration. When APs are enabled, for a given AP, we set task id and core id in a shared memory location for each processor, where a core can access its own coreid. `setHTaskIDasm32(taskid)` and `setAPIdx2asm32(coreid)` are functions that will store the values in shared memory at 0x06fe0 and 0x06ff0 for core 1. Similarly, for core 2 and core 3, the values are stored in shared memory at different locations. As described before, APs are running in an APIIdle loop and check for the match of their coreid in the shared memory. If the coreid matches, the core will invoke `runTask(n)` to run the HTSK. In a given AP, when the HTSK is complete, it returns back to the idle task.

In the 32-bit Web server, there is only one processor to schedule MTSK, RTSK and HTSK. As shown in Figure 14, “3E” connection point illustrates the new configuration for migration. In this case, we have BSP, AP1, AP2 and AP3 processors connected to the circular list. These execution elements have to be load balanced by the logic shown in Figure 25. In the current model, we do not perform any optimal load balancing techniques, but simply allocate HTSKs to APs in a round robin manner.

This section described many aspects of migration needed for the Web server code to run on multi-core processors in detail. We found that the migration path is complex to implement not only in a bare PC but also in OS environments.

## 5.4 FUNCTIONAL OPERATION AND DATA

Results of performance studies using the 32-bit bare PC Web server in single mode and in split server mode are given in [3] and [18] respectively. This Web server is implemented using C/C++/MASM/NASM languages. The new code implemented for migration includes 1,595 lines of assembly and 9,366 lines of C/C++ code (with comments). Additional code will be needed for optimization and load balancing of cores in the future. The server is tested on an Optiplex 9010 with 4 core for functional operation. It is tested using an IE (Internet Explorer 11; it can be any other browser) browser and the http\_load stress tool. This section includes functional testing measures and data to validate the design and implementation of migration.

### 5.4.1 *First Migration Attempt*

The first migration attempt does not include any multi-core operation. The http\_load stress tool is used to collect this data. This data shows the stress tool output for 600 seconds requesting 1000 requests per second. The important data to be considered is mean connect time (0.185272 milli-secs) and first response time (0.169373 milli-seconds) measured in this run.

### 5.4.2 *Single Core with Paging*

In this model, http\_load was run for 1000 requests per second over a 10-minute period. The output shows the mean connect time is 0.193595 milli-secs and the first response time is 0.145881 milli-secs. This is a little slower than in the first attempt in connect time, but faster in the response time. We also run this with and without paging. There is not much of

a difference in their performance as all page entries are populated for the given size of memory, and there were no page faults in the bare implementation.

#### 5.4.3 *Multi-Core*

We have activated 3 APs and tested the multi-core operation for its functionality. We did not run `http_load` on this system, as we have not addressed synchronization and concurrency control mechanisms for multi-core processors in a bare PC environment. This issue along with performance benchmarking will be addressed in the future. The requests coming from IE will be processed in a round robin fashion by AP1, AP2 and AP3 cores. The BSP processor will only run MTSK and RTSK as described earlier. The core ids are 0, 2, 4 and 6 for BSP, AP1, AP2 and AP3 processors respectively. We show results of making IE browser requests along with a bare PC screen shot and a Wireshark capture for one request. The bare PC screen shot is shown in Figure 26. Notice the core ids for APs are 2, 4, and 6 and the task ids shown are 0x1390, 0x138f and 0x138e. The screen shows 3 requests dispatched to 3 cores. The rest of the details on the screen are bare PC related debug traces to show the activity in the Web server. The IE browser makes a request for a .gif file (Bare Rocks), which is about 10K in size. The Wireshark output shows the details of the client request and that the time taken for a single request is 0.058504 secs as shown in Figure 27. This is not a true measure of multi-core performance as it only compares a single request versus a stress tool that produces a large number of requests. However, it is evident that multi-core Web server performance should be better compared to a single processor. The ultimate performance will also depend upon the implementation of synchronization and concurrency control mechanisms needed for multi-core in a bare PC

environment. There are fewer shared memory conflicts in a bare PC than in a conventional OS-based system. The preliminary results above show that it is possible to build bare PC Web servers using multi-core processors to achieve higher performance with the added benefits of the bare machine paradigm including simplicity and security.

