CS1217 - Spring 2023 - Homework 3

Bhumika Mittal, Saptarishi Dhanuka

Contributions of individual team members:
Bhumika: Boot xv6, Exercise 0 and Exercise 1
Saptarishi: Boot xv6, Exercise 0, Exercise 2
PS - In general, we sat together for most of the assignment and worked on it.

BOOT xv6

As per the given instructions, we get the following output after setting the breakpoint:

```
For help, type "help".
Type "apropos word" to search for commands related to "word".
+ 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) br * 0x0010000c
Breakpoint 1 at 0x10000c
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x10000c:    mov    %cr4,%eax

Thread 1 hit Breakpoint 1, 0x0010000c in ?? ()
(gdb) 
```

Exercise 0

```
(gdb) info reg
eax            0x0                 0
ecx            0x0                 0
edx            0x1f0               496
ebx            0x10094             65684
esp            0x7bdc              0x7bdc
ebp            0x7bf8              0x7bf8
esi            0x10094             65684
edi            0x0                 0
eip            0x10000c            0x10000c
eflags         0x46                [ IOPL=0 ZF PF ]
cs             0x8                 8
ss             0x10                16
ds             0x10                16
es             0x10                16
fs             0x0                 0
gs             0x0                 0
fs_base        0x0                 0
gs_base        0x0                 0
k_gs_base      0x0                 0
cr0            0x11                [ ET PE ]
cr2            0x0                 0
cr3            0x0                 [ PDBR=0 PCID=0 ]
cr4            0x0                 [ ]
cr8            0x0                 0
efer           0x0                 [ ]
```

```
(gdb) x/24x $esp
0x7bdc: 0x00007d87      0x00000000      0x00000000      0x00000000
0x7bec: 0x00000000      0x00000000      0x00000000      0x00000000
0x7bfc: 0x00007c4d      0x8ec031fa      0x8ec08ed8      0xa864e4d0
0x7c0c: 0xb0fa7502      0xe464e6d1      0x7502a864      0xe6dfb0fa
0x7c1c: 0x16010f60      0x200f7c78      0xc88366c0      0xc0220f01
0x7c2c: 0x087c31ea      0x10b86600      0x8ed88e00      0x66d08ec0
(gdb)
```

The first two lines of the printout is actually the stack i.e.

0x7bdc: 0x00007d87 0x00000000 0x00000000 0x00000000
0x7bec: 0x00000000 0x00000000 0x00000000 0x00000000

This is because the stack pointer (esp) has 0x7bdc and the frame pointer (ebp) has 0x7bf8 and the stack lies between these values. When we see the first 24 values from stack pointer onwards, the third line starts from 0x7bfc which is ahead of the ebp value, hence only the first two lines are actually the stack.

The only non-zero value on the stack is 0x00007d87 which we can see from line 323 of bootblock.asm, is the place where the following call occurs:

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

Using gdb to set a breakpoint at 0x00007d87, pressing "c" to continue, executing info reg, then single stepping, then info reg again we see the eip value has been changed to `0x10000c`.

```
(gdb) br * 0x00007d87
Breakpoint 1 at 0x7d87
(gdb) c
Continuing.
The target architecture is assumed to be
=> 0x7d87:      call   *0x10018

Thread 1 hit Breakpoint 1, 0x00007d87 in
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x1f0    496
ebx            0x10094  65684
esp            0x7be0   0x7be0
ebp            0x7bf8   0x7bf8
esi            0x10094  65684
edi            0x0      0
eip            0x7d87   0x7d87
eflags         0x46     [ PF ZF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x0      0
gs             0x0      0
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x1f0    496
ebx            0x10094  65684
esp            0x7bdc   0x7bdc
ebp            0x7bf8   0x7bf8
esi            0x10094  65684
edi            0x0      0
eip            0x10000c 0x10000c
eflags         0x46     [ PF ZF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x0      0
gs             0x0      0
```

The stack gets changed to the following:

```
(gdb) x/24x $esp
0x7bdc: 0x00007d8d      0x00000000      0x00000000      0x00000000
0x7bec: 0x00000000      0x00000000      0x00000000      0x00000000
0x7bfc: 0x00007c4d      0x8ec031fa      0x8ec08ed8      0xa864e4d0
0x7c0c: 0xb0fa7502      0xe464e6d1      0x7502a864      0xe6dfb0fa
0x7c1c: 0x16010f60      0x200f7c78      0xc88366c0      0xc0220f01
0x7c2c: 0x087c31ea      0x10b86600      0x8ed88e00      0x66d08ec0
```

0x00007d8d is immediately after 7d87

Hence, this non-zero value on the stack is the address to which we need to return to after calling entry() in bootmain

The stack pointer is initialized at the line 65 (`bootasm.S`) which is **mov $0x7c00,%esp** in bootasm.S.

```
(gdb) si
=> 0x7c41:      mov    %eax,%gs
0x00007c41 in ?? ()
(gdb) si
=> 0x7c43:      mov    $0x7c00,%esp
0x00007c43 in ?? ()
(gdb) si
=> 0x7c48:      call   0x7d3d
0x00007c48 in ?? ()
(gdb)
```

