CS344 OPERATING SYSTEM LAB-0

AMAN BUCHA

200123006

Assignment-0A

Exercise-1

os1.cpp

```cpp
// Simple inline assembly example
#include <stdio.h>
int main(int argc, char **argv)
{
    int x = 1;
    printf("Hello x = %d\n", x);
    //
    // Put in-line assembly here to increment
    // the value of x by 1 using in-line assembly
    asm ( "addl %%ebx, %%eax;" : "=a" (x) : "a" (x), "b" (1) );
    //
    printf("Hello x = %d after increment\n",x);
    if(x == 2){
        printf("OK\n");
    }
    else{
        printf("ERROR\n");
    }
}
```

PROBLEMS OUTPUT TERMINAL JUPYTER DEBUG CONSOLE

```
cd "/home/amanbucha/Desktop/os/" && g++ os1.cpp -o os1 && "/home/amanbucha/Desktop/os/"os1
→  os cd "/home/amanbucha/Desktop/os/" && g++ os1.cpp -o os1 && "/home/amanbucha/Desktop/os/"os1
Hello x = 1
Hello x = 2 after increment
OK
→  os
```

Code Explained:

The operands are x and 1, stored in registers ebx and eax respectively. x is output The operation is add, so sum=x+1 is saved to x. Thus the value of x is incremented.

Exercise-2

si(Step Instruction) executes next machine instruction

```
(gdb) source .gdbinit
+ target remote localhost:26000
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0x3630,$0xf000e05b
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) si
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb) si
[f000:e066]    0xfe066: xor    %edx,%edx
0x0000e066 in ?? ()
(gdb) si
[f000:e068]    0xfe068: mov    %edx,%ss
0x0000e068 in ?? ()
```

Here, I have showed the first 5 instructions

Every instruction is output in the following format:

[Instruction CS: Instruction IP] Instruction Address: Instruction Code Operand1 Operand2*

*Optional, depends on instruction

The instructions are:

1. ljmp transfers program execution to 0x3630, 0xf000e05b
2. cmpw: compares the two words and sets zero flag if they are equal.
3. jne: jumps to 0xd241d0b2 conditionally
4. xor: performs logical xor on the operands. In this case, the operands are the same register. So it basically sets the value of edx register to be 0
5. mov: moves the content of ss register to edx register

Exercise-3

The code for readsect() in bootmain.c is given below

```
// 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);
}
```

The code of the for loop that reads the sectors of kernel from the disk given below:

```
// 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 first instruction of this for loop is

```
  7d8d: 39 f3                 cmp    %esi,%ebx
```

The last instruction of this for loop is

```
  7da4: 76 eb                 jbe    7d91 <bootmain+0x48>
```

The explanation for the first instruction is that the first operation on entering the for loop will be comparison between the values of ph and eph because the loop will run only when ph < eph.
The explanation of last instruction is that the loop ends when the values of ph and eph become equal and hence the loop jumps to the next instruction at 0x7d91. Hence the jump instruction will be the last instruction of the for loop. The next instruction after the for loop is

```
   7d91: ff 15 18 00 01 00      call   *0x10018
```

Making a breakpoint at that address and then stepping into further instructions gives the following output.

```
(gdb) b *0x7d91
Breakpoint 1 at 0x7d91
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 1, 0x00007d91 in ?? ()
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
(gdb) si
=> 0x10000f:    or     $0x10,%eax
0x0010000f in ?? ()
(gdb) si
=> 0x100012:    mov    %eax,%cr4
0x00100012 in ?? ()
(gdb) si
=> 0x100015:    mov    $0x10a000,%eax
0x00100015 in ?? ()
(gdb) si
=> 0x10001a:    mov    %eax,%cr3
0x0010001a in ?? ()
(gdb) si
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb) si
=> 0x100020:    or     $0x80010000,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) si
=> 0x100028:    mov    $0x8010c5c0,%esp
0x00100028 in ?? ()
(gdb) si
=> 0x10002d:    mov    $0x80103040,%eax
0x0010002d in ?? ()
(gdb) si
=> 0x100032:    jmp    *%eax
0x00100032 in ?? ()
(gdb) si
=> 0x80103040 <main>:   endbr32
main () at main.c:19
19      {
(gdb)
```

a)

