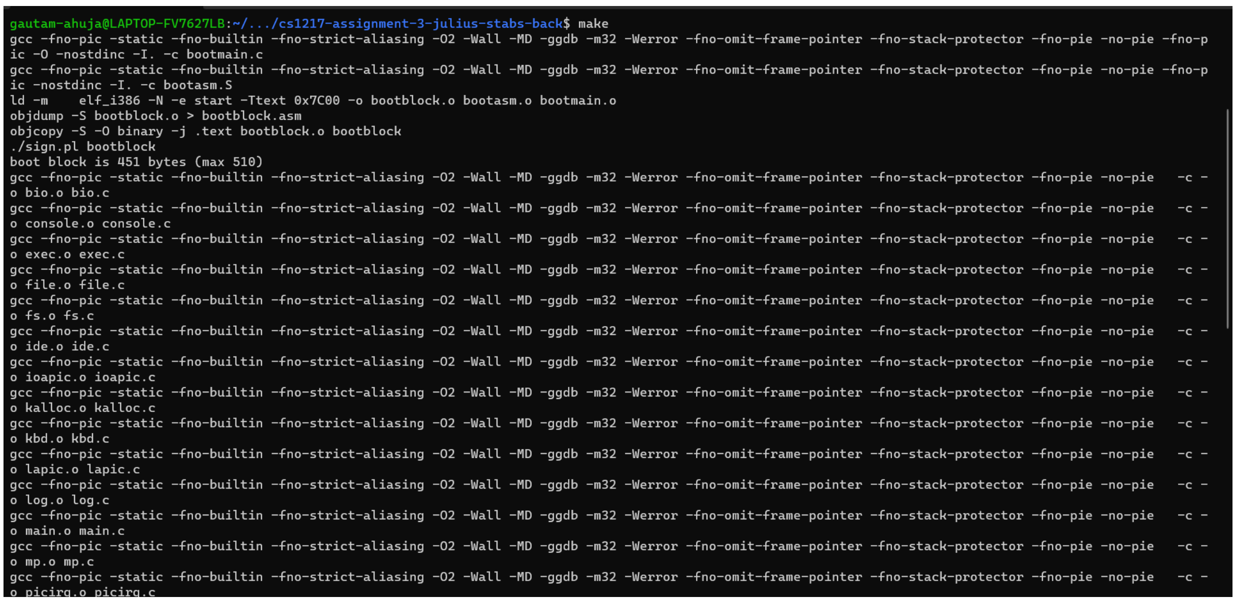
CS1217 - Spring 2023 - Homework 3

Gautam Ahuja, Nistha Singh

Note: All screenshots are on Gautam Ahuja's machine.

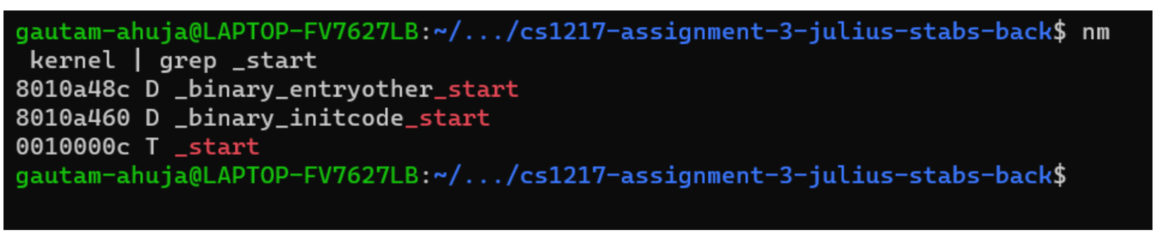
Boot xv6

Done by: Nistha Singh
To boot the `xv6` operating system, we follow the procedure explained in the assignment.
We run the `make` command on the terminal.



