**CS 344 Operating Systems Laboratory Assignment 1**

**TEAM MEMBERS:**

|  |  |
| --- | --- |
| **NAME** | **ROLL NUMBER** |
| RITWIK GANGULY | 180101067 |
| PULKIT CHANGOIWALA | 180101093 |
| SAMAY VARSHNEY | 180101097 |
| UDANDARAO SAI SANDEEP | 180123063 |

**EXERCISE 1:**

Modified Program **ex1.c** which now includes inline assembly that increments the value of x by 1.

#include<stdio.h>

int main(int argc, char \*\*argv)

{

int x=1;

printf(“Hello x = %d\n”, x);

**asm("incl %0": "+r"(x));**

/\*

First method to increment the value of x using inline assembly language having one line

asm("incl %0": "+r"(x));

1. incl command adds 1 to the 32-bit contents of the variable specified
2. %0 refers to the first variable passed (which in this case is x)
3. the ‘+’ sign before r denotes that x acts as both input and output

\*/

/\*

Second method to increment the value of x using inline assembly language having multiple lines (uncomment to use this)

**asm(“mov %1, %0 \n\t”**

**“add $1, %0”**

**: “=r” (x)**

**: “r” (x));**

\*/

printf(“Hello x = %d after increment\n”, x);

if(x == 2) {

printf(“OK\n”);

}

else {

printf(“ERROR\n”);

}

}

**EXERCISE 2:**

We are trying to explain the instructions to guess what BIOS might be trying to do:

**1st instruction**: [f000:fff0] 0xffff0: ljmp $0x3630,$0xf000e05b

* Jump to CS = $0xf000 & IP = 0xe05b
* 0x3630 is jump to this CS (earlier in the BIOS)
* 0xf000e05b is the IP which is different from the lab because it is 32 bits rather than 16 bits and that is all the way into the top of the extended memory location but before the memory mapped PCI device location reserved by the BIOS

**2nd Instruction**: [f000:e05b] 0xfe05b: cmpw $0xffc8,%cs:(%esi)

* Compare content at 0xffc8 & with content at code segment offset with value at esi.
* esi:- 32-bit source index register

**3rd Instruction**: [f000:e062] 0xfe062: jne 0xd241d0b2

* Jump to 0xd241d0b2 if the above comparison does not set ZF

**4th instruction**: [f000:e066] 0xfe066: xor %edx,%edx

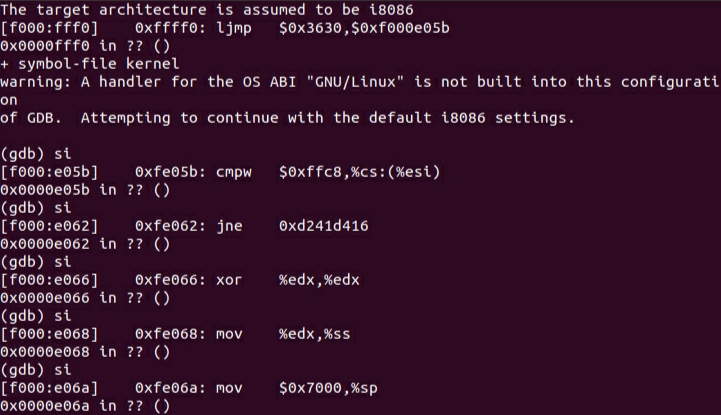
* ZF was set thus jump of previous instruction doesn’t occur
* It set edx to zero, edx is 32-bit general-purpose register.

**5th instruction**: [f000:e068] 0xfe068: mov %edx,%ss

* Move content of stack segment register(ss) to edx

**6th instruction**: [f000:e06a] 0xfe06a: mov $0x7000,%sp

* Move content at the location pointed 16-bit stack pointer(sp) to $0x7000



**EXERCISE 3:**

**Comparison of code at 0x7C00 memory location using first few instructions between original boot loader source code (bootasm.S) and GDB and bootblock.asm :**