```
.code32 # Tell assembler to generate 32-bit code now.
start32:
# Set up the protected-mode data segment registers
 movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
```

As can be inferred from the comments in the above code snippet from bootasm.S, the instruction:

movw $(SEG_KDATA<<3), %ax

is the point where processor start executing 32-bit code .

```
//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
```

As can be inferred from the comments in the above code snippet from bootasm.S, the instruction:

ljmp $(SEG_KCODE<<3), $start32

causes the switch from 16- to 32-bit mode.

b)

```
// Call the entry point from the ELF header.
// Does not return!
entry = (void(*)(void))(elf->entry);
entry();
```

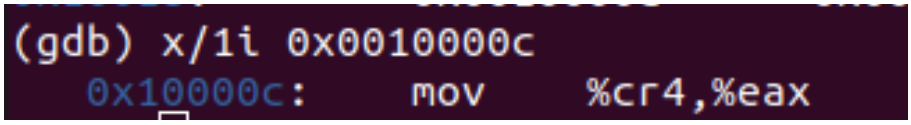
By looking at bootmain.c, I conclude that the last function called from bootloader is **entry**().

```
entry();
  7d91: ff 15 18 00 01 00     call   *0x10018
```

The above code snippet from bootblocker.asm suggests that the entry instruction resides at 7d91 and calls the instruction at 0x10018. By looking at the first word of memory stored at 0x10018, I find that

```
(gdb) x/1x 0x10018
0x10018:        0x0010000c
```

the first instruction of the kernel is stored at 0x0010000c



and is **mov %cr4,%eax**

c)

```
// 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 boot loader keeps loading segments while the condition “ph < eph” is true. The values of ph and eph are determined using attributes phoff and phnum of the ELF header. So the information stored in the ELF header helps the boot loader to decide how many sectors it has to read.

Exercise-4

```
→ ~ cd Desktop/os/xv6-public
→ xv6-public git:(master) ✗ objdump -h kernel

kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000070da  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000009cb  801070e0  001070e0  000080e0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00002516  80108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          0000af88  8010a520  0010a520  0000b516  2**5
                  ALLOC
  4 .debug_line   00006cb5  00000000  00000000  0000b516  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_info   000121ce  00000000  00000000  000121cb  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 00003fd7  00000000  00000000  00024399  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 000003a8  00000000  00000000  00028370  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000eb9  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295d1  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fdef  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      0000002a  00000000  00000000  00030af7  2**0
                  CONTENTS, READONLY
→ xv6-public git:(master) ✗
```

As we can see in the above screenshot, VMA and LMA of .text section is different indicating that it loads and executes from different addresses.

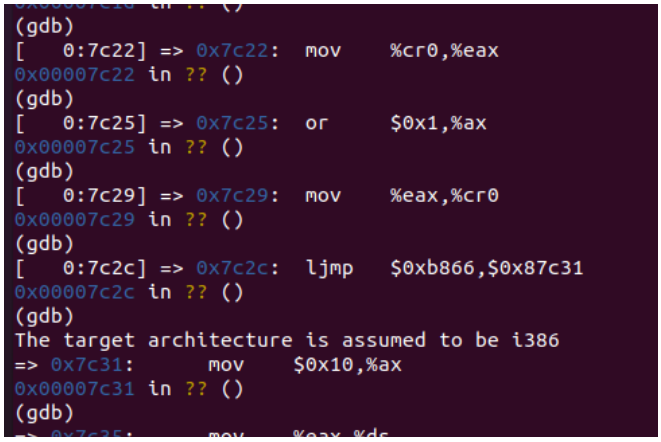
```
→ xv6-public git:(master) ✗ objdump -h bootblock.o

bootblock.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001d3  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000b0  00007dd4  00007dd4  00000248  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      0000002b  00000000  00000000  000002f8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000328  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   000005d2  00000000  00000000  00000368  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 0000022c  00000000  00000000  0000093a  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_line   0000029a  00000000  00000000  00000b66  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_str    0000022e  00000000  00000000  00000e00  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_loc    000002bb  00000000  00000000  0000102e  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_ranges 00000078  00000000  00000000  000012e9  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
→ xv6-public git:(master) ✗
```