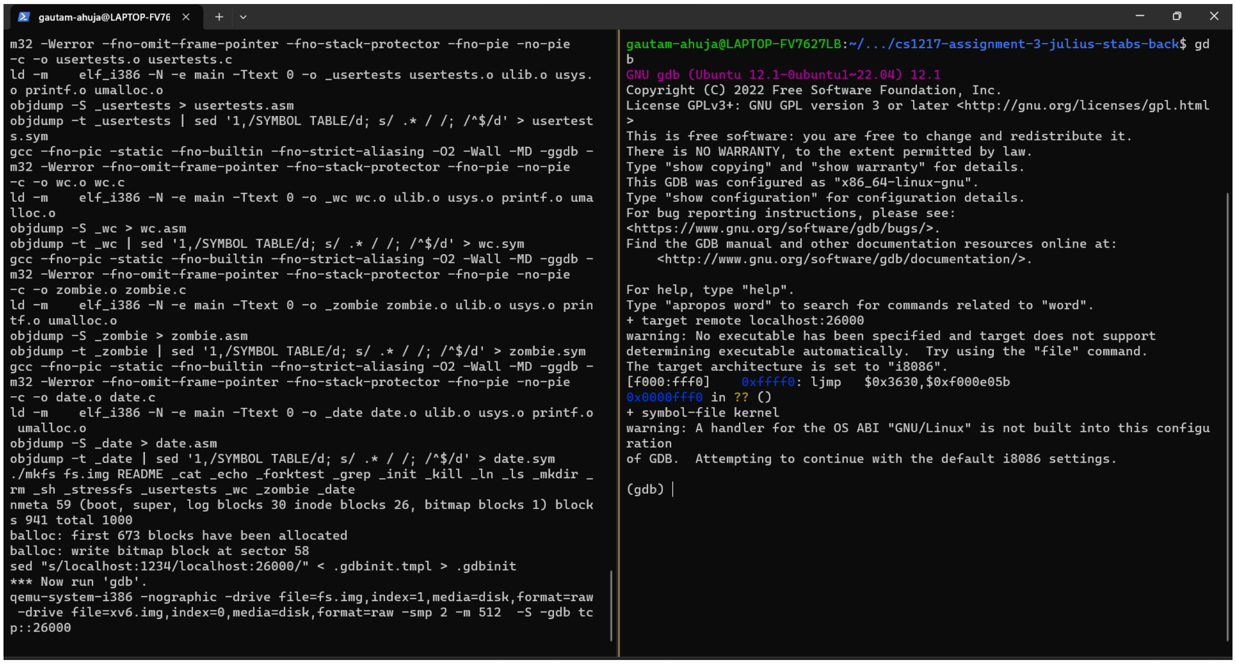
```
gautam-ahuja@LAPTOP-FV7627LB:~/.../cs1217-assignment-3-julius-stabs-back$ make
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -fno-p
ic -O -nostdinc -I. -c bootmain.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -fno-p
ic -nostdinc -I. -c bootasm.S
ld -m    elf_i386 -N -e start -Ttext 0x7C00 -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
boot block is 451 bytes (max 510)
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -
o bio.o bio.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -
o console.o console.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -
o exec.o exec.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -
o file.o file.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -
o fs.o fs.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -
o ide.o ide.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -
o ioapic.o ioapic.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -
o kalloc.o kalloc.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -
o kbd.o kbd.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -
o lapic.o lapic.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -
o log.o log.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -
o main.o main.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -
o mp.o mp.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -
o picirq.o picirq.c
```

We run the `nm kernel | grep _start` to access the starting address of the `kernel`.

