ASSIGNMENT 0

CS344 - Operating Systems Laboratory

Name - Siddharth Bansal
Roll Number - 200101093
CSE Department

ASSIGNMENT 0A:

EXERCISE 1:

```
asm("add %%ebx, %%eax"
    : "=a" (x)
    : "a"(x), "b"(1));
```

Added inline assembly code

In this, we add the value x and 1 and then store the resultant in x, effectively incrementing the value of x. Here we are using extended inline assembly which takes in the statement(written in assembly), output and input separated by colons.

EXERCISE 2:

```
sid@sid-OptiPlex-3020: ~/Desktop/OS lab/xv6-public
line to your configuration file "/home/sid/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
--Type <RET> for more, q to quit, c to continue without paging--
        info "(gdb)Auto-loading safe path"
(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 ?? ()
```

The *si* instruction in gdb is used to execute one machine instruction and then pause the machine again. The above screenshot shows the first 4 instructions which are carried out by the machine.
[f000:fff0] 0xffff0: ljmp $0x3630,$0xf000e05b
Here **f000** is the **Starting Code Segment**, **fff0** is the **Starting Instruction Pointer**, **0xffff0** is the **physical address** where the instruction is stored in, ljmp is the **instruction**, **$0x3630** is the **destination code segment** and **$0xf000e5b** is the **destination instruction pointer**.

ljmp command mves the program counter to the address pointed by the destination code segment : destination instruction pointer pair.
cmpw subtracts the value of operand 1 and operand 2 but doesnt store the resultant instead only changing the flags.
jne is a conditional jump when ZF (the "zero" flag) is equal to 1.
xor performs a logical xor of its operands. In this case it performs the logical xor of the value at edx with itself setting it to 0 which is the same as mov $0x0000, %edx but more efficient.

Exercise 3:

```
// 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);
}
```

Code for readsect().

```
void
readsect(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
```

```
    7c9c:   e8 dd ff ff ff          call   7c7e <waitdisk>
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
    7cb4:   89 d8                   mov    %ebx,%eax
    7cb6:   c1 e8 08                shr    $0x8,%eax
    7cb9:   ba f4 01 00 00          mov    $0x1f4,%edx
    7cbe:   ee                      out    %al,(%dx)
  outb(0x1F5, offset >> 16);
    7cbf:   89 d8                   mov    %ebx,%eax
    7cc1:   c1 e8 10                shr    $0x10,%eax
    7cc4:   ba f5 01 00 00          mov    $0x1f5,%edx
    7cc9:   ee                      out    %al,(%dx)
  outb(0x1F6, (offset >> 24) | 0xE0);
    7cca:   89 d8                   mov    %ebx,%eax
    7ccc:   c1 e8 18                shr    $0x18,%eax
    7ccf:   83 c8 e0                or     $0xffffffe0,%eax
    7cd2:   ba f6 01 00 00          mov    $0x1f6,%edx
    7cd7:   ee                      out    %al,(%dx)
    7cd8:   b8 20 00 00 00          mov    $0x20,%eax
    7cdd:   ba f7 01 00 00          mov    $0x1f7,%edx
    7ce2:   ee                      out    %al,(%dx)
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
    7ce3:   e8 96 ff ff ff          call   7c7e <waitdisk>
    insl(0x1F0, dst, SECTSIZE/4);
}
```
Disassembled code for readsect().

```
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);
 }
```
For loop which reads the sectors of the kernel from the disc.

The begin of the for loop is-
7d8d: 39 f3   cmp   %esi,%ebx

And the end of the for loop is-
7da4:   76 eb   jbe   7d91 <bootmain+0x48>

When the loop is finished, the value of *ph* and *eph* becomes equal and the **PC** jumps to 7d91 and the next instruction to be executed is
7d91:   ff 15 18 00 01 00   call   *0x10018

After setting the breakpoint, continuing to that breakpoint and stepping thrugh the next few instructions, we get the following output on the terminal

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

Thread 1 hit Breakpoint 2, 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    $0x109000,%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    $0x8010b5c0,%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
```

```
# 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

//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.
start32:
# Set up the protected-mode data segment registers
 movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
```

code where the processor switches from 16-bit to 32-bit

(a) The processor starts executing code in 32-bit mode after the line ljmp $(SEG_KCODE<<3), $start32.

(b) Last instruction of the boot loader to be executed is 7d91: ff 15 18 00 01 00   call   *0x10018 , Which calls the kernel. The first instruction of the kernel can be found by using the *si* command a few times as shown in the screenshot showing the terminal output.
0x10000c: mov   %cr4,%eax (first instruction of the kernel)

```
 // 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 abve lines of code, present in *bootmain.c*, is what is used to load the kernel. xv6 first loads **ELF** headers of kernel into a memory location poined by *elf*. Then it stored the starting address of the first segment of the kernel to be loaded in *ph* by adding an offset(*elf->phoff*) to the starting address(*elf*). It also maintains an end pointer *eph* which points to the memory location after the end of the last segment. It then iterates over all the segments. For every segment, *pa* points to the address at which this segment is to be loaded and then it loads the current segment using ***readseg***. Then, if the memory assigned to this sector is more than the data copied, it initializes the excess memory with zeros.

(c) The bot loader loads the segments as long as **ph < eph** i.e. eph - ph segments are loaded. This value is determined using ***phnum*** attribute of the *elf* header. Thus, the information stored in the *elf* header helps the boot loader to decide how many sectors it has to read.

Exercise 4:

```
sid@sid-OptiPlex-3020:~/Desktop/OS lab/xv6-public$ 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
```

The kernel has different VMA and LMA in the .text section indicating that it loads and executes frm different addresses.

```
sid@sid-OptiPlex-3020:~/Desktop/OS lab/xv6-public$ 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
```

The boot sector loads and executes from the same address which can be inferred from the above screenshot as well as the VMA and LMA of the .text section are the same.

