CS-344
ASSIGNMENT 0

PART A

Q1)

CODE (in VIM)

```
// Simple inline assembly example
//
#include <stdio.h>
int
main(int argc, char **argv)
{
        int x = 1;
        printf("Hello x = %d\n", x);

        // in-line assembly code to increment
        // the value of x by 1


        __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");
        }
}
```

OUTPUT

```
→  xv6-public git:(master) ✗ gcc ex1.c
→  xv6-public git:(master) ✗ ./a.out
Hello x = 1
Hello x = 2 after increment
OK
```

EXPLANATION

- The added asm code takes input operands as x & 1 and the output is saved back in x. Since the value of x after increment becomes 2, the output on terminal is OK, we conclude the value successfully incremented by 1.

Q2)

```
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        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 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) si
[f000:e066]    0xfe066: xor    %edx,%edx
0x0000e066 in ?? ()
(gdb) si
[f000:e068]    0xfe068: mov    %edx,%ss
0x0000e068 in ?? ()
(gdb)
```

EXPLANATION

- The “si” instruction is used to execute one machine line instruction and move to the next instruction
- The above screen shows the first five instructions in xv6 OS
    1. First instruction starts at [f000 : fff0]. 0xfff0 is the Physical Address of the instruction. ljmp is the instruction which means long jump and moves the code to the destination IP 0xf000e05b
    2. The cmp instruction compares the value of register cs with 0xffcb.
    3. The jnz instruction means jump on the no zero and jne means jump on not equal. It will jump to the location if zero flag is removed. Jnz is used to test if and when something is not equal whereas jne is after a compare instruction.
    4. XOR instruction does logical XOR operation on the values of the registers and stores it within itself.
    5. MOV instruction is to move the content of source register to destination register.

Q4)

vim pointers.c

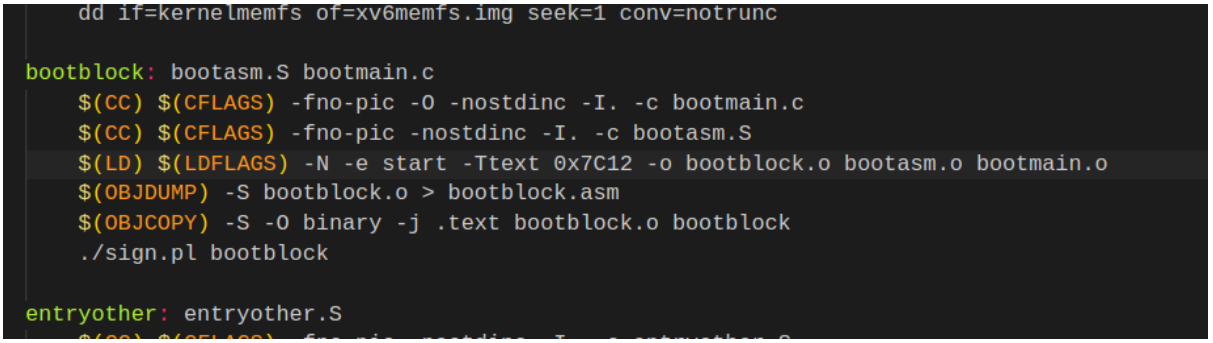
```
#include <stdio.h>
#include <stdlib.h>

void
f(void)
{
    int a[4];
    int *b = malloc(16);
    int *c;
    int i;

    printf("1: a = %p, b = %p, c = %p\n", a, b, c);

    c = a;
    for (i = 0; i < 4; i++)
        a[i] = 100 + i;
    c[0] = 200;
    printf("2: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
           a[0], a[1], a[2], a[3]);

    c[1] = 300;
    *(c + 2) = 301;
    3[c] = 302;
    printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
           a[0], a[1], a[2], a[3]);

    c = c + 1;
    *c = 400;
    printf("4: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
           a[0], a[1], a[2], a[3]);

    c = (int *) ((char *) c + 1);
    *c = 500;
    printf("5: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
           a[0], a[1], a[2], a[3]);

    b = (int *) a + 1;
    c = (int *) ((char *) a + 1);
    printf("6: a = %p, b = %p, c = %p\n", a, b, c);
}

int
main(int ac, char **av)
{
    f();
    return 0;
}
~
~
```

EXPLANATION

- Output Line 1 gives us the address of of pointers a, b, c
- Output Line 2 gives us the values of the array a, which have been updated to 100, 101, 102, 103. But then c[0] is changed to 200 which is reflected in a since the pointers are the same.
- Output Line 3 sets the values of c[1], c[2] and c[3] which are then updated.
- Output Line 4 sets c pointer to c[1] whose value is then updated to 400.
- Output Line 5 increments the pointer to c by 1 byte. This happens because the int* pointer is type casted to char*. The default +1 of int* pointer is 4 Bytes but char* The pointer is 1 Byte. The value for the next 4 bytes is updated to the relevant value of 500 in binary.
- Line 6 returns the final values stored in the a, b, c, pointers.