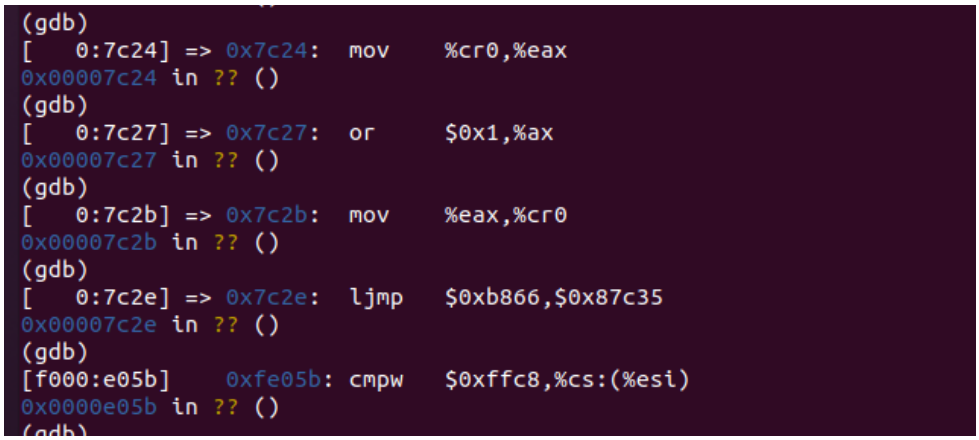
As we can see in the above screenshot, VMA and LMA of .text section is same

indicating that it loads and executes from the same address.

