CS 344 OS Lab

Assignment 0A

Mathematics and Computing

Group 2

by
Aayush Suthar, 220123004
Aaditya Jalan, 220123001
Avanish Gadhikar, 220123075
Chaitanya Chhabra, 220123012

Link for patchfile and ex1.c: [CS344-Ass0A-0B-MNCGroup2]

Task 1

The line required here was

```
__asm__("addl $1, %%eax;" :"=eax"(x) :"eax"(x));
```

Complete code in ex1.c file attached

Task 2

```
0x0000fff0 in ?? ()
+ symbol-file kernel
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb)
[f000:e062]    0xfe062: jne    0xd241d0b0
0x0000e062 in ?? ()
(gdb)
[f000:e066]    0xfe066: xor    %edx,%edx
0x0000e066 in ?? ()
(gdb)
[f000:e068]    0xfe068: mov    %edx,%ss
0x0000e068 in ?? ()
(gdb)
[f000:e06a]    0xfe06a: mov    $0x7000,%sp
0x0000e06a in ?? ()
(gdb)
[f000:e070]    0xfe070: mov    $0xfc1c,%dx
0x0000e070 in ?? ()
(gdb)
[f000:e076]    0xfe076: jmp    0x5576cf2d
0x0000e076 in ?? ()
(gdb)
```

Here we can observe that the BIOS, after being loaded at `0xf000:fff0` (which is almost the end of memory), jumped to `0xf000:e05b` to get more space to use.

A few commands we see here are:

mov
: Copy one register into another

cmpw
: Compares values but doesn't modify operands

jne
: Conditional jump if comparison was “not equal”

xor
: Logical XOR. XOR here of %edx with %edx sets it to zero.

Task 3

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) next
Cannot find bounds of current function
(gdb) step
Cannot find bounds of current function
(gdb) x/10i 0x7c00
=> 0x7c00:      cli
   0x7c01:      xor    %eax,%eax
   0x7c03:      mov    %eax,%ds
   0x7c05:      mov    %eax,%es
   0x7c07:      mov    %eax,%ss
   0x7c09:      in     $0x64,%al
   0x7c0b:      test   $0x2,%al
   0x7c0d:      jne    0x7c09
   0x7c0f:      mov    $0xd1,%al
   0x7c11:      out    %al,$0x64
(gdb)
```

Fig 3.1. GDB Execution

```
(gdb) b *0x7db9
Breakpoint 2 at 0x7db9
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7db9:      mov    $0x0,%eax

Thread 1 hit Breakpoint 2, 0x00007db9 in ?? ()
(gdb) x/10i 0x7db9
=> 0x7db9:      mov    $0x0,%eax
   0x7dbe:      cld
   0x7dbf:      rep stos %al,%es:(%edi)
   0x7dc1:      jmp    0x7d8f
   0x7dc3:      add    %al,(%eax)
   0x7dc5:      add    %al,(%eax)
   0x7dc7:      add    %al,(%eax)
   0x7dc9:      add    %al,(%eax)
   0x7dcb:      add    %al,(%eax)
   0x7dcd:      add    %al,(%eax)
```

Fig 3.1.1 Breakpoint at 0x7c00 followed by disassembly of next 10 instructions

```
Disassembly of section .text:

00007c00 <start>:
# with %cs=0 %ip=7c00.

.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>:

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
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
```

Fig 3.2. bootblock.asm

The file for assembly instructions is bootblock.asm

```
start:
  cli                         # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax             # Set %ax to zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

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

Fig 3.3. bootasm.S

```
// Read a single sector at offset into dst.
void
readsect(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
  outb(0x1F6, (offset >> 24) | 0xE0);
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  insl(0x1F0, dst, SECTSIZE/4);
}

// Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
// Might copy more than asked.
```

Fig 3.4. readsect()

```
00007c8c <readsect>:

// Read a single sector at offset into dst.
void
readsect(void *dst, uint offset)
{
    7c8c:       55                      push   %ebp
    7c8d:       89 e5                   mov    %esp,%ebp
    7c8f:       57                      push   %edi
    7c90:       53                      push   %ebx
    7c91:       8b 5d 0c                mov    0xc(%ebp),%ebx
  // Issue command.
  waitdisk();
    7c94:       e8 e5 ff ff ff          call   7c7e <waitdisk>
}
```