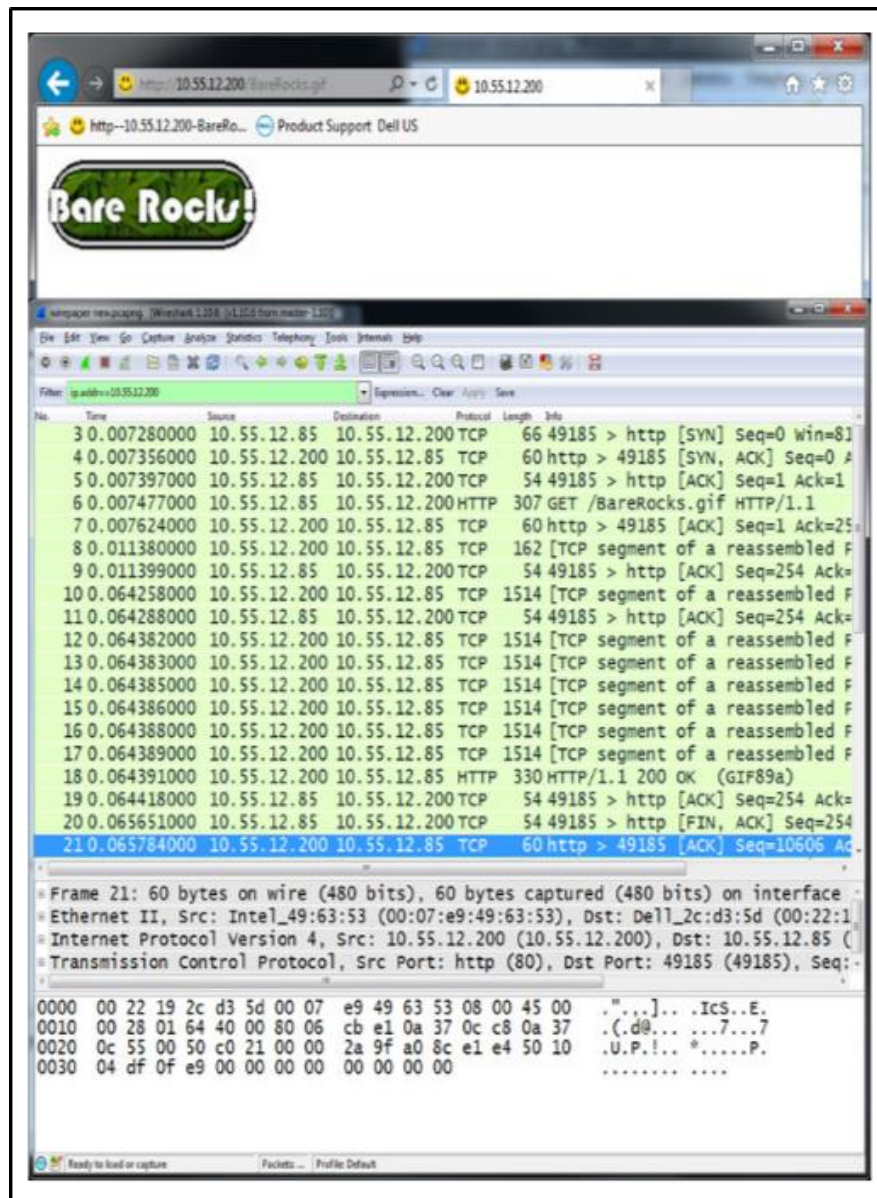


Figure 27: Multi-core Wireshark Output for a Client Request

## 5.5 DISCUSSION

In this section, we make some general observations concerning migration in a bare PC environment and discuss a few avenues for future research. Our migration approach results in a unique bare machine Web server design that leverages the multi-core architecture. In particular we note that:

- (1) On any system, the migration process itself is complex and poses many challenges.
- (2) Many intricacies are involved in implementing the multi-core architecture on a bare PC.
- (3) The implementation details provided will be useful not only for bare PC systems but also for other application domains and platforms such as sensor networks and embedded systems, where there is no need for a conventional OS.
- (4) Migration provides a basis for future research in innovative multi-core designs for bare machine applications.
- (5) The first attempt at migrating bare PC code illustrates the flexibility of this code and the ability to make it work with just a few changes on a 64-bit system when 64-bit/multi-core features are not used.
- (6) Several novelties of bare PC applications are evident in the migration outcomes. For example, we have seen that it is possible for an end user program to have complete control of applications, interrupts, GDT, IDT, MP configuration, and direct hardware interfaces.
- (7) The multi-core code developed is completely independent of any other environment, application, or external software i.e. it only depends on the underlying multi-core processor and Intel architecture.

We only considered the migration of a bare PC Web server to an Intel multi-core architecture. However, to optimize performance it is also necessary to investigate synchronization and concurrency control mechanisms needed for bare PC applications running on multi-core. Other bare PC applications can be similarly migrated to multi-core using our implementation methodology. In the future, we will attempt to apply the split-server concept to multi-core by splitting protocols, applications and individual components. We end this discussion by noting that the bare machine computing paradigm has the potential to exploit the upcoming growth in multi-cores to build simple, scalable, secure and high-performance systems in the future.

## 5.6 SUMMARY

We presented a novel approach for migrating a bare PC Web server from a single processor system to a multi-core system. Our work also demonstrated a first step in migration that does not require any multi-core features to be implemented. We further showed how paging could be implemented in a basic bare PC system to make it compatible with the 64-bit architecture. In particular, we detailed how to run applications on multi-core processor systems and configure, control and manage their internal elements, including registers and control data needed for implementing a working system. This paper also discussed novel features of this approach and some possibilities for future research. The methodology used for migrating the 32-bit Web server will enable existing and future bare PC applications to be run on a multi-core 64-bit architecture.

## 6 MULTI-CORE BMC PARALLELIZATION

A Web server designed for a single core 32-bit processor is used to study parallelization of the server on a multi-core CPU. It is essential to understand the existing BMC architecture for the Web server before studying the design issues for parallelization. The following sections provide insight into the BMC Web server design and issues faced with parallelizing this server.

### 6.1 *BMC Tasking*

BMC applications need multi-tasking to cope with I/O events and also concurrency. A BMC programmer controls the tasking structure and its operation. When a BMC application suite is running, nothing else is running in the machine, thus the tasking structure is different than in a conventional system. A task in BMC is similar to a thread in OS based systems, as the entire BMC application suite runs as a single process in one address space. The task structure is shown in Figure 28. A task pool is created a-priori based on its need and application suite. There can be many task pools, and each pool performs specific tasks. The task structure consists of three task types. The main task in the system runs at all times to handle other tasks. If there is a network interface in the application, then a receive task runs, whenever a packet arrives. The receive task will process the packet arrived and update the state of an application and return to the main task. The receive task runs as a single thread of execution without waiting for any resources. The running tasks will be placed in a ready queue to be run as the CPU gets available. A circular list data structure is used to maintain tasks to run.



As noted before, a given task runs as a single thread of execution until its next state. If the next state is an end of task, then it will be complete. If the next state is not the end of task, then it will suspend and return to the main task. When a task is running, it will not be interrupted until its thread is suspended. An application programmer decides this execution path during its design.

