Developing Computer Applications without any OS or Kernel in a Multi-core Architecture

Navid Ordouie
Department of Computer & Information Sciences
Towson University
Towson, USA
nordouie@towson.edu

Nirmala Soundararajan
Department of Computer & Information Sciences
Towson University
Towson, USA
nsound1@students.towson.edu

Ramesh Karne
Department of Computer & Information Sciences
Towson University
Towson, USA
rkarne@towson.edu

Alexander L. Wijesinha
Department of Computer & Information Sciences
Towson University
Towson, USA
awijesinha@towson.edu

***Abstract*—Over the years, operating systems (OSs) have grown significantly in complexity and size providing attackers with more avenues to compromise their security. By eliminating the OS, it becomes possible to develop general-purpose non-embedded applications that are free of typical OS-related vulnerabilities. Such applications are simpler and smaller in size, making it easier secure the application code. Bare machine computing (BMC) applications run on ordinary desktops and laptops without the support of any operating system or centralized kernel. Many BMC applications have been developed previously for single-core systems. We show how to build BMC applications for multicore systems by presenting the design and implementation of a novel UDP-based bare machine prototype Web server for a multicore architecture. We also include preliminary experimental results from running the server on the Internet. This work provides a foundation for building secure computer applications that run on multicore systems without the need for intermediary software.**

Keywords—Operating System Security, Bare Machine Computing, Web Server, Multicore, UDP

I. INTRODUCTION

Operating systems (OSs) continue to grow in complexity and size. This makes it increasingly difficult to secure the OS and the applications that depend on it. There are many secure OSs and numerous techniques are used to harden an OS. An alternative approach is to eliminate the OS. It is then easier to analyze the application code for security flaws because of the reduction in system complexity and size. Bare Machine Computing (BMC) is a novel alternative approach to conventional OS-based computing that enables general-purpose non-embedded applications to run on an ordinary desktop or laptop without the support of any OS or kernel. Previously, a variety of BMC applications were built that run on single core systems. In this paper, we show how to build BMC applications for a multicore architecture by presenting details of the design, and implementation of a new UDP-based BMC multicore Web server. While this server is a prototype and the results presented are preliminary, our work shows the feasibility of developing BMC applications for a multicore architecture that will be easier to secure because of reduced system complexity.

BMC applications are currently designed for Intel x86-based CPUs. The entire BMC application has no dependence on any intermediary or external software. USB mass storage (a flash drive) is used as an external detachable medium to store all the code and data for the server and likewise for the client. The BMC application integrates the application code with the necessary network and security protocols, and device drivers. For example, a bare driver that works with an Intel Gigabit Ethernet controller is integrated with the server code.

The focus of this paper is on providing developers with the details needed to build a BMC application for a multicore system. As such, we do not undertake a security analysis of the code or demonstrate that this BMC application can withstand typical attacks that target the underlying OS or kernel. Nevertheless, we believe that the design and implementation details provided here can be used in the future for building secure OS-independent applications that run on multicore systems. The rest of the paper is organized as follows. Section II describes related work. Section III describes the system architecture, gives a step-by-step approach for bare machine multicore application design, and provides key implementation details of the server. Section IV presents preliminary Internet measurements using the server and a bare client. Section V contains the conclusion.

II. RELATED WORK

OSs have grown in size and complexity over the years, and a variety of techniques have been devised to move some of the OS functions to applications and reduce the OS footprint. Such techniques are used in Exokernel [1], Microkernel [2], Tiny-OS [3], IO_Lite [4], Palacio and Kitten [5], Bare Metal Linux [6], OS-Kit [7] and Raw Architecture [8]. BMC systems are similar to existing minimal systems except that applications run with no OS or kernel support. It is possible to write a simple operating system from scratch as in [9], and to build one's own OS with small image sizes as in [10]. However, such systems typically use OS concepts to build rapid prototypes that are compatible with existing evolutionary applications. Pushing parts of kernel code to applications has many benefits. For example, UDP is used for HTTP clients and servers enabling Web applications to avoid the complexity and overhead of TCP. CoAP for IoT devices [11] and the QUIC protocol [12], support UDP-based Web applications. QUIC provides similar services to those offered by TCP and TLS, and serves as the basis for HTTP/3 [13]. HTTP over UDP for a multicore BMC server in this paper is a customized minimal application protocol design that is considerably simpler than QUIC and not dependent on any OS. This will make it easier to analyze the code and make it secure.

The BMC HTTP over UDP protocol likewise has no relation to CoAP although it can also be adapted for use with a multicore proxy serving IoT devices.

A variety of BMC applications have been built and tested in real environments including local LANs and the Internet. They include Web servers [14], text-only browsers [15], mail servers [16], SIP servers [17], VoIP clients [18], IPv6-IPv4 translators [19], and SQLite database engines [20]. A TCP-based BMC multicore Web server is described in [21], and design issues relevant to such servers are discussed in [22]. The present UDP-based multicore BMC Web server differs from a UDP-based BMC Web server [23] in three important ways. Firstly, that server was designed only for a single core. Secondly, optional ACKS and TIME_OUTS were not used as in the present server to handle UDP packet loss on the Internet. Thirdly, we used the data header format shown in Fig. 3 that has new fields to simplify the bare client and improve reliability. The new UDP-based application multicore design has the benefits of being simpler, more general, and efficient. It can thus be more easily used to develop and secure general-purpose (non-embedded) BMC multicore applications in the future.

III. Architecture and Design

The design of a BMC application requires knowledge of the underlying system. A step by step bottom-up approach is used to design, develop, and test such applications. The following sub-sections give details of the design methodology and include code snippets to illustrate the simplicity and novelty of this approach. These snippets provide basic programming nuggets to construct other bare PC applications or suites.

When there is no OS or kernel running in the machine, the application program becomes event driven. This requires prior knowledge of the execution of program threads as the program might have to wait for required resources. In such cases, the application programmer uses the "Suspend()" mechanism to wait until an event occurs. This avoids the scheduling problem that is required in a centralized OS or kernel approach. The anticipated event will wake up a suspended thread to continue execution. This logic becomes part of the application program and eliminates the need for an OS or kernel.