Fig 3.5. bootblock.asm readsect()

```
  for(; ph < eph; ph++){
    7d7d:       39 f3                   cmp    %esi,%ebx
    7d7f:       72 15                   jb     7d96 <bootmain+0x59>
  entry();
    7d81:       ff 15 18 00 01 00       call   *0x10018
}
    7d87:       8d 65 f4                lea    -0xc(%ebp),%esp
    7d8a:       5b                      pop    %ebx
    7d8b:       5e                      pop    %esi
    7d8c:       5f                      pop    %edi
    7d8d:       5d                      pop    %ebp
    7d8e:       c3                      ret
  for(; ph < eph; ph++){
    7d8f:       83 c3 20                add    $0x20,%ebx
    7d92:       39 de                   cmp    %ebx,%esi
    7d94:       76 eb                   jbe    7d81 <bootmain+0x44>
    pa = (uchar*)ph->paddr;
    7d96:       8b 7b 0c                mov    0xc(%ebx),%edi
    readseg(pa, ph->filesz, ph->off);
    7d99:       83 ec 04                sub    $0x4,%esp
    7d9c:       ff 73 04                push   0x4(%ebx)
    7d9f:       ff 73 10                push   0x10(%ebx)
    7da2:       57                      push   %edi
    7da3:       e8 4c ff ff ff          call   7cf4 <readseg>
    if(ph->memsz > ph->filesz)
    7da8:       8b 4b 14                mov    0x14(%ebx),%ecx
    7dab:       8b 43 10                mov    0x10(%ebx),%eax
    7dae:       83 c4 10                add    $0x10,%esp
    7db1:       39 c1                   cmp    %eax,%ecx
    7db3:       76 da                   jbe    7d8f <bootmain+0x52>
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
    7db5:       01 c7                   add    %eax,%edi
    7db7:       29 c1                   sub    %eax,%ecx
}
```

Fig 3.6. for loop blockasm.S

It reads the remaining sectors of the kernel into memory. Clearly from the code, the for loop is entered from memory location `0x7d7d`. Upon termination of the loop, the instruction at `0x7d81`- "**`call *0x10018`**" is run. This is the last instruction executed by the bootloader after which control is passed onto the kernel. Here the pointer to location `*0x10018` is the pointer to the entry field of the ELF header.

(a) At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

```
# Switch from real to protected mode.  Use a bootstrap GDT that makes
# virtual addresses map directly to physical addresses so that the
# effective memory map doesn't change during the transition.
lgdt    gdtdesc
movl    %cr0, %eax
orl     $CR0_PE, %eax
movl    %eax, %cr0
```

The lgdt command causes the switch from 16 to 32-bit mode.
(Real to Protected mode.)

```
//PAGEBREAK!
  # Complete the transition to 32-bit protected mode by using a long jmp
  # to reload %cs and %eip.  The segment descriptors are set up with no
  # translation, so that the mapping is still the identity mapping.
  ljmp    $(SEG_KCODE<<3), $start32

.code32  # Tell assembler to generate 32-bit code now.
```

This "`ljmp $(SEG_KCODE<<3), $start32`" command is where the transition is completed.

(b) What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

Last instruction that Boot Loader executes:

In bootmain.c it is: `entry = (void(*)(void))(elf→entry);`

In bootblock.asm: `7d87: ff 15 18 00 01 00 call *0x10018`

The first instruction that Kernel executes:

The instruction is "`movl %cr4, %eax`" present at `0x0010000c`

(c) How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

```
// Load each program segment (ignores ph flags).
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);
}
```

The bootloader determines how many sectors to read to load the entire kernel from the disk using information found in the ELF Header.

In the figure above, the bootloader initially reads 8 sectors (4096 bytes) from the disk, including the ELF Header of the kernel image. When inspecting the ELF header in GDB, it contains two attributes: phoff, which is the offset where program headers start, and phnum, which indicates the number of program headers.
These attributes guide the bootloader in loading the entire kernel from the disk:

- The ph pointer is set to the first program header
- The eph pointer is set just past the last program header to be read
- For loop condition is “ph < eph” because these are all the sectors that contain the kernel.

