CS344 LAB 1 Assignment

Yamini Sunil Tasare 220123071
Vishal Choudhary 220123070
Sneh Dadhania 220123062
Subhrajyoti Kunda Roy 220123064

PART 0A-1

Exercise 1

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int x = 1;
    printf("Hello x = %d\n", x);

    // Inline assembly 
    __asm__("incl %0" : "=r"(x) : "0"(x));

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

    if(x == 2){
        printf("OK\n");
    }
    else{
        printf("ERROR\n");
    }
}
```

```
PS E:\whyy> gcc ex1.c
PS E:\whyy> ./a.exe
Hello x = 1
Hello x = 2 after increment
OK
PS E:\whyy> 
```

Exercise 2

```
(gdb) target remote localhost:26000
Remote debugging using localhost:26000
mycpu () at proc.c:48
48          for (i = 0; i < ncpu; ++i) {
(gdb) si
0x80103907      48          for (i = 0; i < ncpu; ++i) {
(gdb) si
0x80103909      48          for (i = 0; i < ncpu; ++i) {
(gdb) si
0x8010390b      48          for (i = 0; i < ncpu; ++i) {
(gdb) si
0x8010390d      48          for (i = 0; i < ncpu; ++i) {
(gdb) si
49            if (cpus[i].apicid == apicid)
(gdb) si
0x8010391d      49            if (cpus[i].apicid == apicid)
(gdb) si
0x80103924      49            if (cpus[i].apicid == apicid)
(gdb) si
0x80103926      49            if (cpus[i].apicid == apicid)
(gdb) si
50              return &cpus[i];
(gdb) si
0x8010392b      50              return &cpus[i];
(gdb)
```

Commands

$ cd xv6-public
$ gdb kernel
// the above command launches GDB and loads the debugging symbols for the kernel executable. This allows us to debug the xv6 kernel.
(gdb) target remote localhost:26000

What the code seems to be doing :
Iterating through an array of CPU structures(cpus)
Checking if the apicid of each CPU matches a given apicid, if so, returning a pointer to that CPU structure.

This function appears to be part of the kernel's CPU management, likely used to find the data structure representing the current CPU based on its APIC ID (Advanced Programmable Interrupt Controller ID).

While `si` isn't directly accessing line 48 of proc.c, it's allowing us to step through the machine instructions that correspond to that line and subsequent lines of source code, instruction by instruction.Each time we enter `si`, GDB executes one assembly instruction.
After each `si`, GDB shows:
The memory address of the next instruction (e.g., 0x80103907, 0x80103909, etc.)
The corresponding line number in the source code.
The source code line itself.

Exercise 3

Setting breakpoint at 0x7c00 :

```
(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 set to "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    0xd241d0b0
0x0000e062 in ?? ()
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) si
[   0:7c01] => 0x7c01:  xor    %eax,%eax
0x00007c01 in ?? ()
(gdb) si
[   0:7c03] => 0x7c03:  mov    %eax,%ds
0x00007c03 in ?? ()
(gdb) |
```

a)
The processor starts executing 32-bit code -
Corresponding code in bootblock.asm ->

```
//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
    7c2c:       ea                      .byte 0xea
    7c2d:       31 7c 08 00             xor    %edi,0x0(%eax,%ecx,1)
```

Corresponding code in bootasm.S ->

```
0x7c29:        mov     %eax,%cr0
0x7c2c:        ljmp    $0xb866,$0x87c31
0x7c33:        adc     %al,(%eax)
0x7c35:        mov     %eax,%ds
```

b)
The last instruction of the boot loader executed
In bootblock.asm ->

```
    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
```

Corresponding code in bootmain.c ->

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

The address 0x10018 is called. (First instruction of Kernel)
First instruction in kernel -> movl %cr,%eax

```
// bootmain() loads an ELF kernel image from the disk starting at
// sector 1 and then jumps to the kernel entry routine.
```

c)

```
  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);
  }
```

ph is a pointer to the program header, elf contains the number of sectors required to fetch the entire kernel, eph is the pointer to the last sector.
The for loop loads each sector from ph upto eph incrementing ph at each iteration. This information is present in elf headers.

Exercise 5

In bootblock.asm, the lgdtl (%esi) will break or do the wrong thing if we set the boot loader's link to the wrong address.

```
  # 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
    7c1d:       0f 01 16                lgdtl  (%esi)
    7c20:       78 7c                   js     7c9e <readsect+0x12>
```

Changed the bootloader's link address to 0x8c00 from 0x7c00.

```
bootblock: bootasm.S bootmain.c
	$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
	$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
	$(LD) $(LDFLAGS) -N -e start -Ttext 0x8C00 -o bootblock.o bootasm.o bootmain.o
	$(OBJDUMP) -S bootblock.o > bootblock.asm
	$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
	./sign.pl bootblock
```

Further, ran the ‘make clean’ and ‘make’ commands

```
yamini@yamini:~/xv6-public$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*.o *.d *.asm *.sym vectors.S bootblock entryother \
initcode initcode.out kernel xv6.img fs.img kernelmemfs \
xv6memfs.img mkfs .gdbinit \
_cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _use
rtests _wc _zombie
yamini@yamini:~/xv6-public$ make
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -
m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -f
no-pic -O -nostdinc -I. -c bootmain.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -
m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -f
no-pic -nostdinc -I. -c bootasm.S
ld -m    elf_i386 -N -e start -Ttext 0x8C00 -o bootblock.o bootasm.o bootmai
n.o
objdump -S bootblock.o > bootblock.asm
objcopy -S -O binary -j .text bootblock.o bootblock
./sign.pl bootblock
boot block is 451 bytes (max 510)
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -
m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie
-c -o bio.o bio.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -
m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie
-c -o console.o console.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -
m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie
-c -o exec.o exec.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -
m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie
-c -o file.o file.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -
m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie
-c -o fs.o fs.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -
m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie
-c -o ide.o ide.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -
m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie
-c -o ioapic.o ioapic.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -
m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie
```

After running these commands, we observe that the address in the bootloader changes but the ip is still 7c00.