**In Bootasm.S:**

.code16 # Assemble for 16-bit mode

.globl start

start:

cli # BIOS enabled interrupts; disable

xorw %ax,%ax # Set %ax to zero

movw %ax,%ds # -> Data Segment

movw %ax,%es # -> Extra Segment

movw %ax,%ss # -> Stack Segment

seta20.1:

inb $0x64,%al # Wait for not busy

testb $0x2,%al

jnz seta20.1

movb $0xd1,%al # 0xd1 -> port 0x64

outb %al,$0x64

**In Bootblock.asm**

.code16 # Assemble for 16-bit mode

.globl start

start:

cli # BIOS enabled interrupts; disable

7c00: fa cli

# Zero data segment registers DS, ES, and SS.

xorw %ax,%ax # Set %ax to zero

7c01: 31 c0 xor %eax,%eax

movw %ax,%ds # -> Data Segment

7c03: 8e d8 mov %eax,%ds

movw %ax,%es # -> Extra Segment

7c05: 8e c0 mov %eax,%es

movw %ax,%ss # -> Stack Segment

7c07: 8e d0 mov %eax,%ss

00007c09 <seta20.1>:

seta20.1:

inb $0x64,%al # Wait for not busy

7c09: e4 64 in $0x64,%al

testb $0x2,%al

7c0b: a8 02 test $0x2,%al

jnz seta20.1

7c0d: 75 fa jne 7c09 <seta20.1>

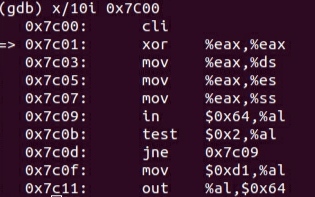
movb $0xd1,%al # 0xd1 -> port 0x64

7c0f: b0 d1 mov $0xd1,%al

outb %al,$0x64

7c11: e6 64 out %al,$0x64

**In GDB:**

****

**Tracing into Bootmain and Readsect:**

Statements in readsect in bootmain.c:

1. waitdisk(); // Issue command

Assembly Instruction:

**7c98: e8 e1 ff ff ff call 7c7e <waitdisk>**

1. outb(0x1F2, 1); // count = 1
2. outb(0x1F3, offset);
3. outb(0x1F4, offset >> 8);

Assembly Instruction:

**7cb0: 89 d8 mov %ebx,%eax**

**7cb2: c1 e8 08 shr $0x8,%eax**

**7cb5: ba f4 01 00 00 mov $0x1f4,%edx**

**7cba: ee out %al,(%dx)**

1. outb(0x1F5, offset >> 16);

Assembly Instruction:

**7cbb: 89 d8 mov %ebx,%eax**

**7cbd: c1 e8 10 shr $0x10,%eax**

**7cc0: ba f5 01 00 00 mov $0x1f5,%edx**

**7cc5: ee out %al,(%dx)**

1. outb(0x1F6, (offset >> 24) | 0xE0);

Assembly Instruction:

**7cc6: 89 d8 mov %ebx,%eax**

**7cc8: c1 e8 18 shr $0x18,%eax**

**7ccb: 83 c8 e0 or $0xffffffe0,%eax**

**7cce: ba f6 01 00 00 mov $0x1f6,%edx**

**7cd3: ee out %al,(%dx)**

**7cd4: b8 20 00 00 00 mov $0x20,%eax**

**7cd9: ba f7 01 00 00 mov $0x1f7,%edx**

**7cde: ee out %al,(%dx)**

1. outb(0x1F7, 0x20); // cmd 0x20 - read sectors
2. waitdisk(); // Read data.

Assembly Instruction:

**7cdf: e8 9a ff ff ff call 7c7e <waitdisk>**

1. insl(0x1F0, dst, SECTSIZE/4);

**Begin and end of loop which reads remaining sectors:**

for(; pa < epa; pa += SECTSIZE, offset++)

readsect(pa, offset);

**What code will run after running out of loop:**

readseg(pa, ph->filesz, ph->off);

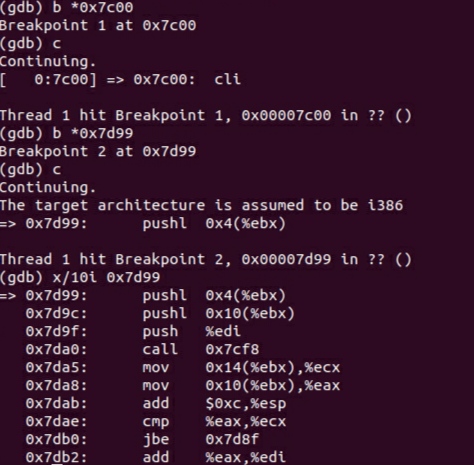
7d99: ff 73 04 pushl 0x4(%ebx)

