OS 344 - ASSIGNMENT 1
G20

-Anjali Godara -180101008
-Niharika Bhamer-180101048
-Tanmay Jain-180123050
-Varhade Amey Anant-180101087

EX.1) //Added the following line to the code for incrementing the value of x

- asm("incl %0": "+r"(x));

```c
#include <stdio.h>
int main(int argc, char **argv)
{
int x = 1;
printf("Hello x = %d\n", x);
asm("incl %0": "+r"(x));
printf("Hello x = %d after increment\n", x);
if(x == 2){
printf("OK\n");
}
else{
printf("ERROR\n");
}
}
```

EX.2) The GDB's si (Step Instruction) command is used to execute one machine instruction, then stop and return to the debugger.

```
        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 configuratio
n
of GDB.  Attempting to continue with the default i8086 settings.

(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb)
[f000:e062]    0xfe062: jne    0xd241d416
0x0000e062 in ?? ()
(gdb)
[f000:e066]    0xfe066: xor    %edx,%edx
0x0000e066 in ?? ()
(gdb)
```

In the above Screenshot, we have the results of 3 si commands.

[f000:e05b]   0xfe05b: cmpw $0xffc8,%cs:(%esi)

[f000:e05b] - This represents the values of CS and IP [CS:IP].

0xfe05b - Represents the physical address of the place in memory where this instruction is stored. Physical address can be calculated using the formula:

physical address = 16 * segment + offset.

cmpw $0xffc8,%cs:(%esi) - Inline assembly instruction where *cmpw* compares two words.

EX.3)

```
void
readsect(void *dst, uint offset)
{
  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
  waitdisk();
  insl(0x1F0, dst, SECTSIZE/4);
}
```

- **waitdisk();**
  7c98:   e8 e1 ff ff ff   call   7c7e <waitdisk>
- **outb(0x1F2, 1);   // count = 1**
- **outb(0x1F3, offset)**
- **outb(0x1F4, offset >> 8);**
  7cb0:   89 d8   mov   %ebx,%eax
  7cb2:   c1 e8 08   shr $0x8,%eax
  7cb5:   ba f4 01 00 00   mov $0x1f4,%edx
  7cba: ee   out   %al,(%dx)
- **outb(0x1F5, offset >> 16);**
  7cbb: 89 d8   mov   %ebx,%eax
  7cbd: c1 e8 10   shr $0x10,%eax
  7cc0:   ba f5 01 00 00   mov $0x1f5,%edx
  7cc5:   ee   out   %al,(%dx)
- **outb(0x1F6, (offset >> 24) | 0xE0);**
  7cc6:   89 d8   mov   %ebx,%eax
  7cc8:   c1 e8 18   shr $0x18,%eax
  7ccb: 83 c8 e0   or $0xffffffe0,%eax
  7cce: ba f6 01 00 00   mov $0x1f6,%edx
  7cd3:   ee   out   %al,(%dx)
  7cd4:   b8 20 00 00 00   mov $0x20,%eax
  7cd9:   ba f7 01 00 00   mov $0x1f7,%edx
  7cde: ee   out   %al,(%dx)
- **outb(0x1F7, 0x20); // cmd 0x20 - read sectors**

- **waitdisk();**
  7cdf: e8 9a ff ff ff call 7c7e <waitdisk>
- **insl(0x1F0, dst, SECTSIZE/4);**

Code for the “for loop” :

```
 for(; ph < eph; ph++){
   7d83:       39 f3                   cmp    %esi,%ebx
   7d85:       72 0f                   jb     7d96 <bootmain+0x5b>
  entry();
   7d87:       ff 15 18 00 01 00       call   *0x10018
   7d8d:       eb d5                   jmp    7d64 <bootmain+0x29>
  for(; ph < eph; ph++){
   7d8f:       83 c3 20                add    $0x20,%ebx
   7d92:       39 de                   cmp    %ebx,%esi
   7d94:       76 f1                   jbe    7d87 <bootmain+0x4c>
```

Start line of the for loop : 7d83: 39 f3 cmp %esi,%ebx
End line of the for loop : 7d94: 76 f1 jbe 7d87 <bootmain+0x4c>

```
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 runs the loop.

entry();
7d87: ff 15 18 00 01 00 call *0x10018

The above instruction is run after the loop is completed.