Exercise 5:

I changed the link address from **0x7c00** to **0x7c08**. Since no change has been done to the BIS, it will run normally for both the versions and hand over the control to the boot loader. Since we changed the link address for the bot loader, there should be changes from this point on. To check for these changes, i used si around *50* times for both the configurations and compared their outputs. The first command where a difference was spotted is shown below along with the next few instructions. The first image is from when the link address was correctly set to *0x7c00* and the second one is from when it was changed to *0x7c08*.

```
(gdb) b *0x7c2c
Breakpoint 1 at 0x7c2c
(gdb) c
Continuing.
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31

Thread 1 hit Breakpoint 1, 0x00007c2c in ?? ()
(gdb) si
The target architecture is assumed to be 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
0x00007c3b in ?? ()
(gdb) si
=> 0x7c3f:      mov    %eax,%fs
0x00007c3f in ?? ()
```

```
(gdb) b *0x7c2c
Breakpoint 2 at 0x7c2c
(gdb) c
Continuing.
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c39

Thread 1 hit Breakpoint 2, 0x00007c2c in ?? ()
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb) si
[f000:d0b0]    0xfd0b0: cli
0x0000d0b0 in ?? ()
(gdb) si
[f000:d0b1]    0xfd0b1: cld
0x0000d0b1 in ?? ()
(gdb) si
[f000:d0b2]    0xfd0b2: mov    $0xdb80,%ax
0x0000d0b2 in ?? ()
```

Exercise 6:

For this experiment, we have to examine the 8 words of memory at *0x00100000* at two different points, first when the BIOS enters boot loader and second when the boot loader enters the kernel, The breakpoints will be at 0x7c00 for BIOS to boot loader and at 0x0010000c for boot loader to kernel.

```
(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
```

The address *0x00100000* is the address where the kernel is loaded into the memory. Before the kernel is loaded, this address is uninitialised and by default, all uninitialised values in xv6 are set to 0. Hence when we tried to read 8 words of memory from *0x0010000* at the first breakpoint, we got all zeros. At the second breakpoint, the kernel has been already loaded into the memory and thus these addresses now contain non-zero meaningful data.

ASSIGNMENT 0B:

Exercise 1:

In order to define our own system call in xv6, changes were made in the following files -
syscall.h
syscall.c
sysproc.c
usys.s
user.h

1. Add the following line in *syscall.h* to add the custom system call.

```
#define SYS_draw   22
```

2. Add a pointer to the system call in the *syscall.c* file.

```
[SYS_draw]    sys_draw,
```

3. Add a prototype for our function call in *syscall.c*.

```
extern int sys_draw(void);
```

4. We implement the actual function in *sysproc.c.*

```c
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\
 *****   H*****H*******\n\
 ***     H-___-H  *********\n\
  ***    H     H      *******\n\
   **    H-___-H        *****\n\
     *   H     H         ****\n\
         H     H         ***\n\
         H-___-H         **\n\
         H     H         *\n\
         H-___-H\n\
\n\
         ALOHA!!\n\n";

  if(sizeof(text) > size)
    return -1;

  strncpy((char*)buf, text, size);
  return sizeof(text);

}
```

Function code

5. To add an interface for a user program to call the system call, we add

```
SYSCALL(draw)
```

to *usys.S* and

```c
int draw(void*, uint);
```

to *user.h*

With this, our required system call **sys_draw** is created.

Exercise 2:

To add a user program to call the system call that we made above, i made a file *Drawtest.s* inside xv6 folder and wrote the following code in it

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

int main(void)
{
   static char buf[2000];
   printf(1, "Draw syscall returns %d\n", draw((void*)buf, 2000));
   printf(1, "%s", buf);
```

```
    exit();
}
```

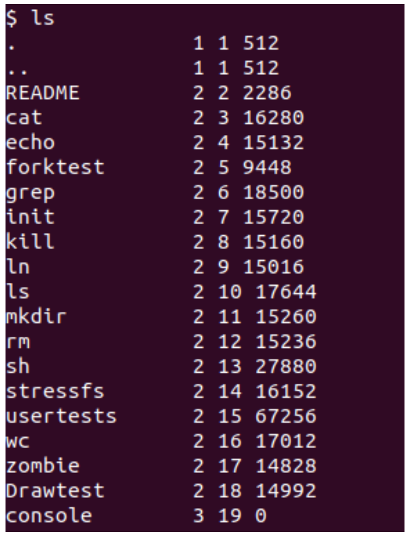
After this, i added *Drawtest* in the makefile under **EXTRA** and **UPROGS**

Working of the *Drawtest* and *ls* command

```
$ Drawtest
Draw syscall returns 464

         ****
     ********
    **  ******
     *   ******     ******
         ******   *********
          ****  *****   ***
          ***  ***     **
    *************       *
  ******************
 *****   H*****H*******
 ***     H-___-H  *********
  ***    H     H      *******
   **    H-___-H        *****
     *   H     H         ****
         H     H         ***
         H-___-H         **
         H     H         *
         H-___-H

         ALOHA!!
```



```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16280
echo           2 4 15132
forktest       2 5 9448
grep           2 6 18500
init           2 7 15720
kill           2 8 15160
ln             2 9 15016
ls             2 10 17644
mkdir          2 11 15260
rm             2 12 15236
sh             2 13 27880
stressfs       2 14 16152
usertests      2 15 67256
wc             2 16 17012
zombie         2 17 14828
Drawtest       2 18 14992
console        3 19 0
```