Process management is avoided by simply creating known processes a priori in the program. For example, a TCP Web server requires a receive task, and many HTTP tasks. A pool of tasks can be created and used at run time as needed. A single main entry in the application will provide execution for tasks based on the relevant scheduling method. In this approach, a TSS (task segment state) required for a given task is also part of the application program, thus avoiding process management by a kernel. Also, there is no user/privileged mode switch. All applications run in privileged mode and they are self-contained, self-executable and self-manageable entities. Similarly, file management and disk management are also avoided as they are part of the application. This does not imply that a general-purpose management system is needed as with a traditional OS. The management functions only focus on the requirements of the given application suite and are tailored to it.

Fig. 1 shows the simple system architecture used in this paper for developing and testing the multicore BMC server and client, and for making preliminary performance measurements. Fig. 2 shows the HTTP over UDP message exchange.

A. Boot Code

The first step is to boot the machine with customized boot code (boot.bin). When the PC is powered on, based on the BIOS configuration, a BIOS interrupt transfers the boot sector from a designated device to the memory location at 0x7c00. The BIOS loads the PC (program counter) with the this address, and then the CPU starts running the boot code, which is usually one sector. Fig. 4 shows the boot process in IBM desktops. The system starts in real mode in Intel processors, implying that the CPU only understands real memory addresses limited to 1 MB or 20-bit addresses. The small boot code is usually written in assembly code, has minimal functionality, and validates a signature at the end to indicate success. Commonly in boot code, a tiny loader loads the startup program.

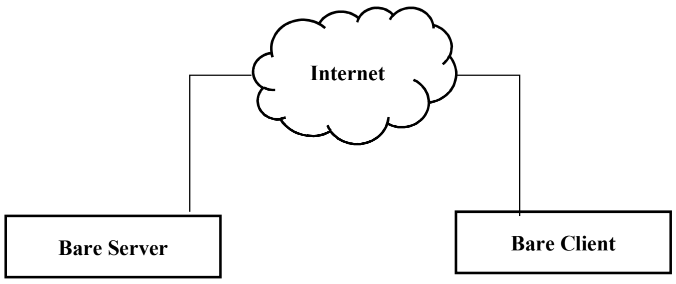


Fig. 1. System Architecture

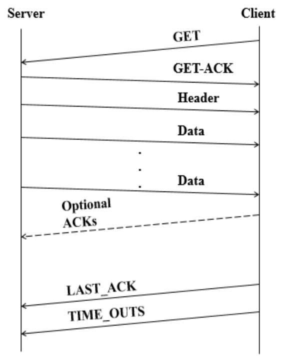


Fig. 2. Client-Server Message Exchange

In real mode, startup code (entry.bin) is loaded from the flash drive to a memory location at 0x2000 by using INT 13H. This is a BIOS interrupt, and it uses a data structure that facilitates loading the entry or start up program (Fig. 5). Once the startup (entry.bin) is loaded, the program performs a jump to go to the startup code. The system is still in real mode and it will never return to the boot program as it performed a jump instruction (hard boot). Alternatively, with a soft boot, it jumps to the boot code at a given point in the program, In many address exception cases, the system usually reboots and executes the boot code.

B. Real-Protected Modes

Once boot code loads the startup program (entry.bin), the startup program runs in real mode. In this case, memory is limited to 1 MB. In order to run large programs, a switch to protected mode is needed to access the full address space and to also protect the memory. In real mode, there is access to BIOS interrupts that are not available while in protected mode. Video memory is available in both modes to display text data. The BIOS interrupt 13H was used to load programs and data from mass storage as shown in the previous section. In the startup code, the actual application code is loaded in memory above 1 M (protected mode area) by using the same BIOS interrupt. This requires the program to read a sector in real mode and store the data read in protected mode. As the file system is not available at this point of execution, INT 13H is the only option to load the application program (test.exe). One sector at a time is loaded to make the code simple for reading and writing to the protected mode area of memory. A data structure similar to the one used in the boot code is also used here for reading sectors from the flash drive. This loader is more sophisticated than the one used in the boot code. Fig. 6 shows a simple control flow of this mechanism. The first time the processor is in real mode, the CR0 is saved and reloaded whenever it goes back to real mode. CR0 plays a major role in switching between real and protected mode.

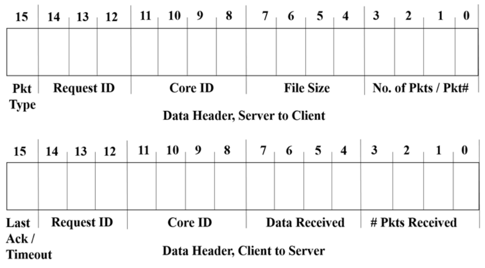


Fig. 3. UDP Data Header

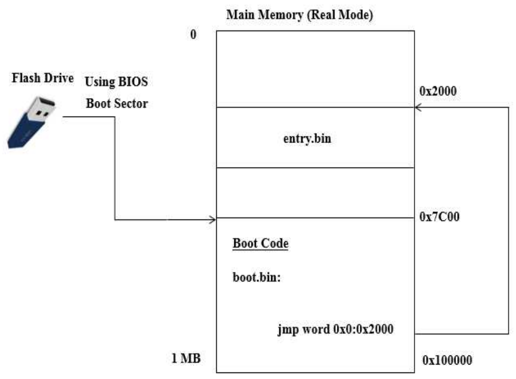


Fig. 4. Boot Process

In order to switch to protected mode, address bit 20 must be enabled (unless it is automatically enabled). Fig. 7 shows the code snippet for enabling address bit 20. The address bit 20 in real mode is the highest bit for addressing and it is disabled. It needs to be enabled in protected mode to address above 1 MB and up to the limit of its address space. There are some key instructions in this code. The instructions CLI and STI are assembly instructions to disable and enable interrupts respectively. While enabling the address bit 20, no interrupts should be processed. Another key element in this code is to balance PUSH and POP instructions, since otherwise the stack will go out of order and cause major issues in the program. In assembly functions, it is customary to push registers that are used in the call and pop those registers back before leaving the function. This will guarantee register contents before the function is called.

```
;-----------------------------------
;INT 13H Data Structure
dapa:
        dw 10h
; Packet size 16 bytes, data structure size
.count:     dw 48
; Block count (no of sectors)
.off:   dw 0x2000
;Offset of buffer (memory location for destination)
.seg:       dw 0000h
; Segment of buffer (seg:offset)
.lbal:      dd 00002009h
;LBA (LSW) sectors
.lbam:      dd 0
; LBA (MSW)
;-----------------------------------
;INT 13H call setup
push    di
mov     di, dapa
mov     si,di
mov     dl,80h
mov     ah,42h
int     13h
;-----------------------------------
```