Task 4

```
root@chai-desktop:~/xv6-public# objdump -h kernel

kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00007188  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000009cb  801071a0  001071a0  000081a0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00002516  80108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          0000afb0  8010a520  0010a520  0000b516  2**5
                  ALLOC
  4 .debug_line   00006aaf  00000000  00000000  0000b516  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_info   00010e14  00000000  00000000  00011fc5  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 00004496  00000000  00000000  00022dd9  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 000003b0  00000000  00000000  00027270  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000de8  00000000  00000000  00027620  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 000050b1  00000000  00000000  00028408  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000845  00000000  00000000  0002d4b9  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .debug_line_str 0000012b  00000000  00000000  0002dcfe  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 12 .comment      0000002b  00000000  00000000  0002de29  2**0
                  CONTENTS, READONLY
root@chai-desktop:~/xv6-public#
```

Fig 4.1. objdump -h kernel

```
root@chai-desktop:~/xv6-public# objdump -h bootblock.o

bootblock.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001c3  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000b0  00007dc4  00007dc4  00000238  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      0000002b  00000000  00000000  000002e8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000318  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   00000585  00000000  00000000  00000358  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 0000023c  00000000  00000000  000008dd  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_line   00000283  00000000  00000000  00000b19  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_str    000001ff  00000000  00000000  00000d9c  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_line_str 0000003a  00000000  00000000  00000f9b  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 0000018d  00000000  00000000  00000fd5  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000033  00000000  00000000  00001162  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
root@chai-desktop:~/xv6-public#
```

Fig 4.2. objdump -h bootblock.o

The figures display various sections of the kernel and bootblock.o binaries like

1. **Name**: Section name, like .text (instructions) or .data (initialized globals)
2. **Size**: Section size in bytes
3. **VMA**: [Virtual Memory Address] **Link address** where the section is expected to run
4. **LMA**: [Load Memory Address] **Load address** in memory. Duh
5. **File off**: Offset from the file's start on disk
6. **Algn**: Data alignment requirements

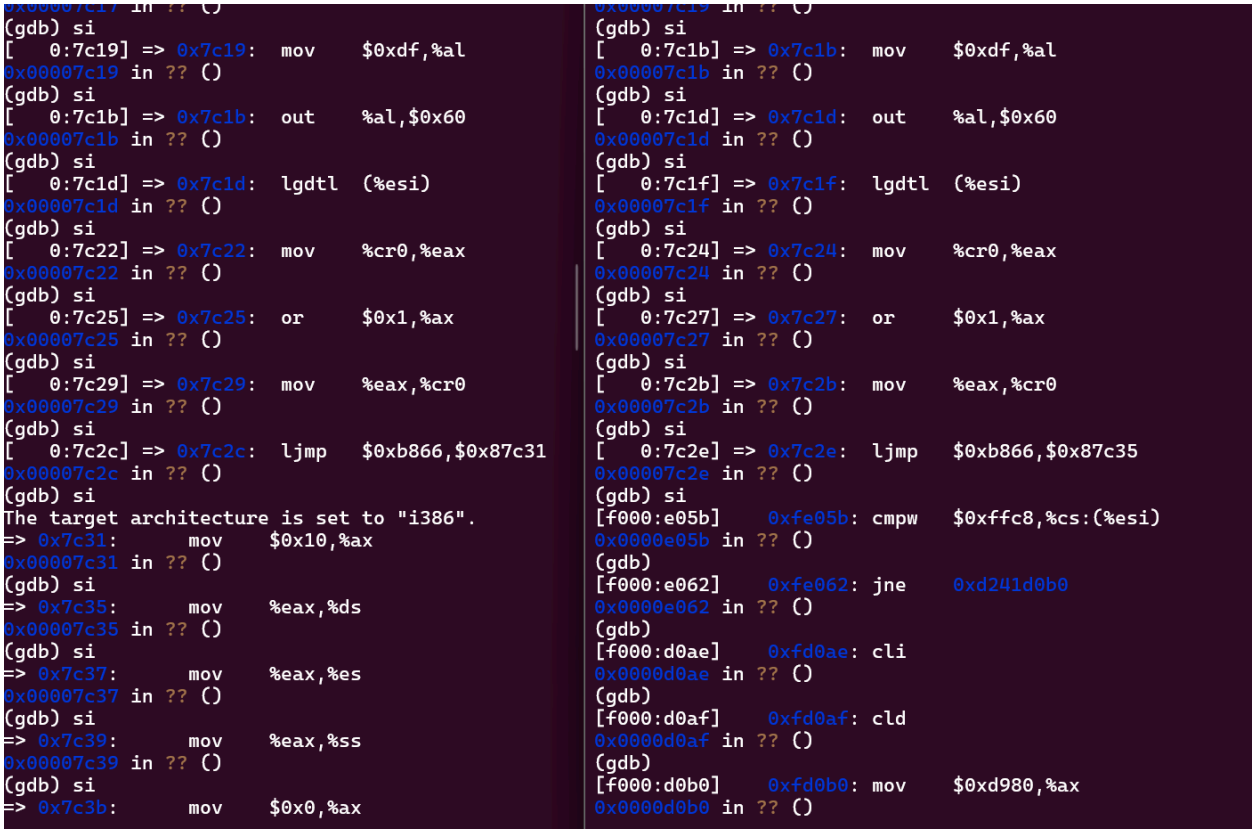
Observation:
For the kernel, VMA and LMA of the .text section are different (**0x80100000** and **0x00100000** respectively), indicating that it loads and executes from different addresses.