7d9c: ff 73 10 pushl 0x10(%ebx)

7d9f: 57 push %edi

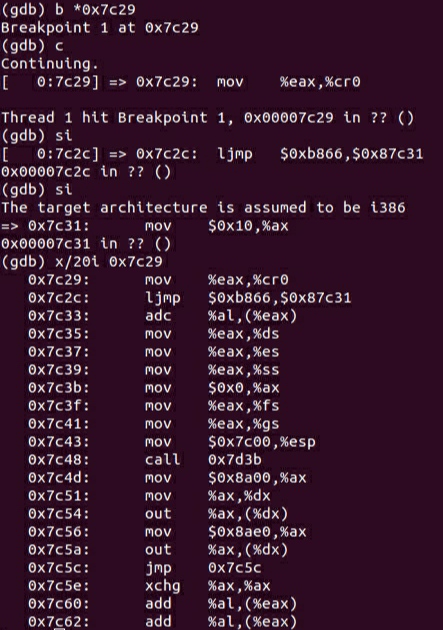
7da0: e8 53 ff ff ff call 7cf8 <readseg>

Setting up breakpoint at 0x7d99 as it is the first instruction after running out of readseg function and loop consecutively.



**Answer the following questions:**

**a)** The command **ljmp $(SEG\_KCODE<<3), $start32** causes the switch from 16 to 32-bit mode in the bootasm.S which occurs at address **0x7C31**.



**b)** Last Instruction of boot loader executed:

in bootmain.c it is:

**entry = (void(\*)(void))(elf->entry);**

**entry();**

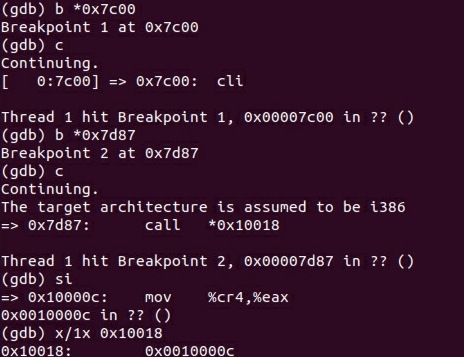
in bootblock.asm it is:

**7d87: ff 15 18 00 01 00 call \*0x10018**

The First Instruction of Kernel it just loaded is:

**0x10000c : mov %cr4,%eax**

Also the first instruction of the kernel should be at **0x10018**.



**c)** The boot loader reads the number the program headers in the ELF header and loads them all. It finds this information in the ELF header.

ph = (struct proghdr\*)((uchar\*)elf + elf->phoff);

eph = ph + elf->phnum;

for(; ph < eph; ph++){

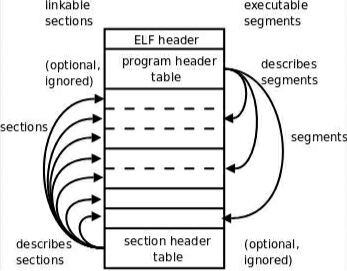
pa = (uchar\*)ph->paddr;

readseg(pa, ph->filesz, ph->off);

if(ph->memsz > ph->filesz)

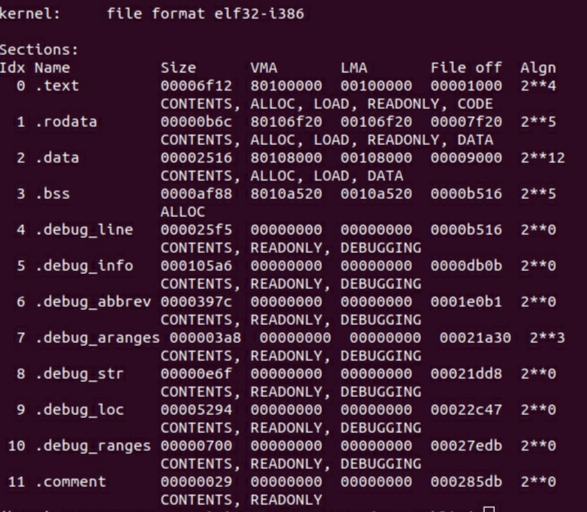
stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);