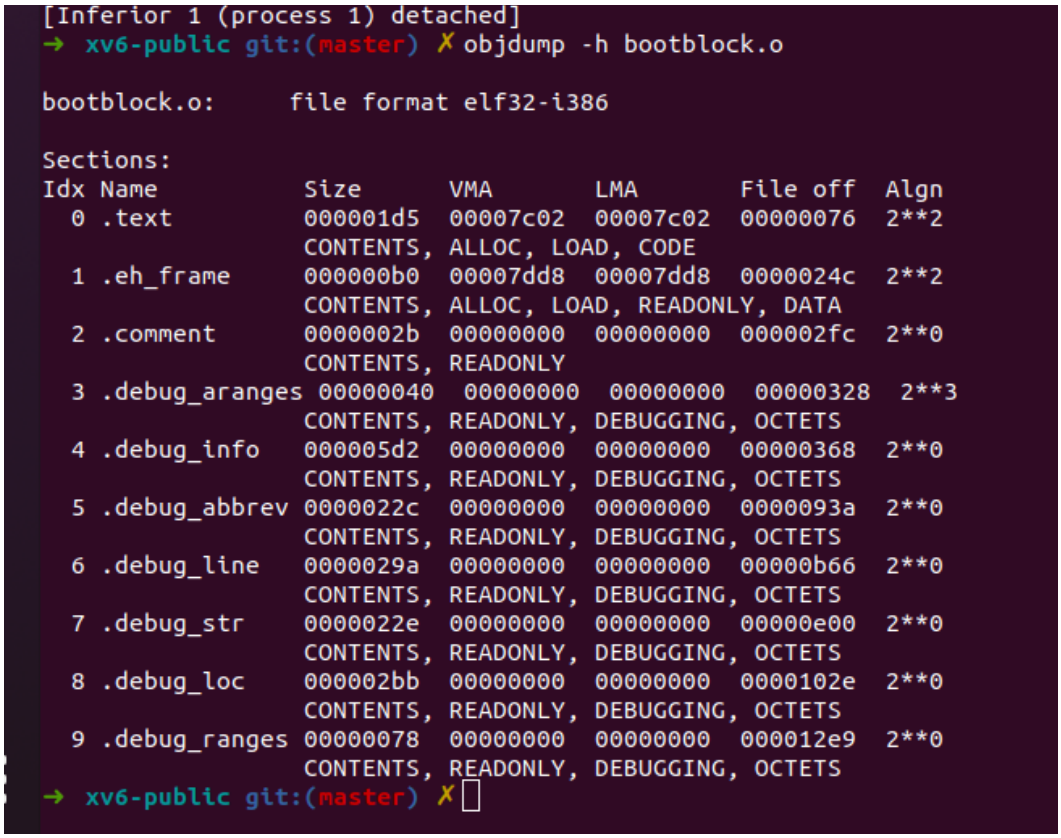
Exercise-5

I changed the link address from 0x7c00 to 0x7c02. Since no change has been done to the BIOS, it will run smoothly for both of the versions and hand over the control to the boot loader. From this point onwards, we have to check for differences between the two files. I did it by using si command repeatedly to get the next 15 (approx.) instructions and then comparing the outputs of the two files.
The first picture is when the link address was correctly set to 0x7c00 and the second picture is when it waschanged to 0x7c02. As we can see, in the second picture, the control should have jumped to address 0x87c35 in accordance with the ljmp instruction, but instead it abruptly goes to 0xfe05b





I am attaching a screenshot of the new object dump file, as a proof that I changed the link address



Exercise-6

I put a breakpoint at the address where the bootloader starts, i.e., 0x7c00. Then I continue the execution till this breakpoint. Now, I check the values of 8 words starting at address 0x00100000. I find all eight of them to be 0