For the bootblock, VMA and LMA of the .text section are the same (**0x007c00** and **0x007c00** respectively), indicating that it loads and executes from the same address.

Task 5

For this task we changed the address in Makefile, from 7c00 to 7c02.

Left = 7c00 (True address) | **Right = 7c02 (Wrong address)**



```
(gdb) si
[   0:7c19] => 0x7c19:  mov    $0xdf,%al
0x00007c19 in ?? ()
(gdb) si
[   0:7c1b] => 0x7c1b:  out    %al,$0x60
0x00007c1b in ?? ()
(gdb) si
[   0:7c1d] => 0x7c1d:  lgdtl  (%esi)
0x00007c1d in ?? ()
(gdb) si
[   0:7c22] => 0x7c22:  mov    %cr0,%eax
0x00007c22 in ?? ()
(gdb) si
[   0:7c25] => 0x7c25:  or     $0x1,%ax
0x00007c25 in ?? ()
(gdb) si
[   0:7c29] => 0x7c29:  mov    %eax,%cr0
0x00007c29 in ?? ()
(gdb) si
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31
0x00007c2c in ?? ()
(gdb) si
The target architecture is set to "i386".
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb) si
=> 0x7c35:      mov    %eax,%ds
0x00007c35 in ?? ()
(gdb) si
=> 0x7c37:      mov    %eax,%es
0x00007c37 in ?? ()
(gdb) si
=> 0x7c39:      mov    %eax,%ss
0x00007c39 in ?? ()
(gdb) si
=> 0x7c3b:      mov    $0x0,%ax
```

```
(gdb) si
[   0:7c1b] => 0x7c1b:  mov    $0xdf,%al
0x00007c1b in ?? ()
(gdb) si
[   0:7c1d] => 0x7c1d:  out    %al,$0x60
0x00007c1d in ?? ()
(gdb) si
[   0:7c1f] => 0x7c1f:  lgdtl  (%esi)
0x00007c1f in ?? ()
(gdb) si
[   0:7c24] => 0x7c24:  mov    %cr0,%eax
0x00007c24 in ?? ()
(gdb) si
[   0:7c27] => 0x7c27:  or     $0x1,%ax
0x00007c27 in ?? ()
(gdb) si
[   0:7c2b] => 0x7c2b:  mov    %eax,%cr0
0x00007c2b in ?? ()
(gdb) si
[   0:7c2e] => 0x7c2e:  ljmp   $0xb866,$0x87c35
0x00007c2e in ?? ()
(gdb) si
[f000:e05b]     0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb)
[f000:e062]     0xfe062: jne    0xd241d0b0
0x0000e062 in ?? ()
(gdb)
[f000:d0ae]     0xfd0ae: cli
0x0000d0ae in ?? ()
(gdb)
[f000:d0af]     0xfd0af: cld
0x0000d0af in ?? ()
(gdb)
[f000:d0b0]     0xfd0b0: mov    $0xd980,%ax
0x0000d0b0 in ?? ()
```

The easiest difference to note is that every instruction on the right hand side is delayed by 2 bytes, above address 7c2c (on left) (or 7c2e on right), but the content of the instructions remains the same.

The first error occurs at instruction 7c2c “ljmp”. The 2nd argument is messed up. Its jump address is delayed by 4 bytes (0x87c35 instead of 0x87c31), which is the size of one instruction. Thus, *target architecture* is never set to *i386*, and incorrect instructions begin to execute.

We observe that the GUI for QEMU never finishes booting, and is in a perpetual state of loading. When the instruction pointer reaches the end of instructions, it overflows, restarts from the top, and hence never terminates.