```
0x0000fff0 in ?? ()
+ symbol-file kernel
warning: A handler for the OS ABI "GNU/Linux" is not built into this configurati
on
of GDB.  Attempting to continue with the default i8086 settings.

(gdb) b *0x7d87
Breakpoint 1 at 0x7d87
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d87:      call   *0x10018

Thread 1 hit Breakpoint 1, 0x00007d87 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 ?? ()
```

Set the breakpoint at 0x7d87 and then step into the next few instructions by si command.

A) ljmp $(SEG_KCODE<<3), $start32, this is where the transition of the processor to 32-bit protected mode is completed.

```
# 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+0xe>
  movl    %cr0, %eax
    7c22:   0f 20 c0                mov    %cr0,%eax
  orl     $CR0_PE, %eax
    7c25:   66 83 c8 01             or     $0x1,%ax
  movl    %eax, %cr0
    7c29:   0f 22 c0                mov    %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
```

movw $(SEG_KDATA<<3), %ax

From the above instruction the processor starts executing 32-bit code.

```
 # 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
    7c34:   ea                      .byte 0xea
    7c35:   39 7c 08 00             cmp    %edi,0x0(%eax,%ecx,1)

00007c39 <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
    7c39:   66 b8 10 00             mov    $0x10,%ax
```

B) The last instruction of the boot loader executed :

In bootmain.c:

entry = **(void(*)(void))(elf->entry)**;

entry();

In bootblock.asm:

7d87: ff 15 18 00 01 00 call *0x10018

The first instruction executed of the kernel it just loaded is present at *0x10018.

At *0x10018 , using gdb we found the address 0x10000c.Hence the first instruction is:

0x10000c: mov %cr4,%eax

```
Terminal
File  Edit  View  Search  Terminal  Help
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 configuratio
n
of GDB.  Attempting to continue with the default i8086 settings.

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) b *0x7d87
Breakpoint 2 at 0x7d87
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d87:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d87 in ?? ()
(gdb) x/1x 0x10018
0x10018:        0x0010000c
(gdb) x/1i 0x0010000c
   0x10000c:    mov    %cr4,%eax
(gdb) 
```

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);
  }
```

In bootmain.c we have the above code segment which helps the bootloader to fetch the entire kernel from the hard disk.elf is a pointer of type struct elfhdr and points to address 0x10000.ph and eph are both pointers of type struct proghdr. ph points to the first program header's sector's address whereas eph is pointing to the point where all program headers end.This information is provided in the elf->phnum attribute.

In the for loop in each iteration we are storing the physical address of the place in memory where a program header resides and we read this into pa using readseg() function. We do so until ph is less than eph, as we do not want to exceed above the point where last program header is stored.Hence until the condition ph<eph is satisfied, bootloader keeps reading the memory sectors.If at all more memory than the file size is read, we set the extra memory to 0 in the last if condition used.

EX.4) Using the **$ objdump -h** command.

$ objdump -h kernel:

```
anjali@anjali-ThinkPad-T480:~/xv6-public$ objdump -h kernel

kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00006f52  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00001040  80106f60  00106f60  00007f60  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   00002600  00000000  00000000  0000b516  2**0
                  CONTENTS, READONLY, DEBUGGING
```

In the above image we can see the various other sections present in the kernel. VMA(link address) and LMA(load address) of the kernel are different, i.e it is loaded at a different memory location whereas it executes from a different memory location.

$ objdump -h bootblock.o:

```
anjali@anjali-ThinkPad-T480:~/xv6-public$ objdump -h bootblock.o

bootblock.o:     file format elf32-i386

Sections:
Idx Name           Size      VMA       LMA       File off  Algn
  0 .text          000001c0  00007c00  00007c00  00000074  2**2
                   CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame      000000bc  00007dc0  00007dc0  00000234  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment       00000029  00000000  00000000  000002f0  2**0
                   CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000320  2**3
                   CONTENTS, READONLY, DEBUGGING
  4 .debug_info    0000050b  00000000  00000000  00000360  2**0
                   CONTENTS, READONLY, DEBUGGING
```

In the above image we can see that the VMA and LMA are same,i.e it is loaded and executed from the same locations in memory.