The stack after the call to bootmain is as follows:

```
(gdb) si
=> 0x7c48:      call   0x7d3d
0x00007c48 in ?? ()
(gdb) x/24x $esp
0x7c00: 0x8ec031fa      0x8ec08ed8      0xa864e4d0      0xb0fa7502
0x7c10: 0xe464e6d1      0x7502a864      0xe6dfb0fa      0x16010f60
0x7c20: 0x200f7c78      0xc88366c0      0xc0220f01      0x087c31ea
0x7c30: 0x10b86600      0x8ed88e00      0x66d08ec0      0x8e0000b8
0x7c40: 0xbce88ee0      0x00007c00      0x0000f0e8      0x00b86600
0x7c50: 0xc289668a      0xb866ef66      0xef668ae0      0x9066feeb
(gdb)
```

The first assembly instructions of bootmain are

```
7d3d:   55          push   %ebp
7d3e:   89 e5       mov    %esp,%ebp
7d40:   57          push   %edi
7d41:   56          push   %esi
7d42:   53          push   %ebx
7d43:   83 ec 10    sub    $0x10,%esp
```

These instructions push values of the specific registers on the stack to save them (so that when the function returns, these values are preserved). It also copies the stack pointer to the base pointer. Also, it allocates the 16 bytes on the stack. Basically, these instructions are preparing the stack by adjusting the value of stack and frame pointer.

Exercise 1

Steps to print the name of the system call and the return value while booting xv6:

1. In the syscall.c file, we need to modify the function syscall such that it prints the name of the system call.
2. First, generate an array with the name of the system calls, as follows:

```
//generate a sysnames array to print the name of the system call
static const char* sysnames[] = {
 [SYS_date]    "date",
 [SYS_fork]    "fork",
 [SYS_exit]    "exit",
 [SYS_wait]    "wait",
 [SYS_pipe]    "pipe",
 [SYS_read]    "read",
 [SYS_kill]    "kill",
 [SYS_exec]    "exec",
 [SYS_fstat]   "fstat",
 [SYS_chdir]   "chdir",
 [SYS_dup]     "dup",
 [SYS_getpid]  "getpid",
 [SYS_sbrk]    "sbrk",
 [SYS_sleep]   "sleep",
 [SYS_uptime]  "uptime",
 [SYS_open]    "open",
 [SYS_write]   "write",
 [SYS_mknod]   "mknod",
 [SYS_unlink]  "unlink",
 [SYS_link]    "link",
 [SYS_mkdir]   "mkdir",
 [SYS_close]   "close",
};
```

3. We then use the following function to print the syscall along with the return value.

```
void printsyscall(int num){
   const char* name = sysnames[num];
   cprintf("%s -> %d\n",name , myproc()->tf->eax);
}
```

Here, we declare a variable name to denote the syscall name and then use the pointer to eax from tf from the myproc() function to get the return value of the syscall.

4. Then, call this printsyscall function in the syscall function (if it is a valid syscall) so that it prints the name of the syscall when xv6 boots up.

```
void
syscall(void)
{
 int num;
 struct proc *curproc = myproc();

 num = curproc->tf->eax;
 if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
   curproc->tf->eax = syscalls[num]();
     printsyscall(num);

 }
```

Output –

```
write -> 1
fork -> 2
exec -> 0
open -> 3
close -> 0
$write -> 1
 write -> 1
```

Exercise 2

We had to modify the following files in the process of creating the date system call.
We first used the command grep -n uptime *.[chS] which enabled us to find all the pieces of code that corresponded to the uptime syscall and simply replicated the code for date.

syscall.c
[SYS_date] “date”
As per the first exercise, we created an array of syscalls that needed to be printed when booting xv6, so the date syscall needs to be added here as well.

syscall.c
extern int sys_date(void)
Since the actual content of the sys_date function will be in another file (sysproc.c), we need to add a function prototype to the syscall.c file, similar to all the other syscalls

syscall.c
[SYS_date] sys_date
This adds a function pointer corresponding to the date syscall to the array “static int (*syscalls[])(void)” which contains function pointers to all syscall functions already present in syscall.c

syscall.h
#define SYS_date 22
This defines the index of the function pointer array present in syscall.c that will correspond to the sys_date syscall. There were 21 syscalls already present so the sys_date syscall will have to reserve index 22 in the array. This acts as the syscall number.

sysproc.c
sys_date(void)
The actual syscall for date is implemented in this file using the cmostime function.
We declare a pointer to a struct rtcdate as “r”.

We use the argptr in a similar sense as the other syscalls have used argint. Since date syscall uses a pointer to a struct rtcdate, we use argptr function with appropriate arguments as per the function definition. The struct rtcdate * r is the first argument retrieved by using 0 in the argptr function call. We use void * since we have a struct rtcdate * while the argptr function uses char **. Then we pass the struct rtcdate * r to the function cmostime to get the current date and time.

user.h
int date(struct rtcdate *)
This is the function prototype that the user program “date.c” will call

usys.S
SYSCALL(date)
This acts like a macro and sets up an interface between the user program and the actual syscall. When the user uses the date() function in any user program, the system will map that to the syscall numbered 22 i.e. sys_date.

Makefile
We added _date to the UPROGS definition in Makefile in order to make the date program available to run from the xv6 shell

date.c
Then we created the date.c file which calls the date function as per the source given in the question document and prints it in a format.

Output –

```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ date
Date: 20-2-2023
Time: 9:57:33
$
```