Running *objdump -f kernel* :

```
root@chai-desktop:~/xv6-public# objdump -f kernel

kernel:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

This shows that the boot loader enters the kernel at 0x10000c.

Task 6

The point where **BIOS** enters the **Boot Loader** = 0x007c00
The point where **Boot Loader** enters the **Kernel** = 0x10000c

```
root@chai-desktop: ~/xv6-pul  ×   root@chai-desktop: ~/xv6-pu  ×   +
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x10000c:    mov    %cr4,%eax

Thread 1 hit Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb)
```

<u>Before</u> `0x10000c` Kernel entry

The eight words at `0x100000` contain zeroes, because they were initialized thusly.

<u>After</u> `0x10000c` Kernel entry

The eight words at `0x100000` now contain useful values stored by the OS.

CS 344 OS Lab

Assignment 0B

Mathematics and Computing

Group 2

by
Aayush Suthar, 220123004
Aaditya Jalan, 220123001
Avanish Gadhikar, 220123075
Chaitanya Chhabra, 220123012

Task 1

We have edited the following files for this task

syscall.h: This file assigns a number to every system call, sys_draw was given number 22. Line added: #define SYS_draw 22. There were 21 existing system calls and thus this was assigned 22.

```
#define SYS_draw   22  // Edited line, added the index of the new system call to the list.
```

syscall.c: This file contains an array of function pointers which uses numbers defined in syscall.h, we put our draw function here – no implementation is done. Extern is used because the implementation of the function is out of the current file.

Line 1 added: extern int sys_draw(void)

```
extern int sys_draw(void); // Edited line, added draw to the list.
```

Line 2 added: [SYS_draw] sys_draw;

```
[SYS_draw]    sys_draw,   // Edited line, added the draw function pointer.
```

sysproc.c : Implementation of our system call is written. We have declared 2 pointers to take data from the user program stack since the function is of void type given that it is a system call. The function checks if the given ascii text image size is less than the memory allocated by the user, if NOT, the function returns -1. If sufficient memory is allocated, then the function returns the number of bytes used, which in this case is the length of the string.

```
// function to implement the draw function call.
int
sys_draw(void){
  void* buff;
  uint size;

  argptr(0, (void*)&buff, sizeof(buff));
  argptr(1, (void*)&size, sizeof(size));

  char ascii_text [] = "                    **********\n             **********\n          ******   *    *      *
  if(sizeof(ascii_text) > size){
    return -1;
  }
  strncpy((char*)buff, ascii_text, size);
  return sizeof(ascii_text);
}
```

PS: The entire ASCII text image is not visible in this image.

user.h : Act as interface to access system call. Only the function prototype is specified here and not the implementation.
Line added: int draw(void*, int)

```
int draw(void*, uint size); // Edited line, added prototype definition of syscall.
```

usys.S: Also acts as interface to access system call
Line added: SYSCALL(draw)

```
SYSCALL(draw)
```

Task 2

Edited files:
Makefile: Necessary modifications to be done:

in UPROG list, `making` → `_drawtest\` is included
in EXTRA, drawtest.c is added

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _drawtest\
```

```
EXTRA=\
    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
    printf.c umalloc.c drawtest.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

Drawtest.c: Contains the instructions to allocate space so that the system call can print the ASCII art