Fig. 5. INT 13h Data Structure and its Call

It is necessary to keep separate stacks for real and protected modes. Switching to protected mode involves control register 1; details are shown in Fig. 8. The important bits to modify in CR0 are PE (protected mode enable bit to 1) and CD (cache disable to 0). After the PE bit is enabled in CR0, the GDT and IDT registers must be loaded to address memory and process interrupts respectively. In real mode, segment registers are used to address beyond one segment (size of 64K). In protected mode, segment registers are loaded with selectors to address memory. A selector points to an 8-byte descriptor located in the GDT (global descriptor table). Descriptors are simply indexed from the beginning of the GDT in memory. For a given application program, code selector, data selector and stack selector have to be defined to address the memory. Each descriptor provides a start address and a limit for a memory region. The hardware checks for these addressing limits for each memory access. The control flow shown in Fig. 6 accomplishes loading of an application suite into the protected memory area. All application code, data and stack are situated in the protected mode area of memory. Once this loading process is done, issues related to processes and multiple cores are handled.

C. Process Setup

GDT and IDT Setup: In 32-bit mode, in order to create a separate process, a GDT (Global Descriptor Table) entry and a TSS (Task Segment State) table are needed. The GDT entry is an index in the GDT table, and it is used as a selector for addressing and validating memory. These selectors are loaded into the segment registers as needed to address memory. In the multicore architecture, the boot processor is designated as BSP (Boot Strap Processor) and other processors are referred to as APs (Application Processors). Fig. 9 shows three sample selectors for code, data and stack for the BSP process, and other important details of the 8-byte descriptor, such as the 8 bytes illustrating the “appcodedesc.” In this model, a 4GB limit and base address of 0x00110000 is assumed. In addition, it uses system privilege level (DPL 00 – lowest ring), granularity of 4K segments (making it to 4GB address space) and 32-bit segments. More details of the descriptors are in [23]. The descriptors must be defined appropriately as shown in the figure to guarantee correct operation of the system. The descriptor fields provided here are the actual GDT descriptors used in the prototypes. Other descriptors for AP1, AP2, and AP3 are similarly defined. Notice that the selector base address is at 0x00110000, forcing the program to be loaded above 1MB. These selectors cannot be used to address real mode memory below 1MB. In order address the entire address space, there is also need for a ZERO selector, which can address the entire memory. In this selector, the base address is simply all zeros.

Usually, in 32-bit mode, tasks (processes) are created using interrupt gates. In addition, there are trap gates and interrupts that use them. Each type of gate plays its own role in their usage. Fig. 10 shows the format for these three possible descriptors. Mostly, 0 – 31 are used for hardware exceptions such as divide by zero. Interrupts 0-31 use trap gates as these are hardware exceptions. In the Web server prototype, only the timer interrupt and keyboard interrupts are used and they are mapped to interrupts 32 and 33 respectively. These interrupts are hardware interrupts that use interrupt gates. The BSP, AP1, AP2 and AP3 in this system use task gate interrupts. Fig. 11 shows key details used in the design and the actual descriptors used for each type of gate. Byte 5 in the descriptor defines the type of gate. The TGATE, IGATE and KGATE macros help to define all 256 interrupts in Intel x86. These 256 definitions of interrupt descriptors are placed in the IDT (interrupt descriptor table). Similarly, all GDT descriptors are placed in the GDT table. The IDTR (IDT register) is loaded with the starting point of the IDT table. Similarly, the GDTR register is loaded with the starting point of the GDT table. Trap Gate and Interrupt Gate descriptors can point to the interrupt service routine (usually referred to as ISR) as shown in the figure

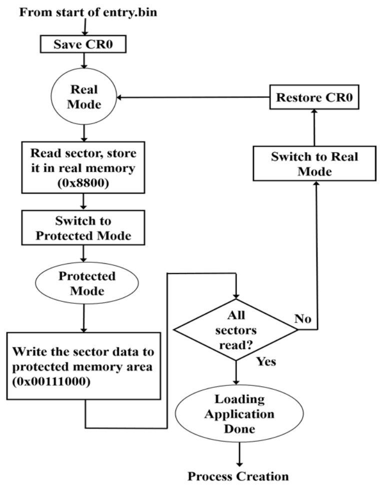


Fig. 6. Real-Protected Mode Cycle

```
enableA20BIOS:
 push bp
 mov  bp, sp
 cli
 push ax
 mov ax, 0x2401 ;enable A20 address line
 int 0x15
 pop ax
 pop bp
 sti
 ret
```

Fig. 7. Enable Address Bit 20

```
mov eax, 0x4000003B
;PG=0, CD=0, NW=0, AM=0, WP=0, NE=1, ET=1, TS=1,
EM=0, MP=1, PE=1
mov cr0, eax
cli
lgdt[gdtreg]
lidt[idtreg]
jmp dword CODE_SEL:PM_Start
;JUMP to protected mode starting point
```