EX.5) In the original makefile, tracing through the instructions it can be observed that when the bootloader enters the kernel i.e at the ljmp instruction , the transition to 32-bit protected mode is completed and we see a message:

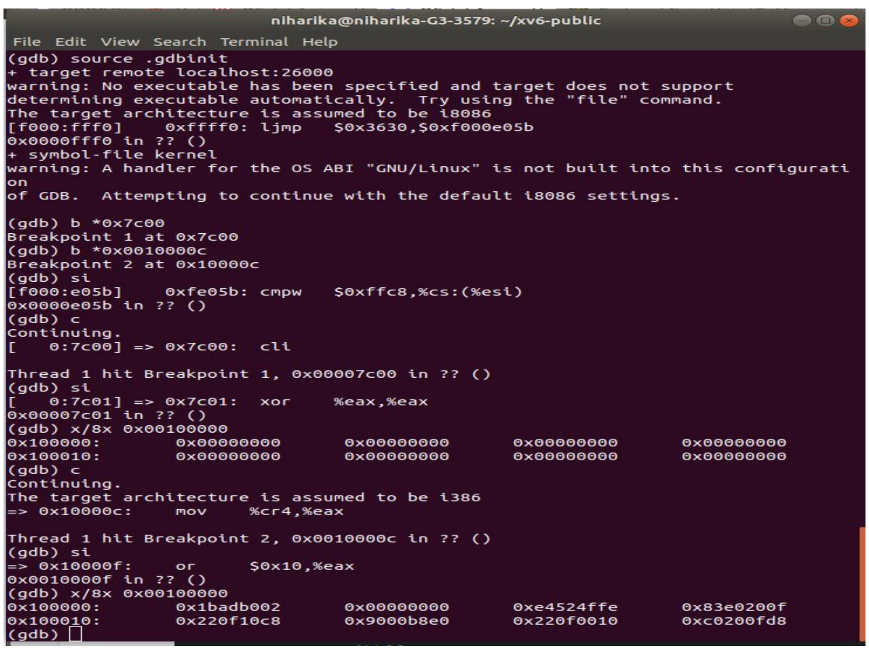
“Target architecture is assumed to be i386” .

```
(gdb)
[   0:7c25] => 0x7c25:  or     $0x1,%ax
0x00007c25 in ?? ()
(gdb)
[   0:7c29] => 0x7c29:  mov    %eax,%cr0
0x00007c29 in ?? ()
(gdb)
[   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
0x00007c37 in ?? ()
(gdb)
=> 0x7c39:      mov    %eax,%ss
0x00007c39 in ?? ()
(gdb)
```

But upon changing the link address in the makefile to **-Ttext 0x7C08**, we can see that after the ljmp instruction , instead of transitioning into 32-bit protected mode, we are reverted back to the BIOS and we can see the same as the next instruction is at the start of the BIOS [f000:e05b].

```
(gdb)
[   0:7c1d] => 0x7c1d:  lgdtl  (%esi)
0x00007c1d in ?? ()
(gdb)
[   0:7c22] => 0x7c22:  mov    %cr0,%eax
0x00007c22 in ?? ()
(gdb)
[   0:7c25] => 0x7c25:  or     $0x1,%ax
0x00007c25 in ?? ()
(gdb)
[   0:7c29] => 0x7c29:  mov    %eax,%cr0
0x00007c29 in ?? ()
(gdb)
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c39
0x00007c2c in ?? ()
(gdb)
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb)
[f000:e062]    0xfe062: jne    0xd241d416
0x0000e062 in ?? ()
```

EX.6)The two outputs of examining memory words at 0x00100000 are different because at the first breakpoint i.e 0x7c00 the BIOS enters the boot loader and hence the kernel has not been yet loaded, so we have all zeroes.
At the second breakpoint i.e 0x001000c the boot loader enters the kernel and the kernel has been loaded. So the output this time is different and is shown in the image below.


```
niharika@niharika-G3-3579: ~/xv6-public
File Edit View Search Terminal Help
(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 configurati
on
of GDB.  Attempting to continue with the default i8086 settings.

(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) b *0x0010000c
Breakpoint 2 at 0x10000c
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(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) x/8x 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    mov    %cr4,%eax

Thread 1 hit Breakpoint 2, 0x0010000c in ?? ()
(gdb) si
=> 0x10000f:    or     $0x10,%eax
0x0010000f in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb)
```