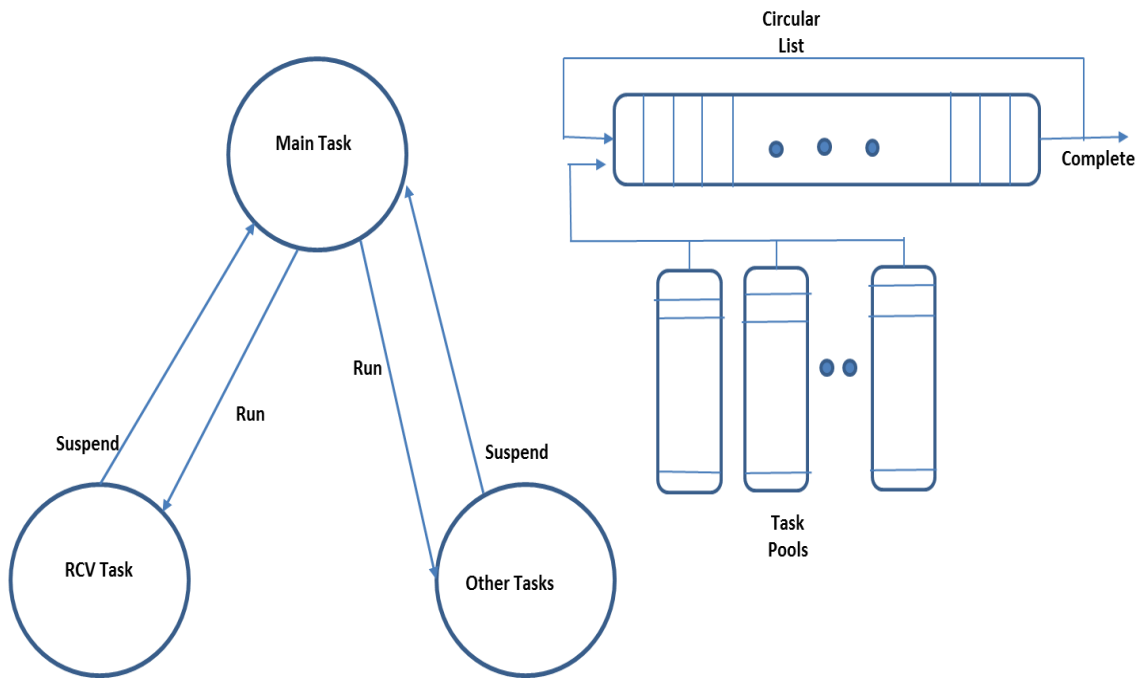


Figure 28: BMC Task Structure

Each task has a TSS (task segment state) which is located in main memory. The task API will manage both TSS in the CPU and other task related functions. Some of the API calls include: insert task into circular list, remove task from the circular list, push and pop from the task pool (which is also in main memory), run task, suspend task and complete task. This approach is different from that of a conventional OS or kernel based system, as the application programmer controls the execution plan during its program design phase.

## 6.2 *Scheduling*

A simple FIFO (first in first out) strategy is used in BMC applications as shown in Figure 28. This works fine as tasks run as a single thread of execution until it is complete or suspended. Each execution cycle is a productive run as it does not waste time in any “wait” condition. There is no need for complex scheduling algorithms in this system. As the scheduling is FIFO, there are no special extensions needed in the CPU. However, we can add other scheduling algorithms (round robin and priority) for special purpose applications through microcode and architectural extensions.

## 6.3 *Existing Inter-Process Communication*

Within the BMC application suite, tasks (processes) need to communicate to perform cooperative work. Thus, this communication will be provided through shared memory facilities. A shared memory can be designed to provide a tag and data field to be accessed by given task. A programmer defined, shared memory area will be provided to inter-process communication. This memory area will be updated atomically to avoid any locking requirements in the software.

## 6.4 *Multi-core Inter-Process Communication*

Multi-core inter-process communication poses different challenges to enable the existing Web server to run on multiple cores. In a single core design, synchronization and locking is avoided by the use of a single thread of execution before taking an interrupt as described before. In a multi-core design, a new software architecture is needed to avoid the majority of the synchronization and locking. In a multi-core design (four core used in our

system), the BMC software architecture needs to be redesigned to parallelize the system. We tried a variety of software architectures with no successful results. The Fig. 29 shows an architecture that works but has poor performance due to unknown causes that need further investigation.

Figure 29: Multi-core Inter-Process Communication

In four core processors, one processor is designated as BSP (Bootstrap Processor) and others are known as application processors (APs). The BSP is responsible for boot and interrupts and other centralized controls. In our Web server system, BSP is used to receive packets and serve as a RCV task. The Web server request operation can be as shown in Fig. 29, illustrating a client request to a server. For each client request, a unique entry in TCP table (known as TCB) is used to keep track of the status of the request from the beginning to the end.

In order to avoid concurrency control and reduce locking, TCB tables are duplicated in each core. BSP receives a SYN packet and makes the decision for partitioning the load. The partitioning algorithm is very simple as it is, based on a round robin for 3 cores. This can be made sophisticated for further research. When the SYN packet arrives, it is placed in an appropriate FIFO (first in first out) for a given AP (FIFO1, FIFO2, or FIFO3). Each AP receives from its own FIFO (by locking the FIFO, contention with BSP) and processes a request. Each request from FIFO is placed in its own circular list for processing until its completion. When a request is complete, it deletes its own TCB entry and sends a message to delete an entry at the BSP. The BSP is attached to the receive task and the Ethernet card to receive packets. Thus, all send and receive packets are done through BSP. It is possible that there is also a need for a send FIFO at the BSP to preserve the order of sending packets. It is also possible that there is a need for FIFO between APs and BSP to communicate between AP and BSP.

Considering the proposed architecture, the performance of multi-core was not any better than that of a single core. We found that only one core was running at time and the cause was not identified. This problem needs to be resolved to fully take advantage of multi-core in a BMC Web server.

Observations:

(1) Multi-core CPU architecture is very complex and hard to comprehend

BSP controller approach is a bottleneck

(2) Multiple cores communicating to a single network card is a bottleneck

- (3) Multi-core design focuses on CPU performance, while ignoring the parallelism with network card
- (4) Shared memory in multi-core is also a bottleneck
- (5) The BMC Web server was designed for a single core, and avoided concurrency control and locking by design; this approach failed for parallelizing the Web server on multi-core
- (6) The BMC Web server chosen to parallelize on multi-core may be a wrong application for parallelizing
- (7) CPU intensive BMC applications may be suitable for multi-core environment
- (8) More investigations are needed to make any additional concluding remarks on this problem.

## 7 CONCLUSION

Bare machine computing applications are designed for 32-bit Intel x86 CPU architectures. The BMC applications directly communicate to hardware without any middleware, centralized OS, or kernel. A variety of applications were developed in the BMC laboratory. This thesis explored the possibility of porting these BMC applications to run on a 64-bit processor with multi-core CPUs. The porting process had many challenges starting from the boot code, loader and multi-core configurations. The Webserver was successfully ported to run on multi-core with four cores. This work laid a foundation to address some of the issues in porting BMC applications. Many design issues were found and resolved. However, the Webserver was not parallelized due to technical issues and time constraints of the researcher. Further research is needed to address some of these issues.

## APPENDIX

This appendix shows numerous details and screenshots of the developmental environment for BMC applications. This will be very useful for people who would like to experiment on their own development of bare machine applications.