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b *0x0010000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    mov    %cr4,%eax

Thread 1 hit Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb)
```

Next, I put a breakpoint at the address where the kernel gets the control,, i.e., 0x0010000c. Then I continue the execution till this breakpoint. Now, I check the values of 8 words starting at address 0x00100000. I find that all eight of them now have different non-zero values.

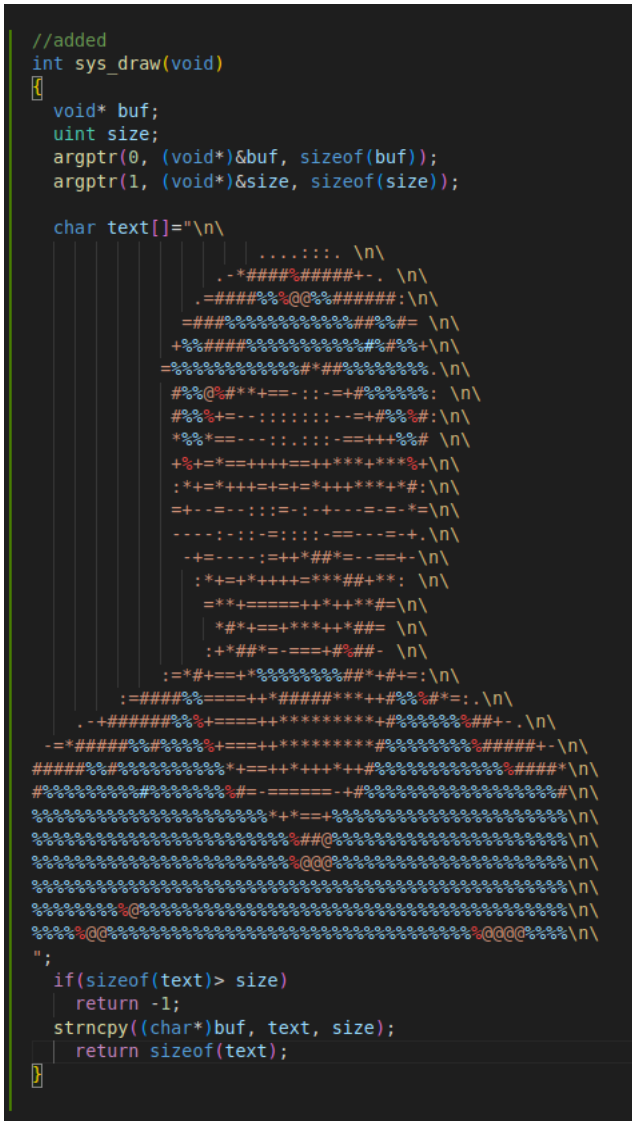
I suspect the reason for this is the fact the address 0x00100000 is the address from where the kernel is loaded into the memory. Before the kernel is loaded into the memory, this address contains no data (i.e. garbage value). We know that uninitialized values are given 0/NULL values.

Hence, when we tried to read the 8 words of memory at 0x00100000 at the first breakpoint, we got all zeroes since no data had been loaded until that point. When we check the values at the second breakpoint, the kernel has already been loaded into the memory and thus this address now contains meaningful data instead of zeroes

Assignment-0B

Exercise-1

I added the following code snippet in sysproc.c