```
yamini@yamini:~/xv6-public$ cat bootblock.asm

bootblock.o:     file format elf32-i386


Disassembly of section .text:

00008c00 <start>:
# with %cs=0 %ip=7c00.

.code16                       # Assemble for 16-bit mode
.globl start
start:
  cli                         # BIOS enabled interrupts; disable
    8c00:       fa                      cli

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax             # Set %ax to zero
    8c01:       31 c0                   xor    %eax,%eax
  movw    %ax,%ds             # -> Data Segment
    8c03:       8e d8                   mov    %eax,%ds
  movw    %ax,%es             # -> Extra Segment
    8c05:       8e c0                   mov    %eax,%es
  movw    %ax,%ss             # -> Stack Segment
    8c07:       8e d0                   mov    %eax,%ss

00008c09 <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
    8c09:       e4 64                   in     $0x64,%al
  testb   $0x2,%al
    8c0b:       a8 02                   test   $0x2,%al
  jnz     seta20.1
    8c0d:       75 fa                   jne    8c09 <seta20.1>
```

Exercise 6

```
(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) x/8i 0x00100000
   0x100000:    add    %al,(%eax)
   0x100002:    add    %al,(%eax)
   0x100004:    add    %al,(%eax)
   0x100006:    add    %al,(%eax)
   0x100008:    add    %al,(%eax)
   0x10000a:    add    %al,(%eax)
   0x10000c:    add    %al,(%eax)
   0x10000e:    add    %al,(%eax)
(gdb) b *0x7d81
Breakpoint 2 at 0x7d81
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7d81:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d81 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
```

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 before the boot loader enters the kernel, it already has performed the 16 to 32 bit transition and setting up of stack which leads to new instructions at address 0x00100000

The address `0x7DFF` comes from the memory range allocated for the boot sector. When the BIOS loads the boot sector into memory, it places it at the address `0x7C00`, which is the starting address.

The boot sector typically occupies 512 bytes of memory, so it spans from `0x7C00` to `0x7DFF`. The `0x7DFF` is simply the last byte of this 512-byte space, calculated as:

Start address: `0x7C00`, Size: 512 bytes,End address: `0x7C00 + 0x0200 - 1 = 0x7DFF`.

PART 0A-1

```
yamini@yamini:~/xv6-public$ cat bootmain.c
// Boot loader.
//
// Part of the boot block, along with bootasm.S, which calls bootmain().
// bootasm.S has put the processor into protected 32-bit mode.
// bootmain() loads an ELF kernel image from the disk starting at
// sector 1 and then jumps to the kernel entry routine.

#include "types.h"
#include "elf.h"
#include "x86.h"
#include "memlayout.h"

#define SECTSIZE  512
```

PART 0B-1

Exercise 1

To define a system call in xv6, we made the following changes

1) syscall.h

We added a new system call #defineSYS_draw at 22nd position as 21 positions were already occupied by the inbuilt system calls in syscall.h

2) syscall.c

We added a pointer [SYS_draw] sys_draw 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_draw(void); is added in syscall.h file which will be called by system call number 22.

3) sysproc.c

System call function is implemented in sysproc.c

```
int sys_draw(void){
  char *a =
"    .--. \n"
"   |o_o |\n"
"   |:_/ |\n"
"  //   \\ \\\n"
" (|     | )\n"
" /'\\_ _/`\\\n"
" \\_/\\__/\n";

  void* buf;
  int size;

  if(argint(1,&size) < 0){
    return -1;
  }

  if(argptr(0,(char**)&buf,size) < 0){
    return -1;
  }
  int p = 0;
  for(p = 0; a[p] != '\0'; p++);

  if(p > size) p = -1;
  else {
    for(int i = 0; i < p; i++) ((char*)buf)[i] = a[i];
  }
  // returning size of wolfie
  // -1 is returned if buffer created is not enough to hold wolfie otherwise wolfie size is returned
  return p;
}
```

4) usys.S

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

SYSCALL(draw)

5) user.h

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

int draw(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_draw` preprocessor directive. The system knows what exactly this system calls and how to handle it.

Exercise 2

We save a C program named `testfile.c` inside the source code directory of xv6 operating system.
Then we edit the MakeFile and added below changes in MakeFile, Under the value `UPROGS` (present at line 168), we added `_testfile\` at the end of `UPROGS` value and Under the value `EXTRA` (present at line 251), we added `testfile.c\`.

Then we run:

'Make clean'

'Make'

'Make qemu'

'ls'

'testfile'.

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15472
echo           2 4 14348
forktest       2 5 8800
grep           2 6 18316
init           2 7 14972
kill           2 8 14436
ln             2 9 14332
ls             2 10 16904
mkdir          2 11 14456
rm             2 12 14436
sh             2 13 28496
stressfs       2 14 15368
usertests      2 15 62868
wc             2 16 15892
zombie         2 17 14016
testfile       2 18 14396
console        3 19 0
$ testfile
Penguin Size = 75 B
    .--.
   |o_o |
   |:_/ |
  //   \ \
 (|     | )
 /'\_ _/`\
 \__/\___/

pid 4 testfile: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffffffff--kill proc
$ |
```