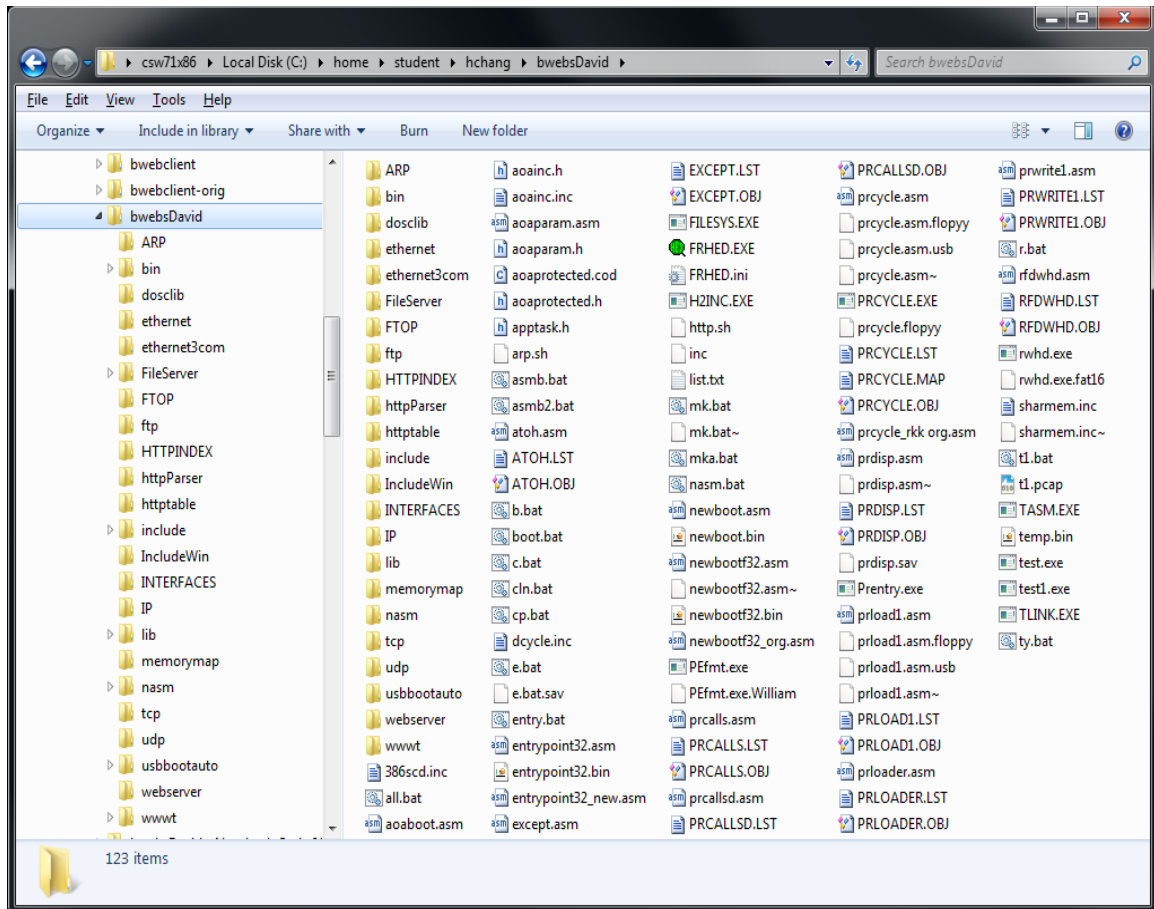


Figure 30: Bare PC Structure

### A. DIRECTORY STRUCTURE

The development process is simple and straight forward. Each directory has its own class definition and implementation files. Some directories have more than one header and source files. The main directory has assembly code related to a menu program. The menu

program helps the developer to understand debug applications. The /webserver directory has the main program and its related task programs. This directory also has a link batch file. The directory structure is shown in Fig. 29.

## B. COMPILATION ENVIRONMENT

The BMC applications use a standard C++ (Visual Studio Library without a need for any header files). There are DOS batch files to compile in command line mode. Whenever a file is changed in a given directory, it should be compiled with a batch file. The asm.bat compiles assembly files, the cpp.bat compiles C/C++ files and the ln.bat links all the object files.

These batch files are shown below:

### a) Assembly Compiling Batch File

All assembly code for the file system and device drivers is compiled using the following batch file. It uses MASM 6.11 assembler.

```
cls
REM needs MASM 6.11 Environment
REM Run the batch file MASM.BAT
ml /c /Cx /coff /Fl runTaskasm32.asm runtaskasmap1.asm
                        runtaskasmap2.asm runtaskasmap3.asm
                        chkstk.asm
nasm newbootf32.asm -onewbootf32.bin
nasm -f bin -o entryptpoint32.bin entryptpoint32.asm -l list.txt
```

Figure 31: Assembly compiling batch file



## b) C/C++ Compiling Batch File

All C/C++ code for the file system and device driver code were compiled using the following batch file.

```
cls
REMit needs microsoft visual C++ environment.rem run the batch
REM file msdn.bat
..\bin\cl /c /Fa /ZI /Gs test.cpp aoatask.cpp apptask.cpp runTask.c
wcirlist.cpp wlinlist1.cpp wlinlist2.cpp wlinlist3.cpp wstack.cpp
wstack1.cpp wstack2.cpp wstack3.cpp wlist.cpp intexception.cpp
wtrace.cpp wserverlist.cpp mtask.c wlist1.cpp wpktlist1.cpp
```

Figure 32: C/C++ Compiling Batch File

```
cls
rem Needs Microsoft Incremental Linker (Microsoft Visual C++ Linker)
erase ..\dosclib\*.obj
call cplib.bat
..\bin\link /MAP /BASE:0x00000000 /NODEFAULTLIB /OPT:NOREF
/MERGE:.rdata=.data /MERGE:.bss=.data /STACK:32000000
/LIBPATH:"..\dosclib" /ENTRY:main test.obj aoatask.obj apptask.obj
aoaprotected.obj runTask.obj runTaskasm32.obj runtaskasmap1.obj runtaskasmap2.obj
runtaskasmap3.obj etherobj.obj cisupport.obj isupport.obj chkstk.obj wcirlist.obj
wstack.obj wtrace.obj wlist.obj ARPObj.obj IPObj.obj rand.obj tcpobj.obj tcpobj1.obj
tcpobj2.obj tcpobj3.obj parserobj1.obj parserobj2.obj parserobj3.obj fhashindex.obj
udpobj.obj ftpobj.obj ftpobj.obj intexception.obj cfiles.obj asmfiles.obj thashindex.obj
thashindex1.obj thashindex2.obj thashindex3.obj wserverlist.obj MCore.obj
PicTimer.obj keyboard.obj page32test.obj mtask.obj nasmfuncs.obj MPCfgTable.obj
wlist1.obj wlinlist1.obj wlinlist2.obj wlinlist3.obj wstack1.obj wstack2.obj wstack3.obj
wpktlist1.obj
dir test.exe
```

Figure 33: Link Batch File

### c) Link Batch File

The batch file in Fig. 32 was used to link all modules in the bare PC system. Notice that this linker batch file uses much more object files than needed for the file system. This is because we tested the multi-core system with a larger application that uses a networking application such as a Web server.

### d) Makefile

The following mk.bat file is the most important file in the system. It copies needed files to a flash drive and makes the flash drive bootable. LoaderSize1 in the prdisp.asm may need to be changed if the test.exe executable file size changes. The loader loads exactly the number of sectors needed for the executable. If you load less sectors, the application will show some erroneous data on the screen.