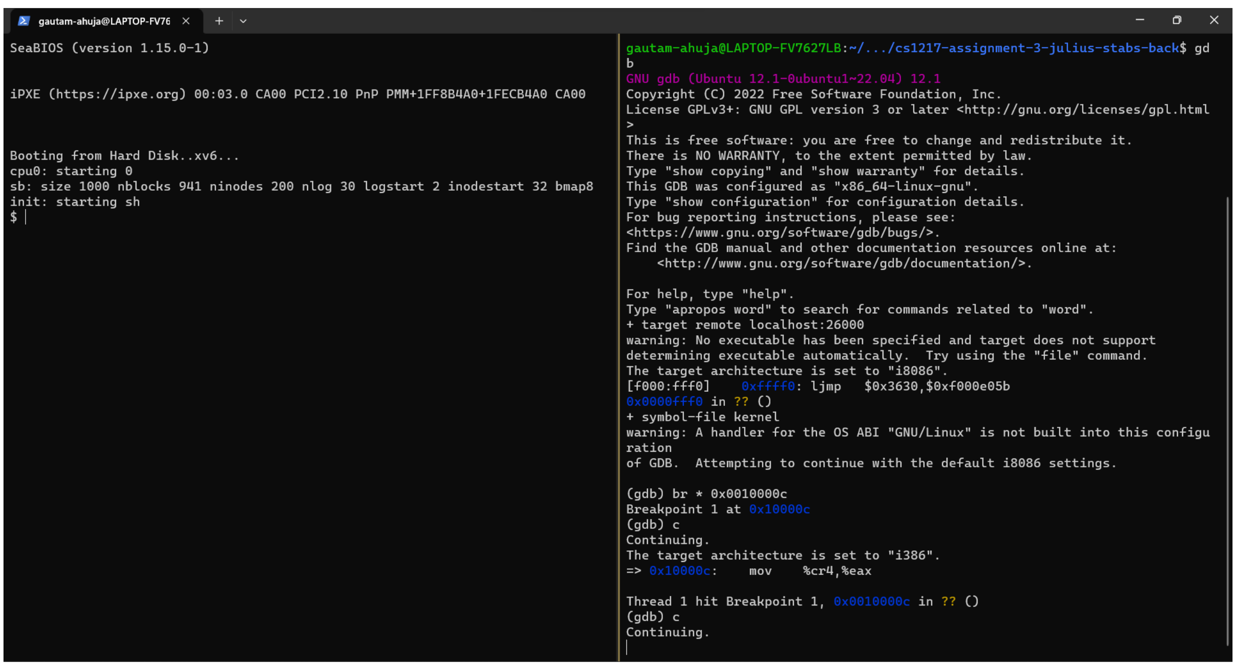


```
gautam-ahuja@LAPTOP-FV7627LB:~/.../cs1217-assignment-3-julius-stabs-back$ nm
 kernel | grep _start
8010a48c D _binary_entryother_start
8010a460 D _binary_initcode_start
0010000c T _start
gautam-ahuja@LAPTOP-FV7627LB:~/.../cs1217-assignment-3-julius-stabs-back$
```

Now we run two commands side by side: `make qemu-nox-gdb` and `gdd` on different terminals. This will link the `gdb` to `kernel`.



Now, we set the break point `br * 0x010000c` and continue to boot up the `xv6` operating system.



We have now booted up the `xv6` OS.

0. Exercise 0

Done by: Nistha Singh
`info reg` gives the content of all the registers while running.

```
Thread 1 hit Breakpoint 1, 0x0010000c in ?? ()
(gdb)
(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                 [ ]
xmm0           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm1           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm2           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm3           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm4           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm5           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
```

x/24x $esp gives the content of the stack.

```
(gdb)
(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)
(gdb) |
```

Solution.
Explaining the non-zero values on the stack:

a. 0x7bdc: This is the return address

b. 0x7bfc: This is saved registers, instructions

c. 0x7c0c: These have instructions

d. 0x7c1c: These have instructions

e. 0x7c2c: These have instructions

As seen above, The part of the stack printout that is actually the stack is from 0x7bdc to 0x7c2c (non-zero values). Since, it is the stack printout because it is not empty and contains data that has been pushed onto the stack during program execution. The other parts of the printout are mostly empty, because they have not yet been used by the program.

Explaining the contents of the stack:

1. In bootasm.S, the stack pointer is initialized in the _start function.

This initialization is done by the instruction `movl %esp, %ebp.` This instruction copies the value of the stack pointer `%esp` to the base pointer register, `%ebp`, which we are using to access function arguments and local variables.

2. For single step call to bootmain, we find its starting address through objdump.

```
gautam-ahuja@LAPTOP-FV7627LB:~/.../cs1217-assignment-3-julius-stabs-back$ objdump -d bootblock.o

bootblock.o:     file format elf32-i386


Disassembly of section .text:

00007c00 <start>:
    7c00:       fa                      cli
    7c01:       31 c0                   xor    %eax,%eax
    7c03:       8e d8                   mov    %eax,%ds
    7c05:       8e c0                   mov    %eax,%es
    7c07:       8e d0                   mov    %eax,%ss

00007c09 <seta20.1>:
    7c09:       e4 64                   in     $0x64,%al
    7c0b:       a8 02                   test   $0x2,%al
    7c0d:       75 fa                   jne    7c09 <seta20.1>
    7c0f:       b0 d1                   mov    $0xd1,%al
    7c11:       e6 64                   out    %al,$0x64
```

After single step through the call to bootmain, the stack will now have the return address and the arguments to call bootmain. The layout of the stack will look like this:

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb)  x/24x $esp
0x6f00: 0xf000d009      0x00000000      0x00006f5e      0x00008148
0x6f10: 0x00008189      0x00000000      0x00000000      0x00008148
0x6f20: 0x00006f5e      0x00007cc5      0x00000000      0x00000000
0x6f30: 0x00006f5e      0x00000000      0x00006210      0x00007c00
0x6f40: 0x00000080      0x00000000      0x00000000      0x000ee5bf
0x6f50: 0x000f1d3c      0x00000000      0x00007c00      0x00000000
(gdb) si
[   0:7c01] => 0x7c01:  xor    %eax,%eax
0x00007c01 in ?? ()
(gdb) si
[   0:7c03] => 0x7c03:  mov    %eax,%ds
0x00007c03 in ?? ()
(gdb) si
[   0:7c05] => 0x7c05:  mov    %eax,%es
0x00007c05 in ?? ()
(gdb) si
[   0:7c07] => 0x7c07:  mov    %eax,%ss
```

```
0x7c00: 0x8ec031fa      0x8ec08ed8      0xa864e4d0      0xb0fa7502
0x7c10: 0xe464e6d1      0x7502a864      0xe6dfb0fa      0x16010f60
(gdb)
0x7c20: 0x200f7c78      0xc88366c0      0xc0220f01      0x087c31ea
0x7c30: 0x10b86600      0x8ed88e00      0x66d08ec0      0x8e0000b8
0x7c40: 0xbce88ee0      0x00007c00      0x0000f0e8      0x00b86600
0x7c50: 0xc289668a      0xb866ef66      0xef668ae0      0x9066feeb
0x7c60: 0x00000000      0x00000000      0x0000ffff      0x00cf9a00
0x7c70: 0x0000ffff      0x00cf9200      0x7c600017      0xf7ba0000
(gdb)
0x7c80: 0xec000001      0x3cc0e083      0xc3f87540      0x57e58955
0x7c90: 0x0c5d8b53      0xffffe5e8      0x0001b8ff      0xf2ba0000
0x7ca0: 0xee000001      0x0001f3ba      0xeed88900      0xe8c1d889
0x7cb0: 0x01f4ba08      0x89ee0000      0x10e8c1d8      0x0001f5ba
0x7cc0: 0xd889ee00      0x8318e8c1      0xf6bae0c8      0xee000001
0x7cd0: 0x000020b8      0x01f7ba00      0xe8ee0000      0xffffff9e
```

```
0x7be0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7bf0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7c00: 0x8ec031fa      0x8ec08ed8      0xa864e4d0      0xb0fa7502
0x7c10: 0xe464e6d1      0x7502a864      0xe6dfb0fa      0x16010f60
(gdb)
0x7c20: 0x200f7c78      0xc88366c0      0xc0220f01      0x087c31ea
0x7c30: 0x10b86600      0x8ed88e00      0x66d08ec0      0x8e0000b8
0x7c40: 0xbce88ee0      0x00007c00      0x0000f0e8      0x00b86600
0x7c50: 0xc289668a      0xb866ef66      0xef668ae0      0x9066feeb
0x7c60: 0x00000000      0x00000000      0x0000ffff      0x00cf9a00
```

3. The first assembly instructions of the function perform stack setup using the following instructions:

```
bootblock.asm
// Read `count` bytes at `offset` from kernel into physical address `pa`.
// Might copy more than asked.
void
readseg(uchar* pa, uint count, uint offset)
{
    7cf4: 55                    push   %ebp
    7cf5: 89 e5                 mov    %esp,%ebp
    7cf7: 57                    push   %edi
    7cf8: 56                    push   %esi
    7cf9: 53                    push   %ebx
    7cfa: 83 ec 0c              sub    $0xc,%esp
    7cfd: 8b 5d 08              mov    0x8(%ebp),%ebx
    7d00: 8b 75 10              mov    0x10(%ebp),%esi
  uchar* epa;
```

a. `push %ebp`: This instruction pushes the current value of the base pointer (ebp) onto the stack. The current frame pointer is saved with this step.
b. `mov %esp, %ebp`: This instruction moves the current value of the stack pointer (ESP) into the base pointer (EBP) register.
c. `sub $0xc, %esp`: This instruction subtracts 8 from the stack pointer (esp) register. It save 8 bytes of space on the stack for local variables in the bootmain.

1. Exercise 1

Done by: Gautam Ahuja

Problem: To print out a line for each system call invocation with System Call name and return value.
Working:
As given in the question, we looked at `syscall()` function in `syscall.c`

