OS344 - Operating Systems Laboratory

Assignment 0

Harshul Gupta

200123023

Mathematics and Computing

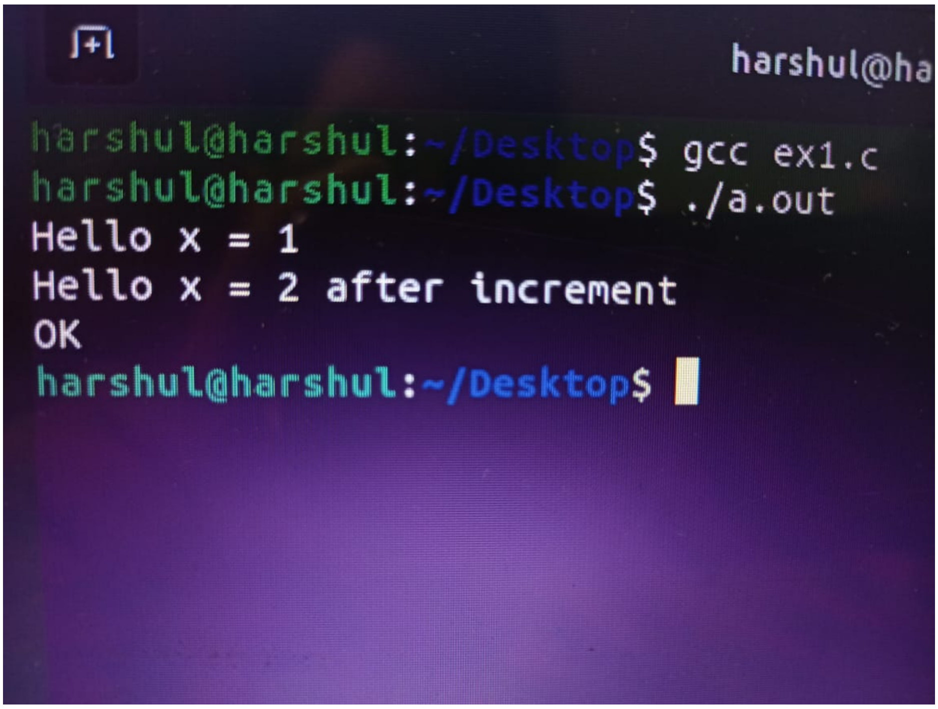
1: PC Boot Strap

```c
// 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
        __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");
        }
}
```



Added code:

__asm__("addl %%ebx, %%eax;" : "=a" (x) : "a" (x), "b" (1))

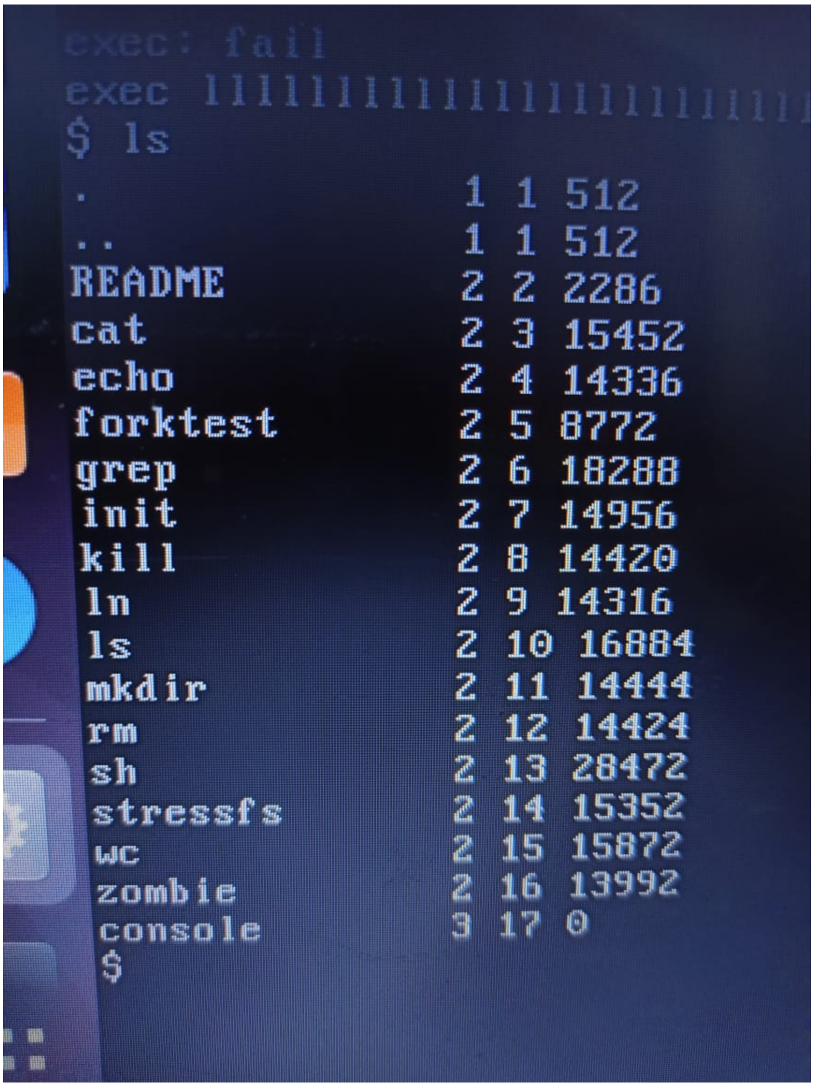
Here,

- input operands: x, 1
- output operands: x
- code: increments the value of x by 1

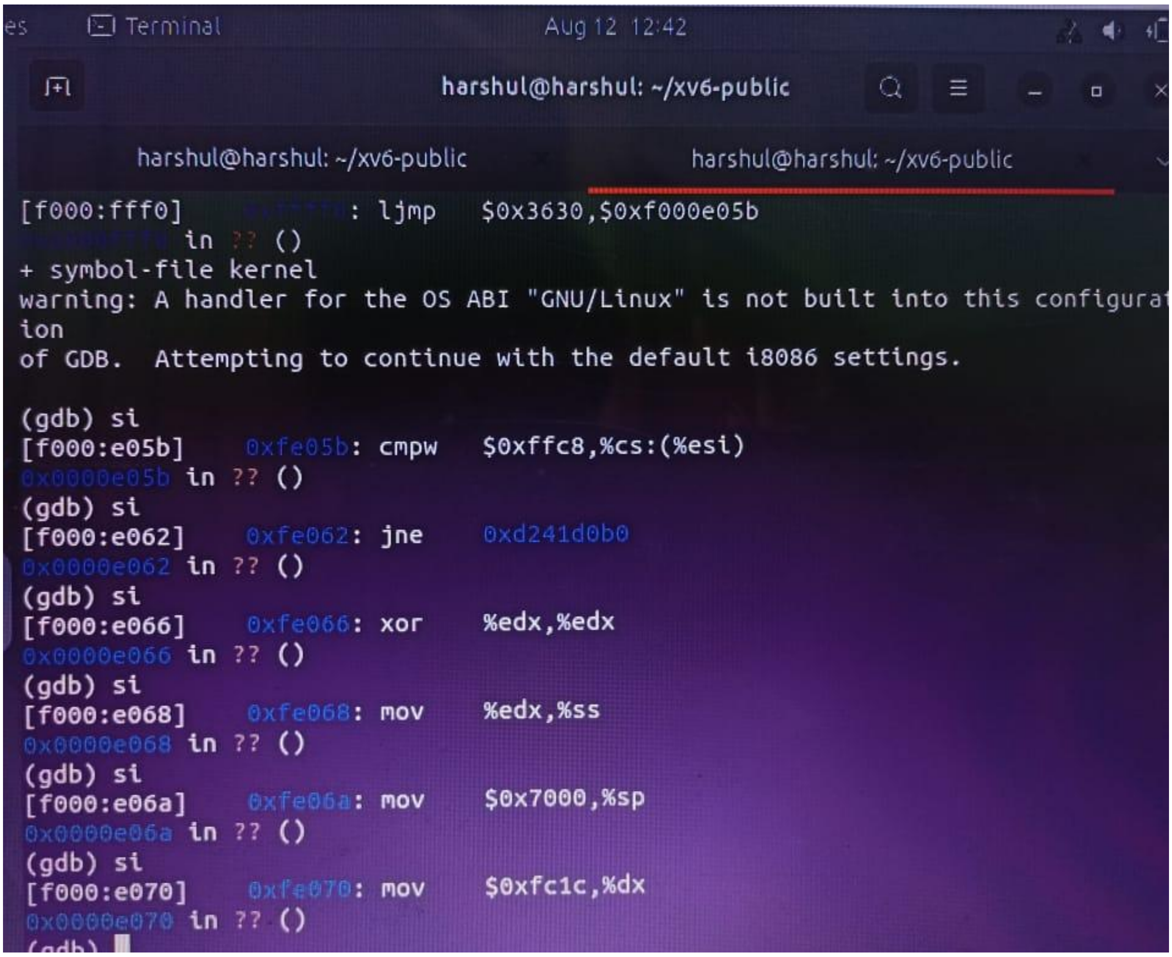
Installing GNU,QEMU and cloning xv6.