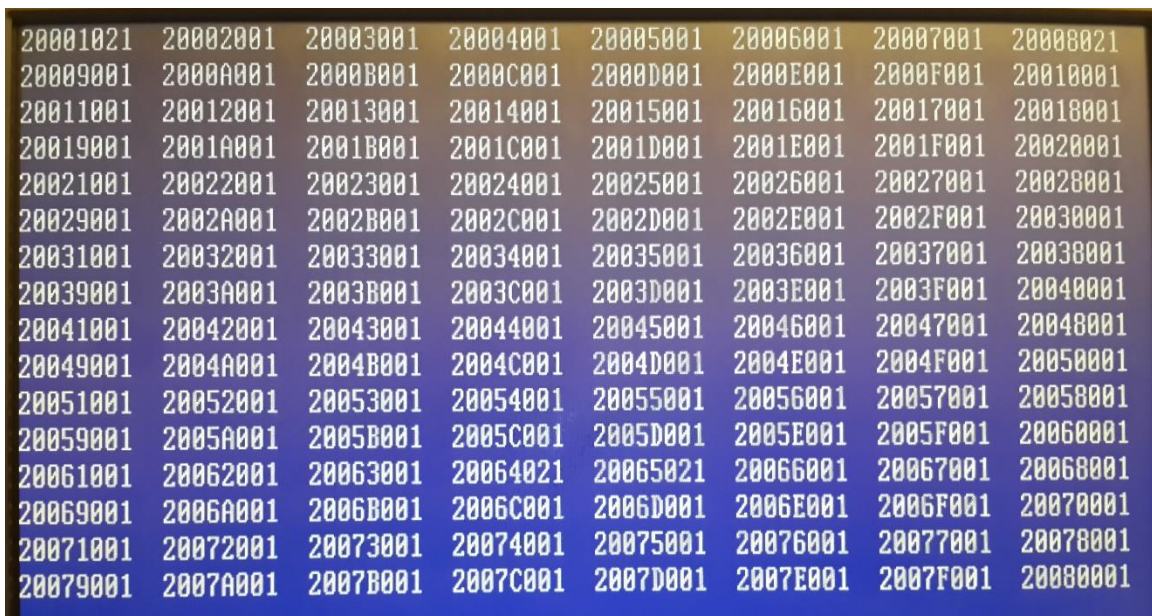
```
call asmb2.bat
call c.bat
format E: /FS:FAT32 /q
copy prcycle.exe E:
copy .\webserver\test.exe E:
REM drive E is 5
rwfd -m 5 newbootfat32.bin
```

Figure 34: Make File

## C. DEBUGGING PROCESS

The most common ways to debug BMC applications are to use memory dump and print statements. At any point in time during execution, you can enter CNTL-Q to return to the

main menu in a single core. Multi-core does not allow that application to return to menu if BSP is not active. The memory dump option is (9) in the menu. You can dump memory as shown in Fig. 34. There are a variety of print statements that can be used in the code to debug programs. The most commonly used ones are AOAPrintText(“Hello”, Line2+20) and AOAPrintHex(value, Line3+20). The display can be only for text and there is no graphics support. The display is divided into 25 lines and 160 bytes in the column. One can display up to 80 characters in one line, thus making each character require 2 bytes. One byte is required for the characteristics of a character and another byte is required for the character code. For example, 16 bytes are required to display 0x12345678.



20001021	20002001	20003001	20004001	20005001	20006001	20007001	20008021
20009001	2000A001	2000B001	2000C001	2000D001	2000E001	2000F001	20010001
20011001	20012001	20013001	20014001	20015001	20016001	20017001	20018001
20019001	2001A001	2001B001	2001C001	2001D001	2001E001	2001F001	20020001
20021001	20022001	20023001	20024001	20025001	20026001	20027001	20028001
20029001	2002A001	2002B001	2002C001	2002D001	2002E001	2002F001	20030001
20031001	20032001	20033001	20034001	20035001	20036001	20037001	20038001
20039001	2003A001	2003B001	2003C001	2003D001	2003E001	2003F001	20040001
20041001	20042001	20043001	20044001	20045001	20046001	20047001	20048001
20049001	2004A001	2004B001	2004C001	2004D001	2004E001	2004F001	20050001
20051001	20052001	20053001	20054001	20055001	20056001	20057001	20058001
20059001	2005A001	2005B001	2005C001	2005D001	2005E001	2005F001	20060001
20061001	20062001	20063001	20064021	20065021	20066001	20067001	20068001
20069001	2006A001	2006B001	2006C001	2006D001	2006E001	2006F001	20070001
20071001	20072001	20073001	20074001	20075001	20076001	20077001	20078001
20079001	2007A001	2007B001	2007C001	2007D001	2007E001	2007F001	20080001

Figure 35: Memory Dump to debug

#### D. USER’S GUIDE AND SCREEN SHOTS

Place your flash drive in one of the USB slots and power on the machine. A menu as shown in Fig.x shows on the screen after the system is booted. Use option 2 to load the application and option 3 to run it. This menu is designed to help the programmer to

understand the process and debug as needed. The test.exe can be one or some group of applications compiled as a single monolithic executable. The application only carries code that is needed for a given application suite. As there is no centralized software running, a given application manages itself when it is running.

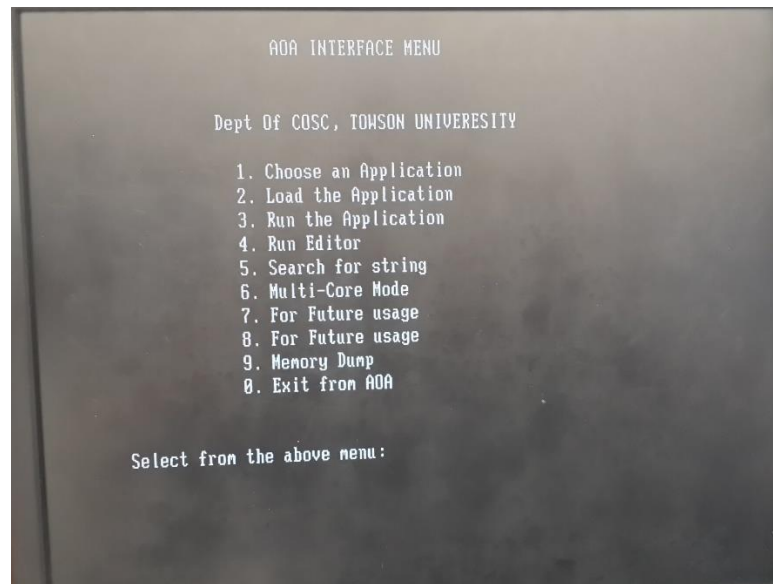


Figure 36: Bare PC Menu



Figure 37: Load (Option (2))