```
//added
int sys_draw(void)
{
  void* buf;
  uint size;
  argptr(0, (void*)&buf, sizeof(buf));
  argptr(1, (void*)&size, sizeof(size));

  char text[]="\n\
                    ....:::. \n\
                 .-*####%#####+-. \n\
               .=#####%%@@%%######:\n\
              =###%%%%%%%%%%%%%%%%%%= \n\
             +%%####%%%%%%%%%%%%%%%%%%+\n\
            =%%%%%%%%%%%%%%%%%%%%%%%%%%.\n\
            #%%@%#**+==-::-=+#%%%%%%: \n\
            #%%%+=--:::::::--=+#%%%%#:\n\
            *%%*==---::.:::-==+++%%# \n\
            +%+=*==++++==++***+***%+\n\
            :*+=*+++=+=+=*+++***+*#:\n\
            =+--=--:::=-:-+---=-=-*=\n\
            ----:-::-=::::-==---=-+.\n\
             -+=----:=++*##*=--===+-\n\
              :*+=+*++++=***##+**: \n\
               =**+======++*++**#=\n\
                *#*+==+***++*##= \n\
               :+*##*=-===+#%##- \n\
             :=*#+==+*%%%%%%%%##*+#+=:\n\
        :=####%%====++*#####***++#%%%#*=:.\n\
     .-+#######%%%+====++*********+#%%%%%%##+-.\n\
  -=*######%#%%%%%+===++*********#%%%%%%%%%#####+-\n\
 #####%%#%%%%%%%%%*+==++*+++*++#%%%%%%%%%%%%%####*\n\
 #%%%%%%%%#%%%%%%%%#=-======-+#%%%%%%%%%%%%%%%%%%#\n\
 %%%%%%%%%%%%%%%%%%%%%%*+*==+%%%%%%%%%%%%%%%%%%%%%\n\
 %%%%%%%%%%%%%%%%%%%%%%%%%##@%%%%%%%%%%%%%%%%%%%%%\n\
 %%%%%%%%%%%%%%%%%%%%%%%%%%%@@@%%%%%%%%%%%%%%%%%%%\n\
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n\
 %%%%%%%%%%@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n\
 %%%%%%@@%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n\
";
  if(sizeof(text)> size)
    return -1;
  strncpy((char*)buf, text, size);
    return sizeof(text);
}
```

Exercise-2

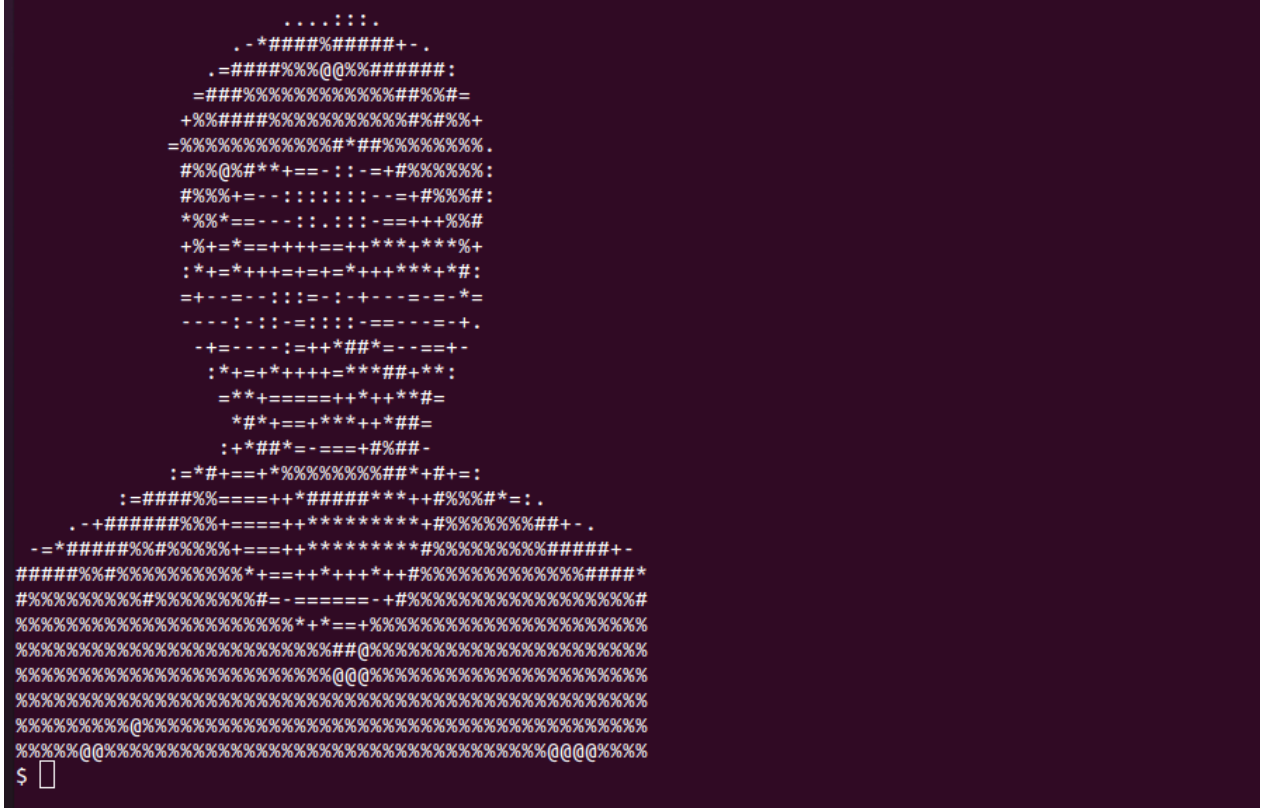
I created the following file in xv6-public and named it Drawtest.c

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

```c
#include "user.h"

int main(void)
{
   static char buf[2000];
   printf(1, "This is me %d\n", draw((void*) buf, 2000));
   printf(1, "%s", buf);
   exit();
}
```

I made the necessary changes in Makefile and then ran the code

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16284
echo           2 4 15136
forktest       2 5 9448
grep           2 6 18500
init           2 7 15720
kill           2 8 15164
ln             2 9 15016
ls             2 10 17648
mkdir          2 11 15260
rm             2 12 15240
sh             2 13 27880
stressfs       2 14 16152
usertests      2 15 67256
wc             2 16 17016
zombie         2 17 14828
Drawtest       2 18 14984
console        3 19 0
$ Drawtest
This is me 1254
```



```
$
```