Fig. 8. Switch to Protected Mode

```
TGATE DEException, INTEXCEPTION_SEL, 0x8f, 0    ; 0    Trap Gate
                                                ; 1 00    0  1111
                                                ; P DPL S Type
IGATE  OtherExc32,   INTEXCEPTION_SEL, 0x8e, 0  ; 32  Interrupt Gate
                                                ; 1 00    0  1110
                                                ; P  DPL S Type
KGATE TSS0_SEL, 0x85  ;P=1,DPL==00, S=0, D=0    ; 235 Task Gate  BSP

TSS0_SEL      EQU     (tss0desc - nulldesc)     ; BSP
tss0desc:
  DW TSS0_LEN - 1
  DW tss0start
  DB 0
  DB 0x89  ;1000 1001  G DPL S  ( G=1, DPL=00, S=0 system, Type is execute only)
  DB 0x00
  DB 0x00

%macro TGATE 4
   DW %1 ;offsetL
   DW %2 ;Selector
   DB 0x00
   DB %3 ;Gtype
   DW %4 ;OffsetH
%endmacro

%macro IGATE 4
   DW %1 ;offsetL
   DW %2 ;Selector
   DB 0x00
   DB %3 ;Gtype
   DW %4 ;OffsetH
%endmacro

%macro KGATE 2
   DW 0x0000 ;offsetL
   DW %1     ;Selector
   DB 0x00
   DB %2     ;Gtype
   DW 0x0000 ;offsetH
%endmacro
```

Fig. 9. GDT descriptors

TSS Setup: A task gate is used to create a process or a task. The interrupts INT 232, INT233, INT234, INT235 are used for creating tasks for AP1, AP2, AP3 and BSP respectively. Fig. 11 shows the BSP task gate as an example. There is no ISR address in this descriptor. Task gates use the TSS (task segment state) structure in Intel 32-bit multicore. When this interrupt is called, its associated hardware stores the state of the machine in the current TSS and then jumps to the TSS that it is supposed to run. During this process, the link field in the new TSS is modified to point to the previous TSS. This mechanism provides a means to return to the previous task when needed. Thus, the TSS is the key data structure for task management in Intel 32-bit. Fig. 12 shows a sample template for the TSS. General purpose registers, program counter, stack pointers and all CPU-related information is stored in the TSS, so that each task can fill in its own state in the CPU. This template is used to create other TSSs by simply filling in appropriate fields for a given task.

Process Invocation: Four processes BSP, AP1, AP2 and AP3 are created using the following methodology. Processes for the APs are created, but they will not run until the BSP enables them to run. As there is no task concept in the entry.asm code up to this point, a dummy TSS is created for each core. A dummy TSS for the BSP is shown in Fig. 13, where the appropriate values are filled in with the entry.asm selectors and related pointers. Similarly, other three dummy TSSs are created for

AP1, AP2 and AP3. When a task switch occurs, the dummy TSS will be filled in with the current state of the machine and a move to the next process occurs. The BSP task invocation steps are shown in Fig. 14. These steps are crucial to the proper invocation of a BSP task.

In the multicore architecture, there is a different mechanism for interrupt handling (not discussed here). The task register is loaded with the Demo selector, which is a demo TSS used to start the BSP processor. Peripheral interface controllers must be initialized in the multicore architecture. Timer and keyboard interrupts are ignored until the BSP starts running in a new task. Fig. 15 shows a part of the assembly code for this initialization. Using the FillTSS0 function (Fig. 16), its TSS is filled in with the appropriate data. The EIP or program counter, is initialized to 0x1000. The base address 0x00111000, 0x00110000 of the test.exe was in the descriptor. For Microsoft executables, the starting address is usually at 0x1000. The timer is set up to run at ¼ millisecond speed using the set timer function (Fig. 17). Calling INT 235 invokes the task gate and makes the appropriate task switching to run the BSP code (test.exe), starting at address 0x00111000 where test.exe was loaded. All cores run concurrently following the boot process after BSP activates them. In the entry.asm code, a shared memory flag (at location 0x8750) is stored by the BSP to enable APs to run after they have been reactivated. Here we assume that only the BSP is running (the APs are not running). Process creation for APs is described later.

D. Multicore Setup

The BSP process starts running at main() in test.exe. Fig. 18 shows the main control flow logic after it starts. All cores go through entry.bin as well as the test.exe code. However, when they enter the test.exe flow, they will take different routes as shown in the figure. When they are in the entry.bin code, they also take different paths based on whether the BSP has set a shared memory flag or not. The BSP only sets the shared memory flag just before they are activated. After the BSP flow is complete, it goes to its MainTask() function and assumes its server role. The BIOS constructs the MP Configuration Table where each processor is identified by a unique local APIC ID (Advanced Programmable Interrupt Controller). The APs are all in sleep or halted mode and passively monitor the APIC bus. They react only after special interrupts like INIT IPI and STARTUP IPI are issued. These are special Inter Processor Interrupts (IPI). Figs. 19 and 20 respectively show details concerning AP activation and AP flow control. Upon receiving an INIT IPI, the local APIC causes an INIT at its processor. STARTUP IPI enables the processor to start executing in real mode from address 000VV000h, where VV is an 8-bit vector that is part of the IPI message. The Interrupt Command Register (ICR) is used to send IPIs between the processors, and writing to the ICR causes the IPI message to be sent.

E. Client-Server Systems

The client design is similar to the server design, except that it is inverted to communicate with the server. In this server prototype, the BSP is running the Web server system and the other APs are simply running in an idle loop while waiting to share the workload from the BSP. Load balancing strategies are not considered here. The client and server systems are implemented in C/C++ supplemented by a small amount of assembly code. The server executable (test.exe) is 549 sectors and the client executable (test.exe) is 230 sectors. Currently, the client is a text-only system used to stress test the server.