$ sudo apt-get update
$ sudo apt-get install build-essential
$ sudo apt-get install qemu
$ git clone git://github.com/mit-pdos/xv6-public.git
$ make qemu
$ sudo apt install qemu-kvm libvirt-daemon-system libvirt-clients bridge-utils virtinst virt-manager

$ ls



2.



"si" instruction in gdb - executes one machine instruction (follows a call).

Screenshot - shows the first 6 instructions of the xv6 operating system. The first instruction is

[f000:fff0] 0xffff0: ljmp $0x3630,$0xf000e05b

Here,

- f000 - starting ode segment
- fff0 - starting instruction pointer
- 0xffff0 - physical address where the instruction resides
- ljmp - instruction
- 0x3630 - destination code segment
- 0xf000e05b - Destination Instruction Pointer.

cmp instruction - used to perform comparison. It's identical to the sub instruction except it does not affect operands.

jnz (or **jne**) instruction - a conditional jump that follows a test. It jumps to the specified location if the Zero Flag (ZF) is cleared (0). jnz is commonly used to explicitly test for something not being equal to zero whereas jne is commonly found after a cmp instruction.

xor instruction - performs a logical XOR (exclusive OR) operation. This is the equivalent to the "^" operator in c++.

mov instruction - **moves data bytes between the two specified operands**. The byte specified by the second operand is copied to the location specified by the first operand. The source data byte is not affected.

Part 2: The Boot Loader

3.

Loop that reads the sectors of kernel from the disk- code is given below:

```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 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>
```

first instruction - first operation on entering the for loop is comparison between the values of ph and eph because the loop will run only when ph < eph.

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)
```

```
  # 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
```

(a) By analyzing the contents of bootasm.S, conclusion - "movw $(SEG_KDATA<<3), %ax" is the first instruction to be executed in 32-bit mode. "ljmp $(SEG_KCODE<<3), $start32" instruction completes the transition to 32-bit protected mode.

(b) By analyzing the contents of bootasm.S, bootmain.c and bootblock.asm, conclusion - that bootasm.S switches the OS into 32-bit mode and then calls bootmain.c which first loads the kernel using ELF header and the enters the kernel using entry(). Hence the last instruction of the bootloader is entry(). Looking for the same in bootblock.asm, we find out the instruction to be

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

which is a call instruction which shifts control to the address stored at 0x10018 since **dereferencing operator** (*) has been used. Now we need to know the starting address of the kernel. We can find this by two methods:

(i) By looking at the first word of memory stored at 0x10018 (by using the command **"x/1x 0x10018"**)
(ii) By looking at the contents of "**objdump -f kernel"**

After getting the starting address of kernel, we need to see what is the instruction stored at that address to get the first instruction of kernel. We can do this by two methods:

(i) By using **"x/1i 0x0010000c"**
(ii) By looking into kernel.asm

```
(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) x/1x 0x10018
0x10018:        0x0010000c
(gdb) x/1i 0x0010000c
   0x10000c:    mov    %cr4,%eax
```

Hence, the first instruction of kernel is