CS Web Server, Running on the bare PC, Towson University							
01 1	2	3	4	5	6	7	8
02 FTC		RcvIPtr	RcvOPtr	RXSize	upCnSet	notFInd	TaskID
03 RCV :		00000000	0000002D	FFFFFFFFE	0001A3E5	00000011	00000004
04	notArIP	ARPCnt	IPcnt	SndINPtr	SndOUTtr		
05		00000001	0000001A	00000010	00000010		
06	TotTime	RcvTime	HttpTime	RCV%	HTIP%	CPU%	
07							
08	runTsk	CirCnt	resCnt	delCnt	State		
09 MAIN :	00000004	00000000					
10		RetCode	HttpCnt	TotHTIP	State	Retr	TaskID
11 HTTP :		00000000	00000000		00000010		00001390
12	MaxNTasks	MaxNTcbs	TraceCnt	DelCount	NoOfRsts	UnMatReq	taskDelG
13	00000001	00000007	00000000	00000001	00000000	00000000	
14	delegCnt	delegPCnt	OwnCnt	Cod45cnt			
15		00000000	00000001				
16	RCV%	HTIP%	CPU%	MaxNTasks	MaxNTCBs	NoOfRsts	UnMatReq
17	00000000	00000000	00000000	00000001	00000007	00000000	00000000
18	delegCnt	delegPCnt	OwnCnt	TotHTIP	Retr	SynCnt	FinCnt
19	00000000	00000000	00000001	00000001	00000000	00000001	00000001
20							
21							
22							
23	0000000A	00000037	0000000C	FFFFFFC8			
24							

Figure 38: Run Application (Option (3))

In multi-core run use the following sequence to load and run:

- (2) load application
- (6) Multi-Core Mode
- (6) Activate Multi-Core

There is a /Fileserver directory where there is a Windows Server program that provides file transfer to a bare machine. In this directory, './run' command will run the file server. Run the file server before you run an application. The /wwwt directory has the files needed to be uploaded to the server. If you need to compile the file server, you need an appropriate

path for the Windows header files. Use the e.bat file to provide the correct path for the system.

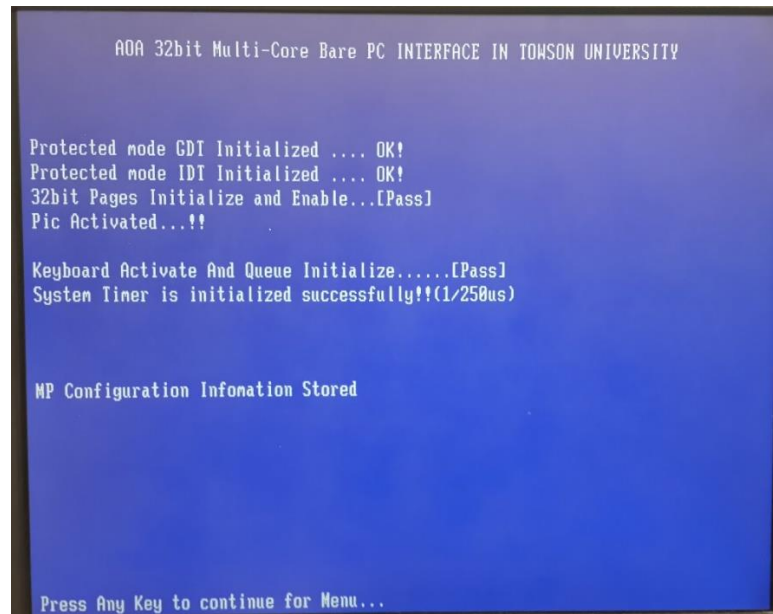


Figure 39: Changing into Multi-Core Mode (Option (6))

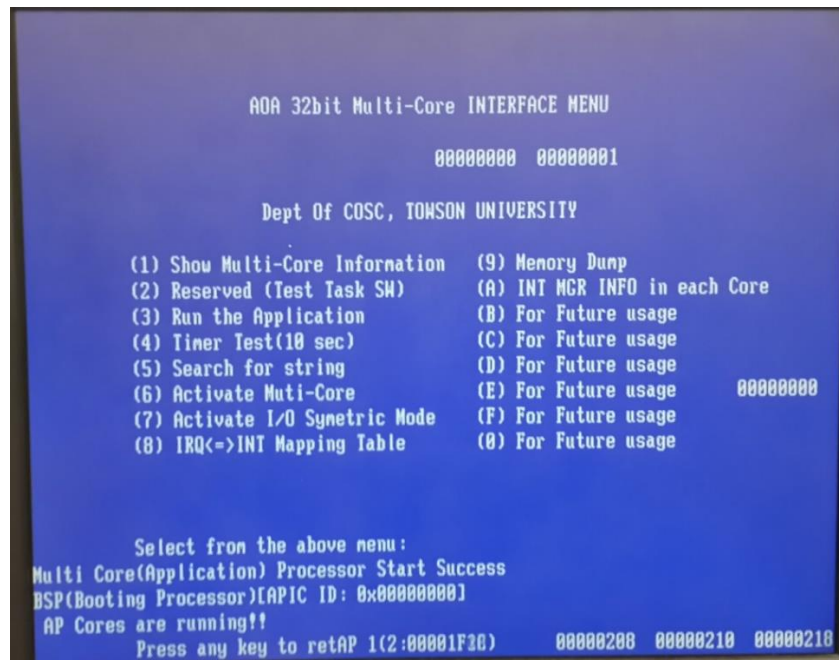


Figure 40: Activate Multi-Core (Option (6))

There is also an e.bat file in the main directory that provides correct paths for the compilation environment. If the directory structure is changed, then you need to change this e.bat file accordingly.