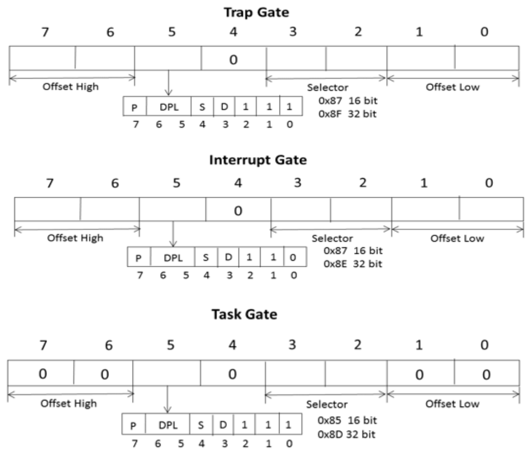


Fig. 10. IDT Descriptor Formats

```
TGATE DEException, INTEXCEPTION_SEL, 0x8f, 0    ; 0    Trap Gate
                                                ; 1 00    0  1111
                                                ; P DPL S Type
IGATE  OtherExc32,   INTEXCEPTION_SEL, 0x8e, 0  ; 32  Interrupt Gate
                                                ; 1 00    0  1110
                                                ; P  DPL S Type
KGATE TSS0_SEL, 0x85  ;P=1,DPL==00, S=0, D=0    ; 235 Task Gate  BSP

TSS0_SEL      EQU     (tss0desc - nulldesc)     ; BSP
tss0desc:
  DW TSS0_LEN - 1
  DW tss0start
  DB 0
  DB 0x89  ;1000 1001  G DPL S  ( G=1, DPL=00, S=0 system, Type is execute only)
  DB 0x00
  DB 0x00

%macro TGATE 4
   DW %1 ;offsetL
   DW %2 ;Selector
   DB 0x00
   DB %3 ;Gtype
   DW %4 ;OffsetH
%endmacro

%macro IGATE 4
   DW %1 ;offsetL
   DW %2 ;Selector
   DB 0x00
   DB %3 ;Gtype
   DW %4 ;OffsetH
%endmacro

%macro KGATE 2
   DW 0x0000 ;offsetL
   DW %1     ;Selector
   DB 0x00
   DB %2     ;Gtype
   DW 0x0000 ;offsetH
%endmacro
```

Fig. 11. Example IDT Descriptors

IV. Preliminary Internet Measurements

In this section, we present the results from limited performance studies using the prototypes. Recall that UDP packet loss is handled by timeouts and retransmissions. Also, out-of-order packets are handled by storing and reordering them after all packets arrive. UDP does not keep any state in a client request, so the client and server maintain some state to ensure reliable transmission at the application level. The maintenance of state for each request results in large queue sizes. This was avoided by doing frequent timeouts every 4 seconds. Such settings may need to be dynamically adjusted in a real system. We were only able to run up to 3000 requests at the client site on the Internet, whereas earlier studies using a UDP-based single core Web server in a LAN achieved over 13000 requests/per second. It is possible that UDP throttling at the ISP level affected our test results. Some interesting and unexpected findings based on the tests are discussed below.

```
tss0start:
FirstTSS:
istruc TSSSEG
at TRLink,      DW    0      ;LINK
                DW    0
at TRESP0,      DW    0      ;ESP0 Stack 0 Length
                DW    0
at TRSS0,       DW    0      ;SS0 Stack 0 Selector
                DW    0
at TRESP1,      DW    0      ;ESP1 Stack 1 Length
                DW    0
at TRSS1,       DW    0      ;SS1 Stack 1 Selector
                DW    0
at TRESP2,      DW    0      ;ESP2 Stack 2 Length
                DW    0
at TRSS2,       DW    0      ;SS2 Stack 2 Selector
                DW    0
at TRCR3,       DD    0      ;CR3
at TREIP,       DD    0      ;EIP
at TREFlag,     DD    0      ;EFLAGS

; ------ Other GPRs ------

at TRLDTR,          DW  0, 0     ;LDT
at TRTrip,          DW  0        ;Trap Sign
at TRIOMap,         DW  +2       ;I/O Mapping Offset
at ROUNDING,        DB   0,0,0,0,0,0,0, 0ffh
;rounding to 112
iend
TSS0_LEN            EQU     ($ - tss0start)
```

Fig. 12. TSS Template

```
at TRLink,      DW     0    ;LINK
                DW     0
at TRESP0,      DW     DemoStk0_LEN
                       ;ESP0 Stack 0 Length
                DW     0
at TRSS0,       DW     DemoStackd0_Sel
                       ;SS0 Stack 0 Selector
                DW     0
at TRESP1,      DW     DemoStk1_LEN
                       ;ESP1 Stack 1 Length
                DW     DemoStackd1_Sel
at TRSS1,       DW     0
                       ;SS1 Stack 1 Selector
                DW     0
at TRESP2,      DW     0
                       ;ESP2 Stack 2 Length
                DW     0
at TRSS2,       DW     0
                       ;SS2 Stack 2 Selector
                DW     0

; -- The rest of the fields are the same as  Figure 12 --

;rounding to 112
iend
```

Fig. 13. Demo TSS for BSP

```
mov ax, TSSBSPDemo_SEL
LTR ax
;load task register with TSSDemo

call initializePICs
call initializeTimer
call IgnoreKBDInt
call IgnoreTimerInt
call FillTSS0
int 235
```

Fig. 14. BSP Invocation

Experiment I: These test results are derived from Wireshark traces and some code measurements using the system architecture shown in Fig. 1. The bare server and client connected to the Internet via a pair of gigabit Ethernet switches and their respective ISP routers (Wireshark ran on a Windows machine). HTTP requests for a 4K file are sent to the server at 250, 500, 1000, 2000, and 3000 requests per second. Timeouts, outstanding requests, out of order requests and no-matches were measured. Timeouts occur every 4 seconds at the server, if the last ACK does not arrive, and at the client if all data packets do not arrive. These timeouts happen asynchronously and result in the removal of requests from their queues. With the present design, out of order requests are easily detected at the client. The non-matches occur due to timeouts, and in some cases due to out-of-order packets. The results also show outstanding requests in the queue (linear lists are used for queues) that are ready to be processed, which determines the degree of parallelism that can be achieved using multiple cores. Each run is made for 5 minutes on the Internet to collect data. The data collected is from the bare server and client based on probe points in the code.