```c
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]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

Here we see that a function `myproc()` is storing its results in another structure `tf (trapframe)` within the `proc` structure.
Looking at the definition, we can see that `proc` structure captures the current state of ongoing process.

```c
// Per-process state
github-classroom[bot], 5 days ago | 1 author (github-classroom[bot])
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  github-classroom[bot], 5 days ago
  struct proc *parent;         // Parent process
  github-classroom[bot], 5 days ago
  struct trapframe *tf;        // Trap frame for current syscall
  github-classroom[bot], 5 days ago
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  github-classroom[bot], 5 days ago
  struct file *ofile[NOFILE];  // Open files
  github-classroom[bot], 5 days ago
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

Wintin this, the `tf (trapframe)` structure holds the information of all the registers for a process (defined in `x86.h`).

```c
struct trapframe {      github-classroom[bot], 5 days ago • Initial commit
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

Here the `eax` (known as "accumulator") register holds the the return value of `myproc()`.

After that it checkes if the value is valid (> 0) and is within the range of defined number of syscalls and a that a syscall of `num` exists.

If result is true, it calls the particular function numbered (which are numbered in `syscall.h`) in a functional structure of `syscalls` defined above. The resulting return value is stored in `eax` register again of above defined `cpuproc` structure.

syscall.h

```c
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
```

Solution:

We defined a static array of pointers (string literals) which contains the syscall names corresponding to their defined number in `syscall.h` as follows:

```c
static const char *syscall_names[] = {
  [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",
  [SYS_date]   "date",
};
```

Then we just run a switch-case inside the `if` statement of `syscall()` which prints out the syscall name and its corresponding number currently in `eax` register as per currernt `num`.

```c
  // eax: tells us about the return value of the system call. It is a register in the trapframe
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
    // Edit Starts Here
    switch(num){
      case SYS_fork:
        cprintf("%s -> %d\n", syscall_names[num], curproc->tf->eax);
        break;
      case SYS_exit:
        cprintf("%s -> %d\n", syscall_names[num], curproc->tf->eax);
        break;
      case SYS_wait:
        cprintf("%s -> %d\n", syscall_names[num], curproc->tf->eax);
        break;
      case SYS_pipe:
        cprintf("%s -> %d\n", syscall_names[num], curproc->tf->eax);
        break;
      case SYS_read:
        cprintf("%s -> %d\n", syscall_names[num], curproc->tf->eax);
        break;
      case SYS_kill:
        cprintf("%s -> %d\n", syscall_names[num], curproc->tf->eax);
        break;
      case SYS_exec:
        cprintf("%s -> %d\n", syscall_names[num], curproc->tf->eax);
        break;
```

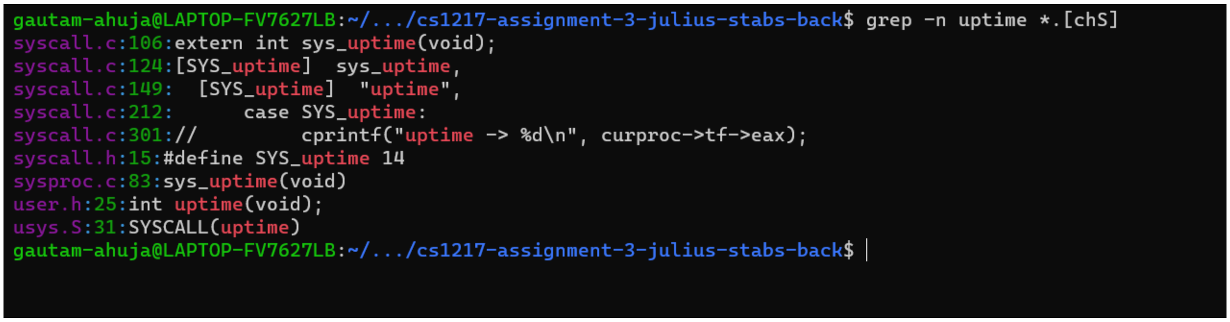
The result of above is:

```
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap8
exec -> 0
open -> 0
dup -> 1
dup -> 2
iwrite -> 1
nwrite -> 1
iwrite -> 1
twrite -> 1
:write -> 1
 write -> 1
swrite -> 1
twrite -> 1
awrite -> 1
rwrite -> 1
twrite -> 1
iwrite -> 1
nwrite -> 1
gwrite -> 1
 write -> 1
swrite -> 1
hwrite -> 1

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