}

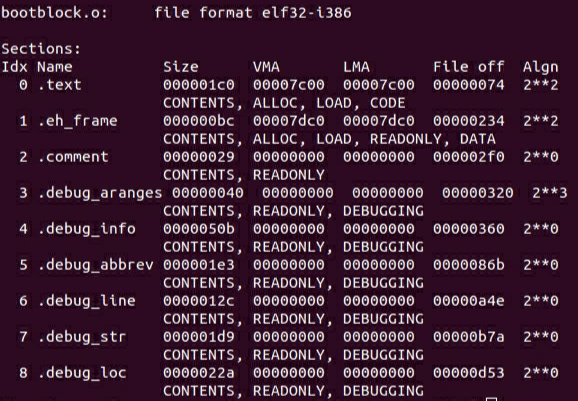


**EXERCISE 4:**

**$ objdump -h kernel**

****

**$ objdump -h bootblock.o**

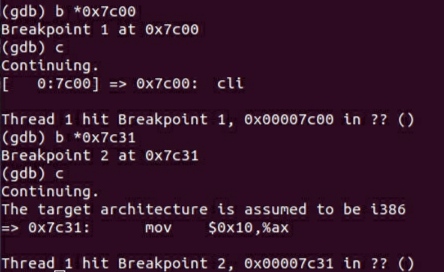
****

**Fields Explanation:**

1. Name: Program Sections Name(Program Headers)
2. Size: Size of the loaded section
3. VMA: Link Address, The link address of a section is the memory address from which the section expects to execute.
4. LMA: Load Address, The load address of a section is the memory address at which that section should be loaded into memory.
5. File off: is this section’s offset from the beginning of the file
6. Algn: It represents alignment
7. CONTENTS, ALLOC, LOAD, READONLY, DATA are flags. They represent that a particular section is to be LOADED or is READ ONLY.

**EXERCISE 5:**

When boot loader’s link address is 0x7C00 then commands are running properly and transition from 16 to 32 bit was occurring at **0x7C31** address location as seen below:



But when the boot loader’s link address is changed to any other address (we took **0x7C24** in this case), after running

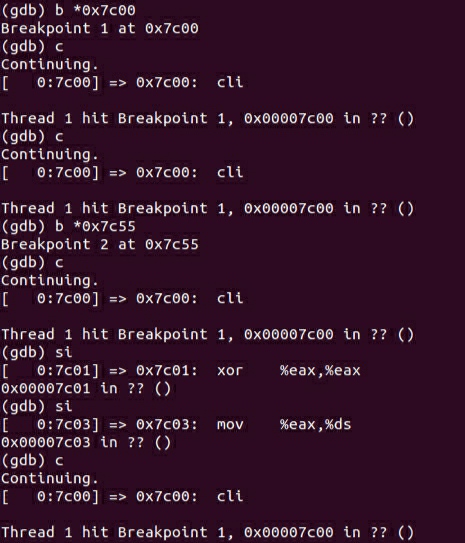
**make clean**

**make**

**and restarting gdb**

**and continuing by putting breakpoint from address location 0x7C00,**

then the boot loader is restarting again and again after running some instructions in the gdb.



As seen in the image above, we tried to run commands after continuing from breakpoint at 0x7C00 address location and we always end up hitting the same breakpoint at 0x7C00.

Also 16 to 32 bit architecture change didn’t occured as breakpoint **b \*0x7C31** is not hitted which should be responsible for architecture change in this case.

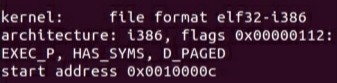
**ljmp $(SEG\_KCODE<<3), $start32** is the first instruction that breaks.