Fig. 21 shows the timeouts for client and server runs using 250, 500, 1000, 2000 and 3000 requests per second. The timeouts vary dramatically as seen in the figure. It was found that these timeouts are directly related to the traffic volume at the ISP router. The difference between timeouts at the server and at the client are due to packet losses or delays at each end. The timeouts increase dramatically after 2000 requests per second at the client, as the client capacity is limited due to the ISP bottleneck. The no-matches and out of order results in Fig. 22 also indicate that after 2000 requests, there are increased no-matches and out of order data. This is directly correlated to the timeouts in the client and the server. Fig. 23 shows the outstanding requests at the client and server. Notice that they both follow the same pattern. The linear list (queue) sizes at client and server is 30000. The outstanding requests observed during the measurements reached a maximum of 1000, but they always converge to the numbers shown in the plot. This is not a queue size issue. Due to timeouts and queue cleanup, the server and client can handle requests without any queue size problems. The client and server systems were run overnight, and they never reached any saturation point or errors during their operation.

Experiment II: For this test, both client and server are connected to Ethernet hubs since there were no switches with port mirroring capability in the home environments used for testing. In order to measure average processing times for requests, Wireshark traces are used, and average times are calculated for 5 random requests. Each run in the measurement is made for 5 minutes on the Internet. Fig. 24 shows the average processing time of a request. We observed some outliers in the traces, which are not included in the average time. As expected, the hub contributes to more delays and timeouts.

V. Conclusion

We described a step-by-step approach for developing bare machine applications for a multicore architecture. We illustrated the approach by designing and implementing a UDP-based multicore Web server. We included internal code snippets that can be used to build similar systems for file transfer, Web mail and other bare machine applications. The design is based on a novel bottom-up approach. Because code images are small and the application only performs intended functions, bare machine systems have less complexity than their OS-based counterparts. This reduction in complexity coupled with the elimination of OS-related vulnerabilities will enable applications to be built in the future that are easier to secure and harder to attack. The Internet performance measurements indicate that these systems can be used to communicate between bare machine servers and clients with tolerable delays. More complex bare machine applications that run on multicore architectures can be built by extending the prototypes described in this work. Further research is needed to address scalability and security issues with respect to multiple cores and multiple nodes.

REFERENCES.

[1] D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: An operating system architecture for application-level resource management," 1995.

[2] D. Black et al., "Microkernel operating system architecture and Mach," J. Inf. Process., 1991.

[3] P. Levis et al., "TinyOS: An operating system for sensor networks," in Ambient Intelligence, 2005.

[4] V. S. Pai, P. Druschel, and W. Zwaenepoel, "IO-Lite: A unified I/O buffering and caching system," ACM Trans. Comput. Syst., 2000.

[5] J. Lange et al., "Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing," 2010.

[6] T. Venton, M. Miller, R. Kalla, and A. Blanchard, "A Linux-based tool for hardware bring up, Linux development, and manufacturing," IBM Syst. J., 2005.

[7] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers, "The Flux OSKit: A Substrate for Kernel and Language Research," Oper. Syst. Rev., 1997.

[8] M. Taylor, Design decision in the implementation of a raw architecture workstation. 1999.

[9] N. Blundell, Writing a Simple Operating System from Scratch. 2010.

[10] E. Baccelli et al., "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," IEEE Internet Things J., 2018.

[11] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252, 2014.

[12] J. Iyengar and M. Thomson, "QUIC: A UDP-based Multiplexed and Secure Transport," draft-ietf-quic-transport-34. 2021.

[13] M. Bishop, "Hypertext Transfer Protocol Version 3 (HTTP/3)," draft-ietf-quic-http-34. 2021.

[14] B. Rawal, R. K. Karne, and A. L. Wijesinha. "Splitting HTTP Requests on Two Servers," The Third International Conference on Communication Systems and Networks: COMSNETS 2011, January 2011, Bangalore, India.

[15] S.Almutairi, R. K. Karne and A.L. Wijesinha, "A Bare PC Text Based Browser," International Conference on Computing, Networking and Communications (ICNC), Honolulu, Hawaii, February 2019.

[16] P. Appiah-Kubi, R. K. Karne, and A. L. Wijesinha. "A Bare PC TLS Webmail Server," International Conference on Computing, Networking and Communications (ICNC), Maui, Hawaii, January 2012.

[17] R. Yasinovskyy, A. Alexander, A. L. Wijesinha and R. K. Karne. "Bare PC SIP User Agent Implementation and Performance for Secure VoIP," International Journal on Advances in Telecommunications, vol 5 no 3 & 4, 2012.

[18] G. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala. "A Peer-to-Peer Bare PC VoIP Application," IEEE Consumer Communications and Networking Conference, Seamless Consumer Connectivity (CCNC), Los Vegas, Nevada, January 2007.

[19] A. Tsetse, A. Wijesinha, R. Karne, A. Loukili and P. Appiah-Kubi, "An Experimental Evaluation of IP4-IPV6 IVI Translation," Applied Computing Review, March 2013, Vol. 13, No. 1.

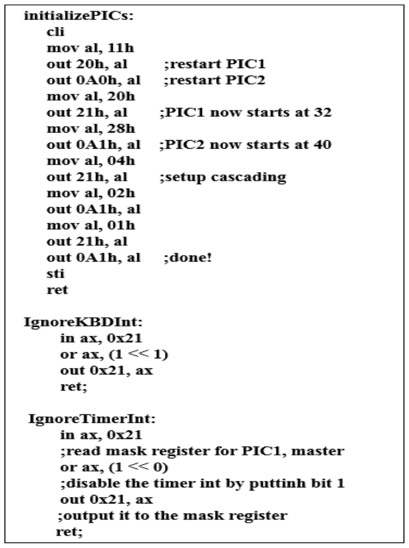
[20] H.Alabsi, R. K. Karne, A. Wijesinha, R. Almajed, B. Rawal, and F. Almansour, "A Novel SQLite-Based Bare PC Email Server," 15th International Conference: Beyond Databases, Architectures and Structures," BDAS, Ustron, Poland, 2019.

[21] H. Chang, R. K. Karne and A.L. Wijesinha, "Migrating a Bare PC Webserver to Multi-core Architecture," IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), Atlanta, Georgia, 2016.