## REFERENCES

- [1] U. Okafor, R. K. Karne, A. L. Wijesinha, and P. Appiah-Kubi, "Eliminating the Operating System via the Bare Machine Computing Paradigm", The Fifth International Conference on Future Computational Technologies and Applications, Future Computing 2013, Valencia, Spain.
- [2] R. K. Karne, "Application-oriented Object Architecture: A Revolutionary Approach," 6th International Conference, HPC Asia 2002, Centre for Development of Advanced Computing, Bangalore, Karnataka, India, December 2002.
- [3] L. He, R. K. Karne, and A. L. Wijesinha, "Design and Performance of a bare PC Web Server," International Journal of Computer and Applications, vol. 15, pp. 100-112, Acta Press, June 2008.
- [4] G. Ford, R.K. Karne, A.L. Wijesinha, and P. Appiah-Kubi. "The design and implementation of a Bare PC email server," COMPSAC'09, 33rd IEEE International Computer and Applications Conference, pp. 480-485, July 2009.
- [5] A. Alexander, A. L. Wijesinha, and R. Karne. "A Study of Bare PC SIP Server Performance," The Fifth International Conference on Systems and Networks Communications. ICSNC 2010, August 22-27, Nice, France.
- [6] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala, "A Peer-to-Peer Bare PC VoIP Application," CCNC'07, Proceedings of the IEEE Consumer and Communications and networking conference, pp. 803-807, IEEE Press, Las Vegas, Nevada, January 2007.
- [7] "Tiny OS," Tiny OS Open Technology Alliance, University of California, Berkeley, CA, 2004, I <http://www.tinyos.net/>.
- [8] J. Lange et al., "Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing", 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010.
- [9] D. Engler, "The Exokernel Operating System Architecture." Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, October 1998.
- [10] J. Schneider, M. Bohn, and R. Robger, "Migration of real-time automotive software to multicore systems: first steps towards an automated solution", 22nd Euromicro Conference on Real-Time Systems (ECRTS), 2010.
- [11] F. Nemati, J. Kraft, and T. Nolte, "Towards migrating legacy real-time systems to multi-core systems", IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2008, pp. 717-720.
- [12] T. G. Brutch, "Migration to multicore: tools that can help", Login: The USENIX Magazine, Vol. 34, No. 5, October 2009.
- [13] R. K. Karne, K.V. Jaganathan, T. Ahmed, and N. Rosa, "DOSC: Dispersed Operating System Computing," 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Onward Track, pp. 55-61, 2005.



- [14] G. Ford, R.K. Karne, A.L. Wijesinha, and P. Appiah-Kubi. "The performance of a Bare Machine email server," SBAC-PAD'09, 21st International Symposium on Computing Architecture and High Performance Computing, pp. 143-150, IEEE, October 2009.
- [15] Ford, G.H., Karne, R.K., Wijesinha, A.L., and Appiah-Kubi, P. "The Performance of a Bare Machine Email Server," 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2009), IEEE / ACM Publications, 28-31 October 2009, So Paulo, SP, Brazil, pp. 143-150
- [16] A. Emdadi, R. K. Karne, and A. L. Wijesinha. "Implementing the TLS Protocol on a Bare PC," ICCRD2010, The 2nd International Conference on Computer Research and Development, Kaula Lumpur, Malaysia, May 2010.
- [17] R. Yasinovskyy, A. L. Wijesinha, R. K Karne and G. Khaksari. "Comparison of VoIP Performance on IPv6 and IPv4 Networks", The 7th ACS/IEEE International Conference on Computer Systems and Applications I(AICCSA), 2009
- [18] B. Rawal, R. Karne, and A. L. Wijesinha. "Splitting HTTP Requests on Two Servers," The Third International Conference on Communication Systems and Networks: COMPSNETS 2011, January 2011, Bangalore, India
- [19] B. Rawal, R. Karne, and A. L. Wijesinha. "Mini Web Server Clusters for HTTP Request Split", 13<sup>th</sup> International Conference on High Performance Computing and Communication, HPCC-2011, Banff, Canada, I Sept 2-4, 2011.
- [20] P. Appiah-Kubi, A. L. Wijesinha, and R. K. Karne. "The Design and Performance of a Bare PC Webmail Server," 12th IEEE International Conference on High Performance Computing and Communications (AHPCN), pp. 521-526, 2010.
- [21] U. Okafor, R. K. Karne, A. L. Wijesinha, and P. Appiah-Kubi, "A Methodology to Transform an OS-based Application to a Bare Machine Application, The 12th IEEE International Conference on Ubiquitous Computing and Communications (ICC-2013), Melbourne, Australia, 16-18 July, 2013.
- [22] A. Peter, R. Karne, A. Wijesinha, and P. Appiah-Kubi, Transforming a Bare PC Application to Run on an ARM Device, IEEE SoutheastCon, April 4-7, Jacksonville, FL, 2013.
- [23] A. Peter, R. Karne, and A. Wijesinha, A Bare Machine Sensor Application for an ARM Processor, IEEE International Electro-Information Technology Conference (EIT), May 9-11, Rapid City, South Dakota, 2013.
- [24] S. Liang, R. K. Karne, and A. L. Wijesinha, "A Lean USB File System for Bare Machine Applications", Proceedings of the 21st International Conference on Software Engineering and Data Engineering, ISCA, pp. 191-196, June 2012
- [25] W. V. Thompson, H. Alabsi, R. K. Karne, S. Linag, A.L. Wijesinha, R. Almajed, and H. Chang, A Mass Storage System for Bare PC Applications Using USBs, International Journal on Advances in Internet Technology, vol 9, no 3 and 4, year 2016. p63-74.

- [26] W. Thompson, R. Karne, A. Wijesinha, H. Alabsi, and H. Chang, Implementing a USB File System for Bare PC Applications, ICIW 2016: The Eleventh International Conference on Internet and Web Applications and Services, p58-63.
- [27] G. Ammons, J. Appayoo, M. Butrico, D. Silva, D. Grove, K. Kawachiva, O. Krieger, B. Rosenberg, E. Hensbergen, R.W. Isniewski, "Libra: A Library Operating System for a JVM in a Virtualized Execution Environment," VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments, June 2007.
- [28] B. Chen, Thesis: "Multiprocessing with the Exokernel Operating System." Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 2000
- [29] C. Coffing, "An x86 Protected Mode Virtual Machine Monitor for the MIT Exokernel." Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1999.
- [30] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullman, "Interface and execution models in the Fluke Kernel," Proceedings of the Third Symposium on Operating Systems Design and Implementation, USENIX Technical Program, New Orleans, LA, February 1999, pp. 101-115.
- [31] B. Ford, and R. Cox, Vx32: "Lightweight User-level Sandboxing on the x86", USENIX Annual Technical Conference, USENIX, Boston, MA, June 2008.
- [32] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt and T. Pinckney. "Fast and flexible application-level networking on exokernel system," ACM Transactions on Computer Systems (TOCS), Volume 20, Issue 1, pp. 49-83, February 2002
- [33] R. K. Karne, S. Liang, A. L. Wijesinha, and P. Appiah-Kubi, "A Bare PC Mass Storage USB Driver", International Journal of Computer and Applications (accepted for publication, March 2013).
- [34] <http://news.bbc.co.uk/2/hi/technology/5107642.stm>. "30 million PCs being dumped each year in the US alone," BBC News: PC users
- [35] V. S. Pai, P. Druschel, and Zwaenepoel. "IO-Lite: A Unified I/O Buffering and Caching System," ACM Transactions on Computer Systems, Vol.18 (1), pp. 37-66, ACM, February 2000.
- [36] "Tiny OS," Tiny OS Open Technology Alliance, University of California, Berkeley, CA, 2004, I <http://www.tinyos.net/>.
- [37] The OS Kit Project," School of Computing, University of Utah, Salt Lake, UT, June 2002, <http://www.cs.utah.edu/flux/oskit>.
- [38] T. Venton, M. Miller, R. Kalla, and A. Blanchard, "A Linux-based tool for hardware bring up, Linux development, and manufacturing," IBM Systems Journal, Vol. 44 (2), pp. 319-330, IBM, NY, 2005.
- [39] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," ACM SIGOPS Operating Systems Review, Volume 43, Issue 2, pp. 76-85, April 2009.