```
gautam-ahuja@LAPTOP-FV7627LB:~/.../cs1217-assignment-3-julius-stabs-back$ gd
b
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html
>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

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 configu
ration
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) c
Continuing.
```

2. Exercise 2

Done by: Gautam Ahuja
Problem: To add a new system call to xv6 for printing date.
Working:
As given in the question we run `grep -n uptime *.[chS]` to clone all of the pieces of code related to `uptime`.



```
gautam-ahuja@LAPTOP-FV7627LB:~/.../cs1217-assignment-3-julius-stabs-back$ grep -n uptime *.[chS]
syscall.c:106:extern int sys_uptime(void);
syscall.c:124:[SYS_uptime]  sys_uptime,
syscall.c:149:  [SYS_uptime]  "uptime",
syscall.c:212:      case SYS_uptime:
syscall.c:301://          cprintf("uptime -> %d\n", curproc->tf->eax);
syscall.h:15:#define SYS_uptime 14
sysproc.c:83:sys_uptime(void)
user.h:25:int uptime(void);
usys.S:31:SYSCALL(uptime)
gautam-ahuja@LAPTOP-FV7627LB:~/.../cs1217-assignment-3-julius-stabs-back$
```

From here we get an idea about the files we need to modify: `syscall.c, syscall.h, user.h, usys.S, sysproc.c, Makefile`.
Next we create a `date.c` file with the code:

```
#include "types.h"
#include "user.h"
#include "date.h"
int
main(int argc, char *argv[])
{
    struct rtcdate r;
    if (date(&r)) {
        // 2 is file descriptor for stderr
        printf(2, "date failed\n");
        exit();
    }
    // 1 is file descriptor for stdout
    printf(1, "Year: %d\nMonth: %d\nDay: %d\nHour : Minute : Seconds :: %d:%d:%d\n",
            r.year, r.month, r.day, r.hour, r.minute, r.second);
    exit();
}
```

In above, the first `printf()` statement inside `if` condition writes to file descripter 2, which is `stderr`.
The second `printf()` statement writes to output, through file descriptor 1, which is `stdout`.

Next, we defined a new syscall in `syscall.h` by `#define SYS_date 22` and give it a number 22.

```c
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_date   22
```

We also define the function type in `user.h`. Since the function which takes in argument a pointer to `rtcdate` structure. It is defined as: `int date(struct rtcdate*);`.

```c
int sleep(int);
int uptime(void);
// adding a new system call of date, it takes a pointer to a rtcdate struct as an argument
int date(struct rtcdate*);
```

Next we add `_date\` to UPROGS which is a list of user-level programs in the `Makefile`.

```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_date\
```

We also define a new syscall to the assembly file for user calls (`usys.S`) as:

usys.S

```
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(date)

```

Lastly we defined the implementation of date syscall in `sysproc.c` which includes system calls that are implemented in relation to management of processes. The `trapframe` discusses in exercise 1 always looks for a definition of a systemcall function in `sysproc.c` when it encounters a syscall.

sysproc.c

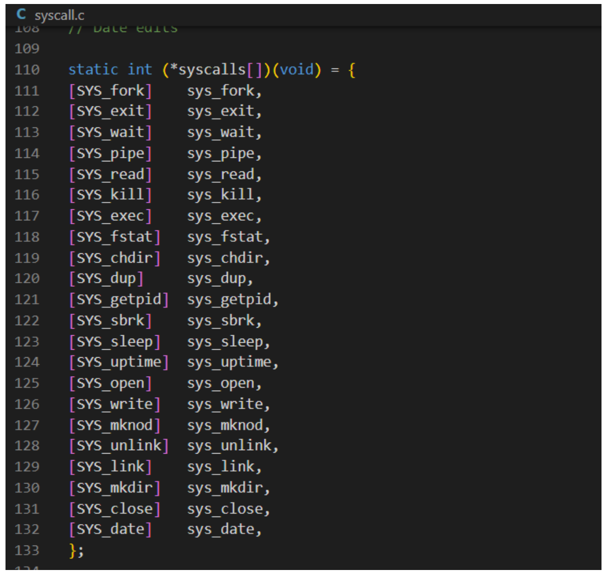
```