[22] N.Soundararajan, H.Chang, R.K.Karne, A.L.Wijesinha, and N. Ordouie. "Design Issues in Running a Web Server on Bare PC Multi-core Architecture," IEEE 44th Annual Computer Software and Applications Conference (COMPSAC), 2020

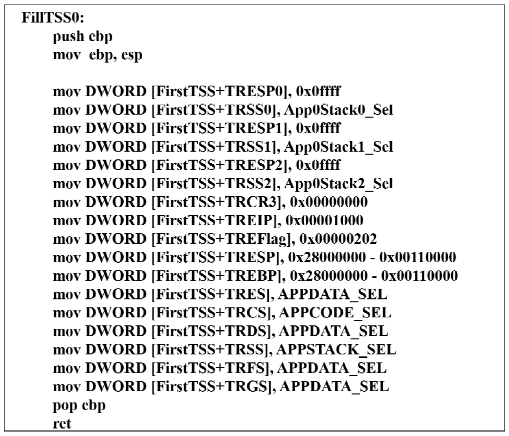
[23] N.Soundararajan, R.K.Karne, A.L.Wijesinha, N.Ordouie, and B.Rawal. "A Novel Client/Server Protocol for Web-based Communication on a Server Architecture based on UDP on a Bare Machine," IEEE Student Conference on Research and Development (SCOReD), 2020.

[24] Intel, "Intel ® 64 and IA-32 Architectures Software Developer's Manual Volume 3A : System Programming Guide, Part 1," System, 2010.



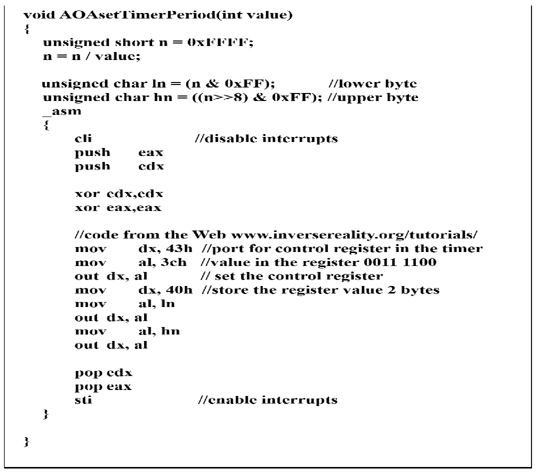
```
initializePICs:
    cli
    mov al, 11h
    out 20h, al      ;restart PIC1
    out 0A0h, al     ;restart PIC2
    mov al, 20h
    out 21h, al      ;PIC1 now starts at 32
    mov al, 28h
    out 0A1h, al     ;PIC2 now starts at 40
    mov al, 04h
    out 21h, al      ;setup cascading
    mov al, 02h
    out 0A1h, al
    mov al, 01h
    out 21h, al
    out 0A1h, al     ;done!
    sti
    ret

IgnoreKBDInt:
      in ax, 0x21
    or ax, (1 << 1)
    out 0x21, ax
      ret;

 IgnoreTimerInt:
      in ax, 0x21
      ;read mask register for PIC1, master
      or ax, (1 << 0)
      ;disable the timer int by puttinh bit 1
      out 0x21, ax
     ;output it to the mask register
     ret;
```

Fig. 15. Initialization in the ASM Code



```
FillTSS0:
      push ebp
      mov  ebp, esp

      mov DWORD [FirstTSS+TRESP0], 0x0ffff
      mov DWORD [FirstTSS+TRSS0], App0Stack0_Sel
      mov DWORD [FirstTSS+TRESP1], 0x0ffff
      mov DWORD [FirstTSS+TRSS1], App0Stack1_Sel
      mov DWORD [FirstTSS+TRESP2], 0x0ffff
      mov DWORD [FirstTSS+TRSS2], App0Stack2_Sel
      mov DWORD [FirstTSS+TRCR3], 0x00000000
      mov DWORD [FirstTSS+TREIP], 0x00001000
      mov DWORD [FirstTSS+TREFlag], 0x00000202
      mov DWORD [FirstTSS+TRESP], 0x28000000 - 0x00110000
      mov DWORD [FirstTSS+TREBP], 0x28000000 - 0x00110000
      mov DWORD [FirstTSS+TRES], APPDATA_SEL
      mov DWORD [FirstTSS+TRCS], APPCODE_SEL
      mov DWORD [FirstTSS+TRDS], APPDATA_SEL
      mov DWORD [FirstTSS+TRSS], APPSTACK_SEL
      mov DWORD [FirstTSS+TRFS], APPDATA_SEL
      mov DWORD [FirstTSS+TRGS], APPDATA_SEL
      pop ebp
      ret
```

Fig. 16. Fill TSS Function for the BSP Processor



```
void AOAsetTimerPeriod(int value)
{
   unsigned short n = 0xFFFF;
   n = n / value;

   unsigned char ln = (n & 0xFF);          //lower byte
   unsigned char hn = ((n>>8) & 0xFF); //upper byte
   _asm
   {
        cli                     //disable interrupts
        push     eax
        push     edx

        xor  edx,edx
        xor  eax,eax

        //code from the Web www.inversereality.org/tutorials/
        mov      dx, 43h  //port for control register in the timer
        mov      al, 3ch  //value in the register 0011 1100
        out  dx, al       // set the control register
        mov      dx, 40h  //store the register value 2 bytes
        mov      al, ln
        out  dx, al
        mov      al, hn
        out  dx, al

        pop edx
        pop eax
        sti                     //enable interrupts
   }

}
```