```
0x10000c:       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);
}
```

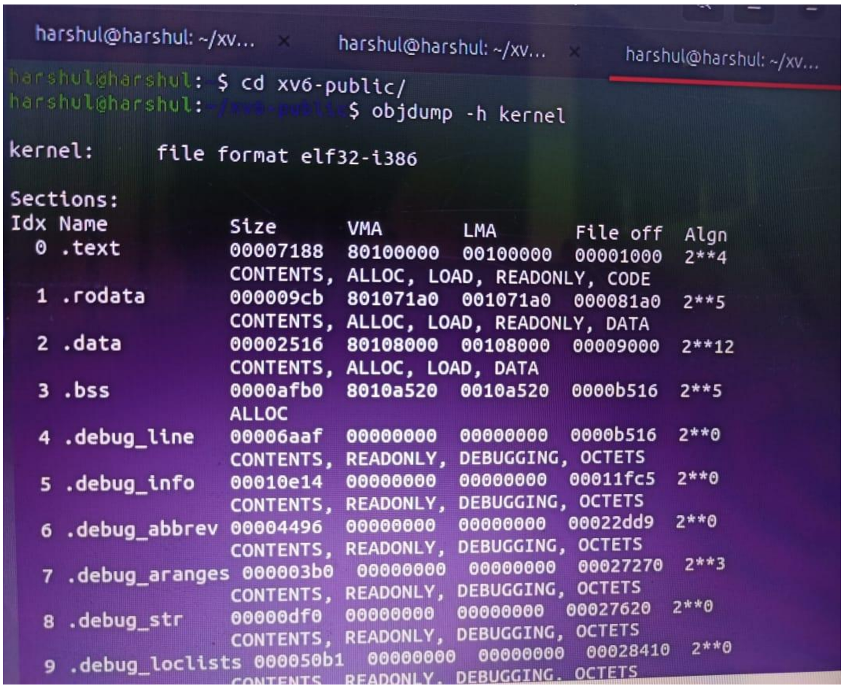
The above lines of code are present in bootmain.c. This is the code that is used by xv6 to load the kernel.

xv6 performs following actions:

- Loads ELF headers of kernel into a memory location pointed to by “elf”.
- Stores 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).
- Maintains an end pointer eph which points to the memory location after the end of the last segment.
- Iterates over all the segments. For every segment, pa points to the address at which this segment has to be loaded.
- Loads the current segment at that location by passing pa, ph->filesz and ph->off parameters to readseg. It then checks the memory assigned to this sector is greater than the data copied. If

this is true, it initializes the extra memory with zeros.Coming back to the question, 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 stores in the ELF header helps the boot loader to decide how many sectors it has to read.

4.



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

```
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      00000026  00000000  00000000  000002e8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000310  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   00000585  00000000  00000000  00000350  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 0000023c  00000000  00000000  000008d5  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_line   00000283  00000000  00000000  00000b11  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_str    00000204  00000000  00000000  00000d94  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_line_str 0000003f  00000000  00000000  00000f98  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 0000018d  00000000  00000000  00000fd7  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000033  00000000  00000000  00001164  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

As we can see in the above screenshot, VMA and LMA of text section are the same , indicating that it loads and executes from the same address.

5.

I have changed the link address from 0x7c00 to 0x7c08. As no changed has been done to the BIOS,it will run simultaneoulsy for both versions and hand over the control to the boot loader.From this points,we have to check the difference between two files.I run ‘si’ command around 200 times instructions and compared the output of two files.

In the first command the difference was spotted attached below along with the next 3 instructions.

In the 1st pic. When the link address was corresctly set to 0x7c00.

In the 2nd pic. It changed to 0x7c08.

I have attached the output files of GDB and also “objdump -h bootmain.o” for both the version because output differ due to changed in link address.

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

```
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c39
0x00007c2c in ?? ()
(gdb)
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb)
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb)
[f000:d0b0]    0xfd0b0: cli
```

```
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31
0x00007c2c in ?? ()
(gdb)
The target architecture is assumed to be i386
=> 0x7c31:  mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb)
=> 0x7c35:  mov    %eax,%ds
0x00007c35 in ?? ()
(gdb)
=> 0x7c37:  mov    %eax,%es
```

6.

For this experiment, we have to examine the 8 words of memory at 0x00100000 at two different instances of time, the first when the BIOS enters the boot loader and the second when the boot loader enters the kernel.

a) For this, we will use the command **“x/8x 0x00100000”** but before that we will have to set our breakpoints. The first breakpoint will be at 0x7c00 because this is the point where the BIOS hands control over to the boot loader.
b) The second breakpoint will be at 0x0010000c because this is the point when the kernel is passed control by the bootloader.

```
(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      0xa000b8e0      0x220f0010      0xc0200fd8
```

As we can see in the diagram, we get different values at both the breakpoints. The explanation to this is as follows:

The address 0x00100000 is actually 1MB which 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). By default, all the uninitialized values are set to 0 in xv6.

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

Adding a System Call

1.

An operating system supports two modes; the kernel mode and the user mode. When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that particular resource. This is done via a **system call**. When a program makes a system call, the mode is switched from user mode to kernel mode.

In order to define our own system call in xv6, changes need to be made to 5 files. Namely, these files are as follows.

(i) syscall.h
(ii) syscall.c
(iii) sysproc.c
(iv) usys.S
(v) user.h