- [40] R. Yasinovskyy, A. L. Wijesinha, R. K. Karne and G. Khaksari. "Comparison of VoIP Performance on IPv6 and IPv4 Networks", The 7th ACS/IEEE International Conference on Computer Systems and Applications I(AICCSA), 2009
- [41] Snghun Han, Principles and Architecture of 64-bit Mutli Core OS, Published by HANBIT Media, Inc., Seoul, Korea, 2011.
- [42] Booting, <http://en.wikipedia.org/wiki/Booting>
- [43] R. K. Karne, K. V. Jaganathan, and T. Ahmed, "How to run C++ applications on a bare PC", SNPD 2005, Proceedings of NPD 2005, 6th ACIS International Conference, IEEE, May 2005, pp. 50-55.
- [44] L. He, R. K. Karne, and A. L. Wijesinha, "The design and performance of a bare PC Web server", International Journal of Computers and Their Applications, IJCA, Vol. 15, No. 2, June 2008, pp. 100-112.
- [45] MultiProcessor Specification Version 1.4, Intel, 1997.  
<http://download.intel.com/design/archives/processors/pro/docs/24201606.pdf>
- [46] Intel® 64 and IA-32 Architectures Software Developers Manual, Intel, 2015.  
<http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html>
- [47] Chang, R. Karne, and A. Wijesinha, Migrating a Bare PC Web Server to a Multi-Core Architecture, 2016 IEEE 40th Annual Computer Software and Applications Conference, p216-221.
- [48] H. Chang, R. Karne and A. Wijesinha, Insight Into the x86-64 Bare PC Application Boot/Load/Run Methodology, The Twenty Second International Conference on Software Engineering and Data Engineering, SEDE 2013, Los Angeles, CA September 25-27.
- [49] Karne, R.K, Wijesinha, A.L, and Ford, G. Opinion-- Stay on course with an evolution or choose a revolution in computing, ACM SIGARCH Computer Architecture News, Volume 36, Number 4, September 2008, p1-6.
- [50] Karne, R.K., Object-oriented Computer Architectures for New Generation of Applications, Computer Architecture News, December 1995, Vol. 23, No. 5, pp. 8-19.
- [51] G. H. Khaksari, A. L. Wijesinha, and R. Karne. Secure VoIP using a Bare PC 3rd International Conference on New Technologies, Mobility and Security (NTMS), 2009.
- [52] W. Agosto-Padilla, R. Karne, and A. Wijesinha, Insights into Transforming a Linux Wireless Device Driver to Run on a Bare Machine, 10th International Conference on Evaluation of Novel Software Approaches to Software Engineering (ENASE 2015).
- [53] A. Loukili, A. Tsetse, A. Wijesinha, R. Karne, and P. Appiah Kubi, Performance of an IPv6 Web Server under Congestion, 12th International Conference on Networks (ICN), 2013.
- [54] A. Loukili, A. L. Wijesinha, R. K. Karne, and A. K. Tsetse, TCP's Retransmission Timer and the Minimum RTO, 21st International Conference on Computer Communications and Networks (ICCCN), 2013.

- [55] A. K. Tsetse, A. L. Wijesinha, R. K. Karne and A. Loukili. A Bare PC NAT Box, International Conference on Communications and Information Technology (ICCIT 2012), June 26-28 2012., Hammamet , Tunisia.
- [56] A. K. Tsetse, A. L. Wijesinha, R. K. Karne and A. Loukili. A 6to4 Gateway with Co-located NAT, IEEE International Conference on Electro Information Technology (EIT 2012), May 6-8 2012 Indianapolis, USA.
- [57] A. K. Tsetse, A. L. Wijesinha, R. K. Karne and A. Loukili. Measuring the IPv4-IPv6 IVI Translation Overhead, ACM Research in Applied Computation Symposium (RACS 2012)- Oct 23-26 2012, San Antonio Texas, USA
- [58] N.Kazemi, A.L. Wijesinha, and R. Karne. Evaluation of IPsec Overhead for VoIP using a Bare PC, 2nd International Conference on Computer Engineering and Technology (ICCET), 2010.
- [59] G. H. Khaksari, R. K. Karne and A. L. Wijesinha. A Bare Machine Application Development Methodology, International Journal of Computers and Their Applications (IJCA), Vol. 19, No.1, March 2012, p10-25.

## CURRICULUM VITAE

NAME: Hojin Chang



PROGRAM OF STUDY: Information Technology

DEGREE AND DATE TO BE CONFERRED: Doctor of Science, August 2017

Secondary education:

Collegiate institutions	attended Dates	Degree	Date of Degree
-----			
Towson University	Aug 2011	D.Sc.	Aug 2017
Soongsil University	Mar 2005	M.S.	Feb 2007

Major: Information Technology for doctoral study and Computer Science for MS study

Professional publications:

1. W. Thompson, H. Chang, R. Karne and A. Wijesinha, A.L., "Interoperable SQLite for a Bare PC", The Thirteenth International Conference, Springer BDAS 2017, Ustron, Poland, p. 177-188.
2. H. Chang, R. Karne and A. Wijesinha, A.L., "Migrating a Bare-PC Webserver to Multi-Core Architecture", IEEE COMPSAC 2016. Atlanta, GA, p. 216-221.
3. W. Thompson, H. Alabsi, R. Karne, S. Liang, A. Wijesinha, R. Almajed, H. Chang, "A Mass Storage System for Bare PC Applications Using USBs", International Journal on Advanced in Internet Technology vol. 9 no 3&4, year 2016, p. 63-74
4. W. Thompson, R. Karne., S. Liang, A. Wijesinha, H. Alabsi, and H. Chang, "Implementing a USB File System for Bare PC Applications", The Eleventh International Conference on Internet and Web Applications and Services, May 22-26, Valencia, Spain, ICIW, 2016, p. 58-63.
5. H. Chang, R. Karne and A. Wijesinha, "Insight Into the x86-64 Bare PC Application Boot/Load/Run Methodology", The Twenty Second International Conference on Software Engineering and Data Engineering, SEDE 2013, p. 25-27.

Professional Certificates: CISSP