```c
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(void){
    static char buff[2000];
    printf(1, "draw SYSCALL returned the value: %d\n", draw((void*) buff, 2000));
    printf(1, "%s\n", buff);
    exit();
}
```

a. **CASE 1:**

The following 2 results are the results of the function call run. We can see that in the first case when the memory allocated is 2000 as seen in drawtest, the function executes perfectly and prints out the number of bytes as well as keeps the buffer in the user stack which contains the string.

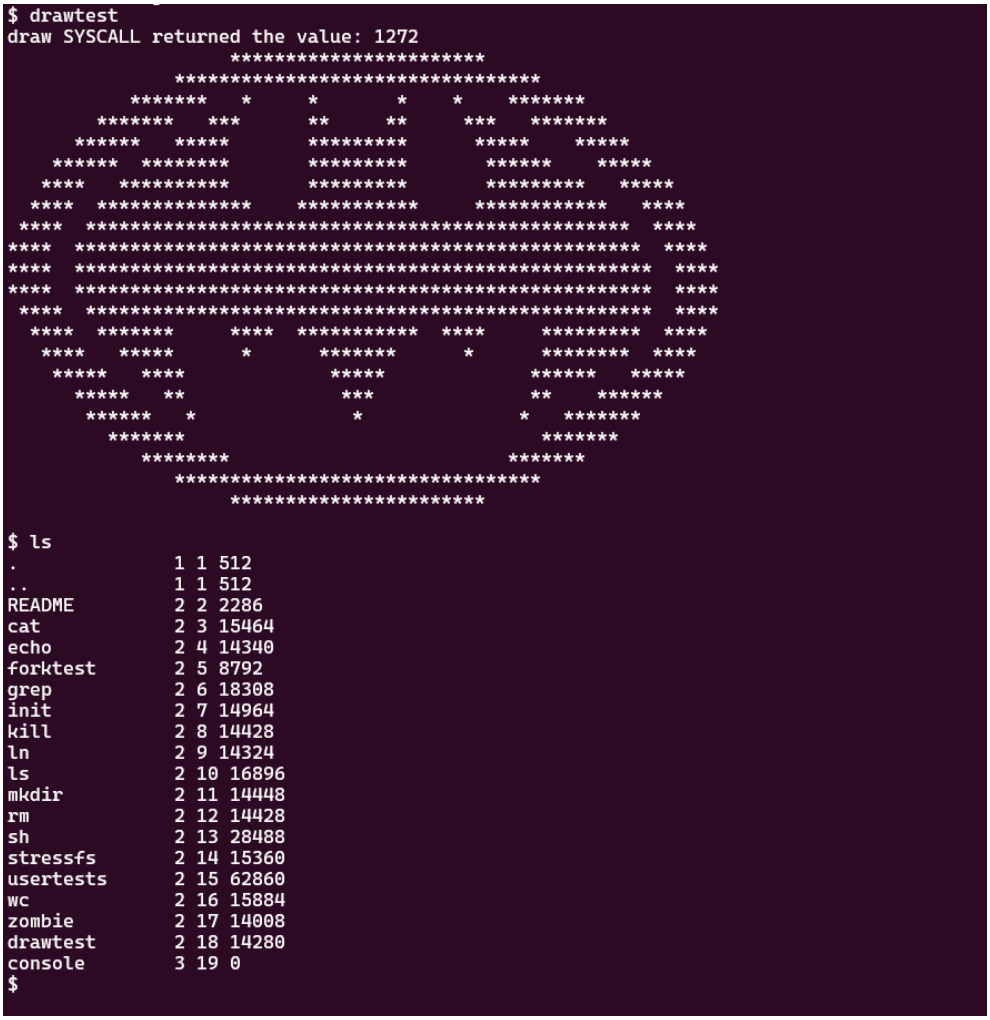


Figure 2.1. Output of “drawtest”, followed by “ls”

We can see in the above image that a binary executable of “drawtest” can be found in the fs.img

b. CASE 2:

If the memory allocated was insufficient, we can see that the program returns -1 as a “test_fail” value and thus does not print the ascii image either.

See figure 2.2.

We also created a patch file, “`patchfile.patch`”, to apply this patch to an original xv6-public folder.

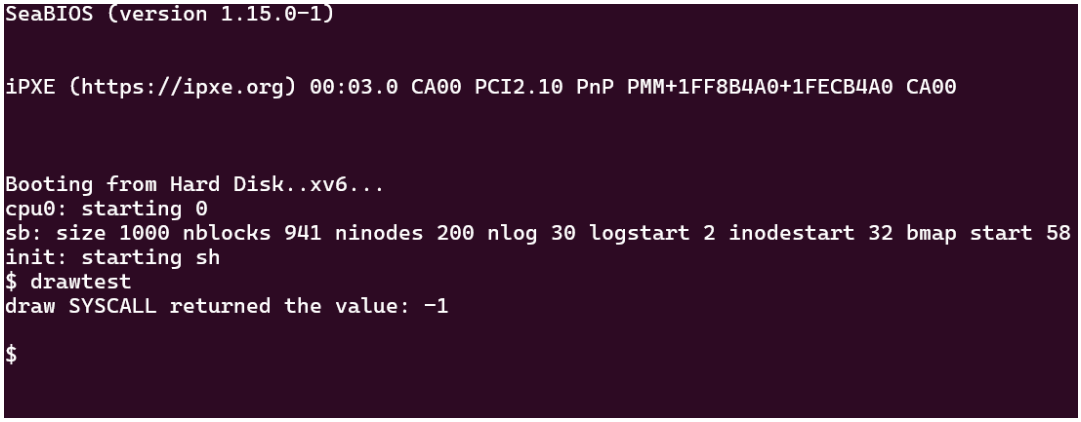


Fig 2.2. Execution in Insufficient Memory case