// Date edits
int
sys_date(void)
{
  struct rtcdate *r;
  if(argptr(0, (void*)&r, sizeof(*r)) < 0)
    return -1;
  cmostime(r);
  return 0;
}
```

The `if` statement checks if the incoming pointer to syscall is valid or not. The `argptr` takes input (through file descriptor 0, which is `stdin`) and checks if it lies within memory space. If not, it exits the syscall with a return code of `-1`. The definition of `argptr` is:

```
int argptr(int, char **, int)
```

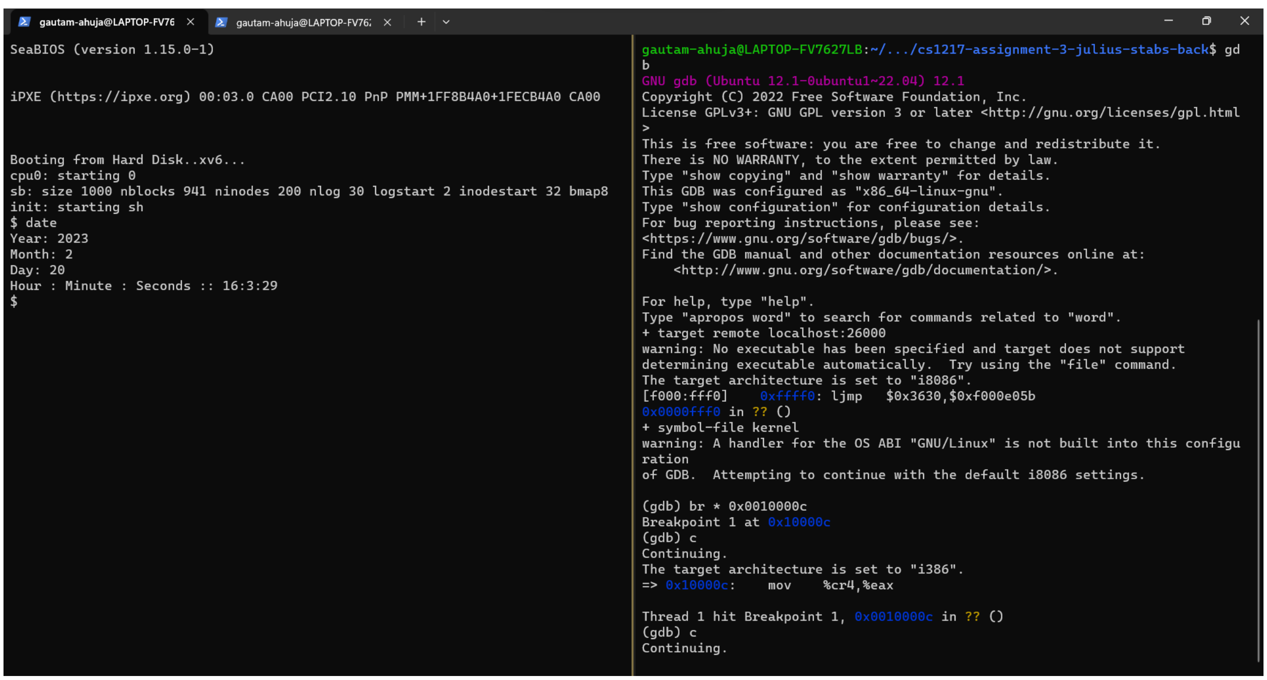
Fetch the nth word-sized system call argument as a pointer to a block of memory of size bytes. Check that the pointer lies within the process address space.

At last we add `sys_date` to the `syscall.c` file to the functional array decleration.



```c
syscall.c

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_date]    sys_date,
};
```

The output of the `date` syscall is:



```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap8
init: starting sh
$ date
Year: 2023
Month: 2
Day: 20
Hour : Minute : Seconds :: 16:3:29
$
```

```
gautam-ahuja@LAPTOP-FV7627LB:~/.../cs1217-assignment-3-julius-stabs-back$ gd
b
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html
>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

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 configu
ration
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) c
Continuing.
```