Fig. 17. Set Timer

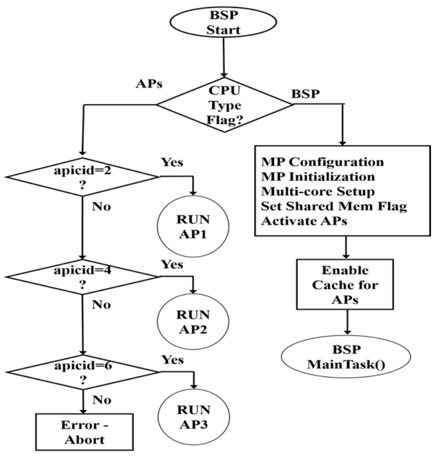
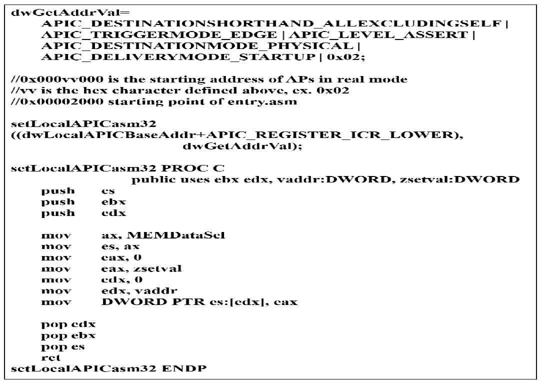
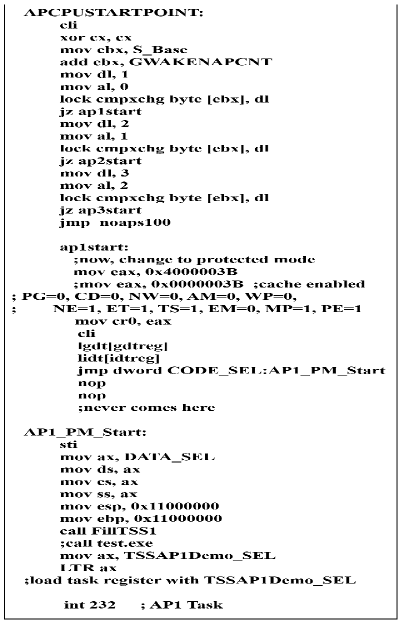


Fig. 18. Main Control Flow



```
dwGetAddrVal=
      APIC_DESTINATIONSHORTHAND_ALLEXCLUDINGSELF |
      APIC_TRIGGERMODE_EDGE | APIC_LEVEL_ASSERT |
      APIC_DESTINATIONMODE_PHYSICAL |
      APIC_DELIVERYMODE_STARTUP | 0x02;

//0x000vv000 is the starting address of APs in real mode
//vv is the hex character defined above, ex. 0x02
//0x00002000 starting point of entry.asm

setLocalAPICasm32
((dwLocalAPICBaseAddr+APIC_REGISTER_ICR_LOWER),
                                dwGetAddrVal);

setLocalAPICasm32 PROC C
                        public uses ebx edx, vaddr:DWORD, zsetval:DWORD
      push      es
      push      ebx
      push      edx

      mov       ax, MEMDataSel
      mov       es, ax
      mov       eax, 0
      mov       eax, zsetval
      mov       edx, 0
      mov       edx, vaddr
      mov       DWORD PTR es:[edx], eax

      pop edx
      pop ebx
      pop es
      ret
setLocalAPICasm32 ENDP
```

Fig. 19. Activating APs



```
   APCPUSTARTPOINT:
          cli
          xor cx, cx
          mov ebx, S_Base
          add ebx, GWAKENAPCNT
          mov dl, 1
          mov al, 0
          lock cmpxchg byte [ebx], dl
          jz ap1start
          mov dl, 2
          mov al, 1
          lock cmpxchg byte [ebx], dl
          jz ap2start
          mov dl, 3
          mov al, 2
          lock cmpxchg byte [ebx], dl
          jz ap3start
          jmp  noaps100

          ap1start:
             ;now, change to protected mode
             mov eax, 0x4000003B
             ;mov eax, 0x0000003B  ;cache enabled
; PG=0, CD=0, NW=0, AM=0, WP=0,
;       NE=1, ET=1, TS=1, EM=0, MP=1, PE=1
             mov cr0, eax
              cli
              lgdt[gdtreg]
              lidt[idtreg]
              jmp dword CODE_SEL:AP1_PM_Start
              nop
              nop
              ;never comes here

   AP1_PM_Start:
          sti
          mov ax, DATA_SEL
          mov ds, ax
          mov es, ax
          mov ss, ax
          mov esp, 0x11000000
          mov ebp, 0x11000000
          call FillTSS1
          ;call test.exe
          mov ax, TSSAP1Demo_SEL
          LTR ax
   ;load task register with TSSAP1Demo_SEL

           int 232     ; AP1 Task
```

Fig. 20. APs Flow Control (Starting Processes)

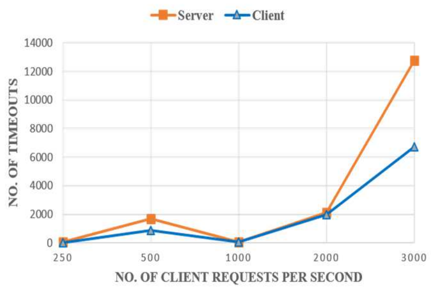


Fig. 21. Timeouts

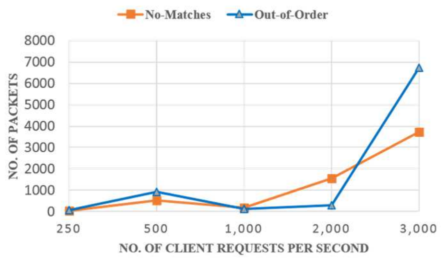


Fig. 22. No Matches and Out-of-Order Packets

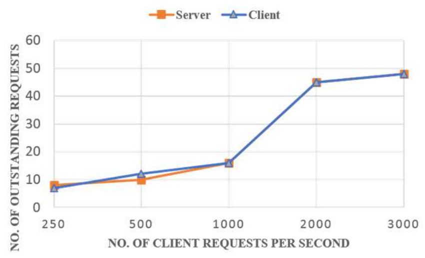


Fig. 23. Outstanding Requests

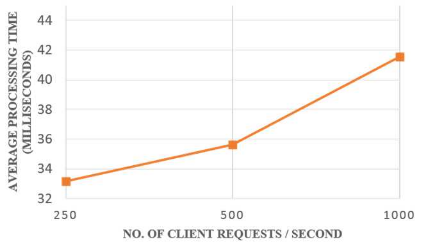


Fig. 24. Average Processing Time