Q5)

I changed the link address from 0x7c00 to 0x7c12. nothing has changed Made in BIOS so it works smoothly both in version and hands Via the controller to the bootloader. From this point on you should check As for the difference between the two files, I did it by repeatedly using the si command Get the next 200 (approximately) statements and compare the output. 2 files. It will give you the first command where a difference was found See below along with the next three instructions. The first photo is from the link The address is correct when he is set to 0x7c00 and the second picture is changed to his 0x7c12. I've attached the output file from gdb to my submission. I have also attached the output files from both "`objdump -h bootmain.o`". The version of the output is different due to the link address change.



```
    dd if=kernelmemfs of=xv6memfs.img seek=1 conv=notrunc

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 0x7C12 -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

entryother: entryother.S
```

```
(gdb) si
[f000:cf3c]    0xfcf3c: or     $0x2,%al
0x0000cf3c in ?? ()
(gdb) si
[f000:cf3e]    0xfcf3e: out    %al,$0x92
0x0000cf3e in ?? ()
(gdb) si
[f000:cf40]    0xfcf40: mov    %cx,%ax
0x0000cf40 in ?? ()
(gdb) si
[f000:cf43]    0xfcf43: lidtl  %cs:(%esi)
0x0000cf43 in ?? ()
```

```
[f000:cf52]    0xfcf52: and    $0xffff,%cx
0x0000cf52 in ?? ()
(gdb) si
[f000:cf59]    0xfcf59: or     $0x1,%cx
0x0000cf59 in ?? ()
(gdb) si
[f000:cf5d]    0xfcf5d: mov    %ecx,%cr0
0x0000cf5d in ?? ()
(gdb) si
[f000:cf60]    0xfcf60: ljmpw  $0xf,$0xcf68
0x0000cf60 in ?? ()
(gdb) si
The target architecture is set to "i386".
=> 0xfcf68:     mov    $0x10,%ecx
0x000fcf68 in ?? ()
```

```
/home/pranav/Desktop/xv6-public
➜  xv6-public git:(master) ✗ objdump -h bootmain.o

bootmain.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000145  00000000  00000000  00000034  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000179  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000179  2**0
                  ALLOC
  3 .debug_info   00000561  00000000  00000000  00000179  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  4 .debug_abbrev 00000228  00000000  00000000  000006da  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_loclists 0000018d  00000000  00000000  00000902  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_aranges 00000020  00000000  00000000  00000a8f  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  7 .debug_rnglists 00000033  00000000  00000000  00000aaf  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_line   0000021c  00000000  00000000  00000ae2  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  9 .debug_str    000001ea  00000000  00000000  00000cfe  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_line_str 00000075  00000000  00000000  00000ee8  2**0
pranav@pranav-Predator-PH315-53:~/Desktop/xv6-public
```

Q6)

Here we are examining 8 words of memory at address 0x00100000 at two different times. The first is done when the BIOS enters the boot loader and second is done when the boot loader enters the kernel. Using command “x/8x 0x00100000” we try to see the 8 succeeding words but first we will have to set out break points for this.

The first break point will be at 0x7c00 when the BIOS hands over control to the boot loader and second breakpoint will be set at 0x0010000c, because at this point the bootloader passes control to the kernel.

Different values are obtained at both these points.

Explanation:

- The address 0x00100000 is a 1MB address from where the kernel is loaded into the memory. Initially it has no data and garbage values. It is set to 0 by default by xv6 OS. Hence reading the first 8 words at first breakpoint yields 0x00000000 memory addresses.
- At the second break point the kernel has already been loaded into the memory and hence we are returned meaningful addresses in the next 8 words.

PART B

Q7 & 8)

- Making a system call by user happens in user mode but for it to be executed the process must convert to kernel mode. Here we create a system call for the xv6 OS to print out a ACSII Art in the terminal of XV6.
- To execute this task we make changes to 6 files and create a new file.