EX.7&8) To add a custom system call, we need to make changes to 5 files:

1. Syscall.h
2. Syscall.c
3. Sysproc.c
4. Usys.s
5. user.h

We would start the procedure by editing ***syscall.h*** in which a number is given to every system call. This file already contains 21 system calls. In order to add the custom system call, the following line needs to be added to this file.

#define SYS_wolfie 22

Next, we need to add a pointer to the system call in the ***syscall.c*** file. This file contains an array of function pointers which uses the above-defined numbers (indexes) as pointers to system calls which are defined in a different location.

[SYS_wolfie] sys_wolfie,

Next we need to add the function prototype in ***syscall.c***,

extern int sys_wolfie(void);

Then we need to implement the system call function in ***sysproc.c***. The file has been attached in the zip folder.

In order for a user program to call the system call, we added the following interface in the ***usys.S*** file

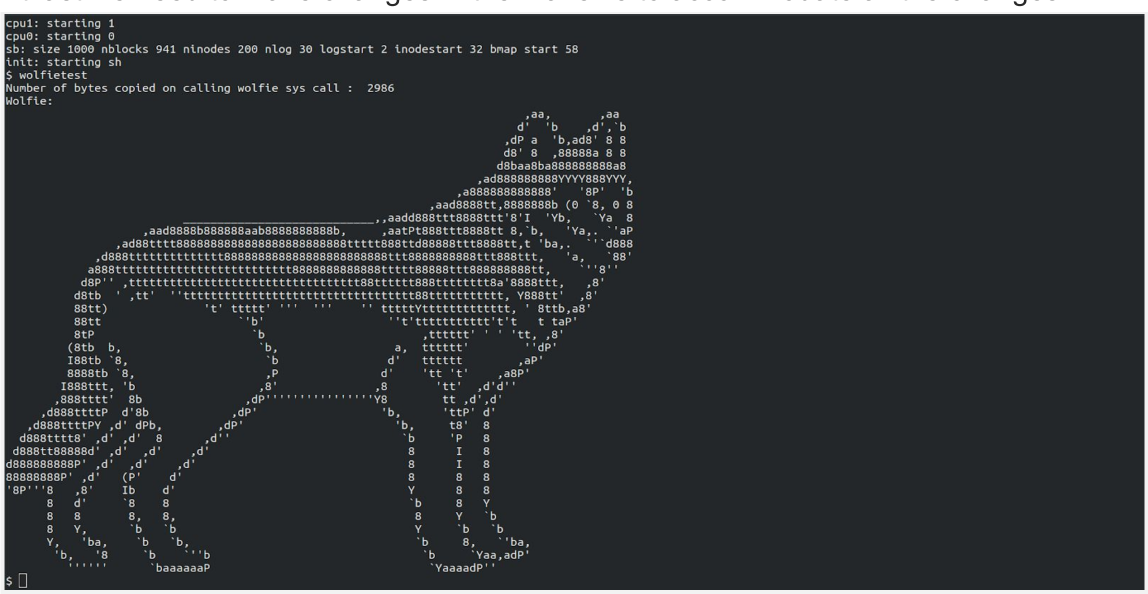
SYSCALL(wolfie)

Then we need to add the function declaration in ***user.h*** file

Int wolfie(void*, uint);

In order to test the functionality of this, we added a user program named ***wolfietest.c*** which calls this system call.

At last we need to make changes in the makefile to accommodate all the changes.



Wolfietest.c code

```c
// User program to call the system call wolfie.
#include "types.h"
#include "stat.h"
#include "user.h"
int main(void){
    char buf[3500];
    printf(1,"Number of bytes copied on calling wolfie sys call :  %d\n",wolfie((void*) buf,3500));
    printf(1,"%s",buf);
    exit();
}
```