Before changing the link address of the boot loader, from address 0x7C00, after performing 2-3

**si 10** instructions, architecture changed from 16 to 32 bit.

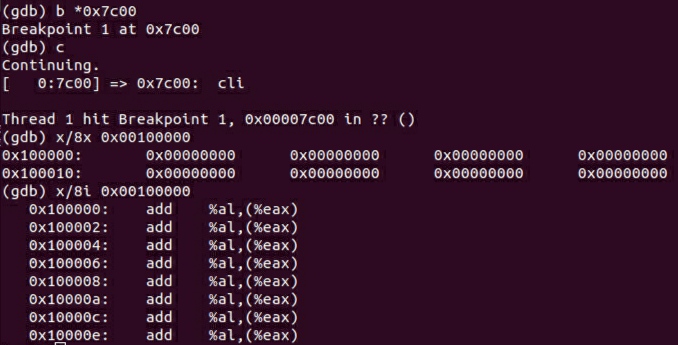
But after changing the link address to 0x7C24, architecture didn’t change which means that the boot loader is not loaded properly at the changed link address.

**$ objdump -f kernel**

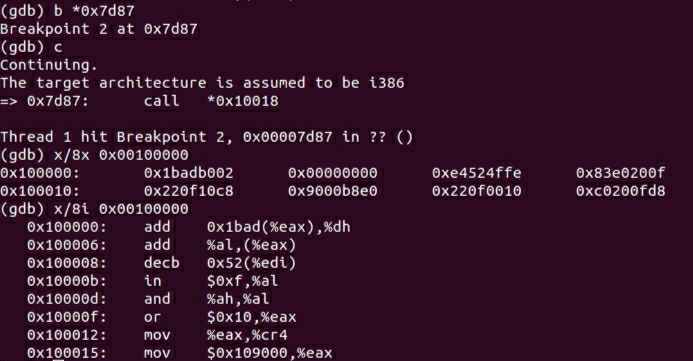
****

**EXERCISE 6:**

At the point when BIOS enters the boot loader (at first breakpoint):



At the point when the boot loader enters the kernel (at second breakpoint):



8 words of instruction at 0x00100000 at the point when BIOS enters the boot loader and 8 words of instruction at 0x00100000 at the point when the boot loader enters the kernel are different as when the BIOS enters and loads the boot loader, then it just loads it in memory location between 0x7C00 and 0x7DFF due to which all the 8 words of instructions are zero at 0x00100000. But when the boot loader enters the kernel, it already has performed the 16 to 32 bit transition and setting up of stack and also the bootloader loads kernel at memory locations including 0x00100000 which leads to new instructions at address 0x00100000.

**EXERCISE 7:**

In order to define our system call in xv6, we changed 5 files mentioned below.

1. **syscall.h:**

We added a new system call **#define SYS\_wolfie 22** at 22nd position as 21 positions we already occupied by the inbuilt system calls in syscall.h.

1. **syscall.c:**

We added a pointer **[SYS\_wolfie] sys\_wolfie** to system call at 22nd position in syscall.c file in order to add our custom system call.

Then a function prototype **extern int sys\_wolfie(void);** is added in syscall.c file which will be called by system call number 22.

1. **sysproc.c:**

System call function is implemented in sysproc.c.

**int**

**sys\_wolfie(void){**

**// code is given in file sysproc.c**

**}**

1. **usys.S:**

For creating an interface for your user program to access system call we added the following line in usys.S.

**SYSCALL(wolfie)**

1. **user.h:**

We added the following function that the user program will be calling in user.h.

**int wolfie(void \*buf, uint size);**

Call to the above function from the user program will be simply mapped to system call number 22 which is defined as **SYS\_wolfie** preprocessor directive. The system knows what exactly is this system call and how to handle it.

**EXERCISE 8:**

We save a C program named **wolfietest.c** inside the source code directory of xv6 operating system.

Then we edit the MakeFile and added below changes in MakeFile:

1. Under the value UPROGS (present at line 168), we added **\_wolfietest\** at the end of UPROGS value.
2. Under the value EXTRA (present at line 251), we added **wolfietest.c\.**

Then we run

1. **make clean** (to delete previous object files)
2. **make** (to generate new object files having changes of Makefile)
3. **make qemu** (to start qemu)
4. **ls** (to list all programs/files present in fs.img / second hard disk)
5. **wolfietest** (to run our application program to print wolfie on the screen)

**Image generated of wolfie from the qemu emulator/ terminal.**



**ls command listing all files present inside fs.img**