Changed files:

1. syscall.h
2. syscall.c
3. sysproc.c
4. user.h
5. usys.S
6. Makefile

New files:

1. drawtest.c

- We first add pointer to the system call in syscall.c file. Since each system call must have a corresponding call number, sys_draw call is assigned a system call 22 since 21 calls were already present in the file. These call numbers are assigned in the header file syscall.h

```
#define SYS_draw    22
```

```
[SYS_draw]   sys_draw,
```

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

- Next we add an interfaceN to the user mode to be able to generate the system call. This is done in usys.S and user.h File

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

```
SYSCALL(draw)
```

- Next we create a new file drawtest.c file for the user to access the system call

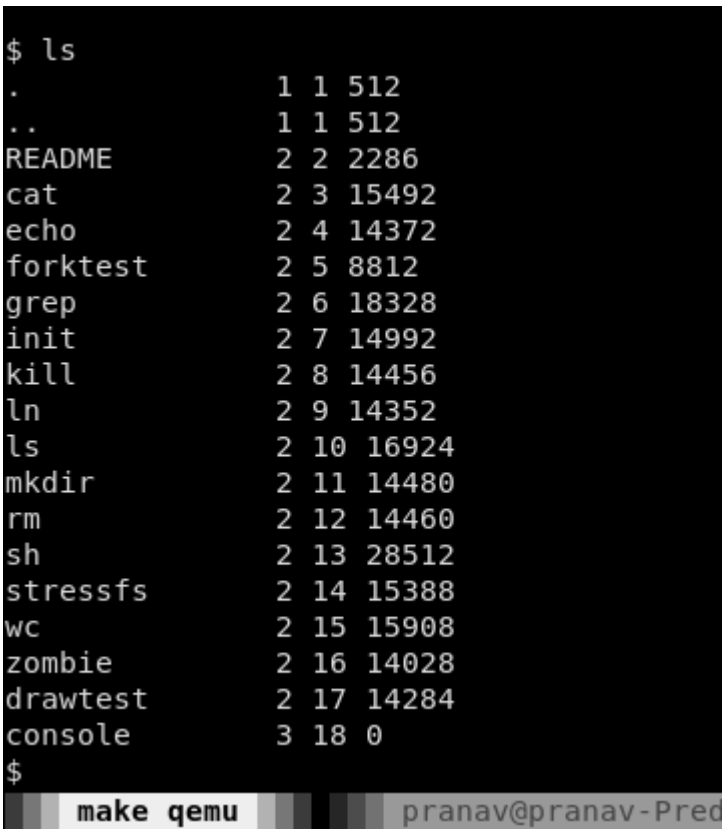
```
#include "types.h"
#include "stat.h"
#include "user.h"

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

    printf(1,"%s",buf);
    exit();
}
```

- In sysproc.c file I added my ASCII art

```
int sys_draw(void)
{
    void *buf;
    uint size;

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

    char text[] = "\n\
    PPPPPP  RRRRRR    AAA   NN   NN   AAA   VV     VV\n\
    PP   PP RR   RR  AAAAA  NNN  NN  AAAAA  VV     VV\n\
    PPPPPP  RRRRRR  AA   AA NN N NN AA   AA  VV   VV\n\
    PP      RR  RR  AAAAAAA NN  NNN AAAAAAA   VV VV\n\
    PP      RR   RR AA   AA NN   NN AA   AA    VVV\n\n";
    if (sizeof(text) > size)
        return -1;

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

- At last I run make clean and make qemu which runs the xv6 OS.
  Then I generate the system call using drawtest.

```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap
t 58
init: starting sh
$ drawtest
Draw sys call returns 267

    PPPPPP  RRRRRR    AAA   NN   NN   AAA   VV     VV
    PP   PP RR   RR  AAAAA  NNN  NN  AAAAA  VV     VV
    PPPPPP  RRRRRR  AA   AA NN N NN AA   AA  VV   VV
    PP      RR  RR  AAAAAAA NN  NNN AAAAAAA   VV VV
    PP      RR   RR AA   AA NN   NN AA   AA    VVV

$ _
```


$ ls
. 1 1 512
.. 1 1 512
README 2 2 2286
cat 2 3 15492
echo 2 4 14372
forktest 2 5 8812
grep 2 6 18328
init 2 7 14992
kill 2 8 14456
ln 2 9 14352
ls 2 10 16924
mkdir 2 11 14480
rm 2 12 14460
sh 2 13 28512
stressfs 2 14 15388
wc 2 15 15908
zombie 2 16 14028
drawtest 2 17 14284
console 3 18 0
$
make qemu
pranav@pranav-Pred