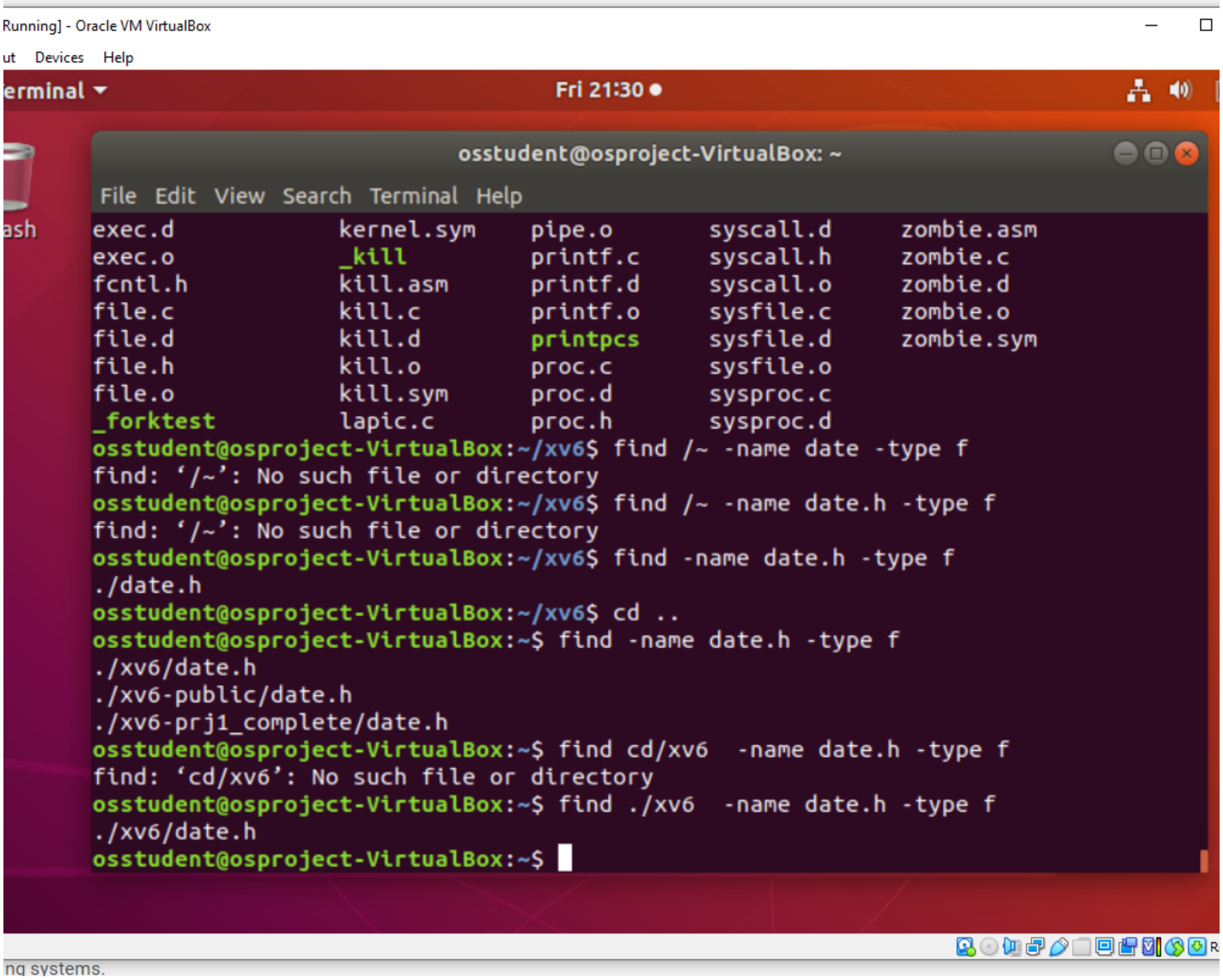
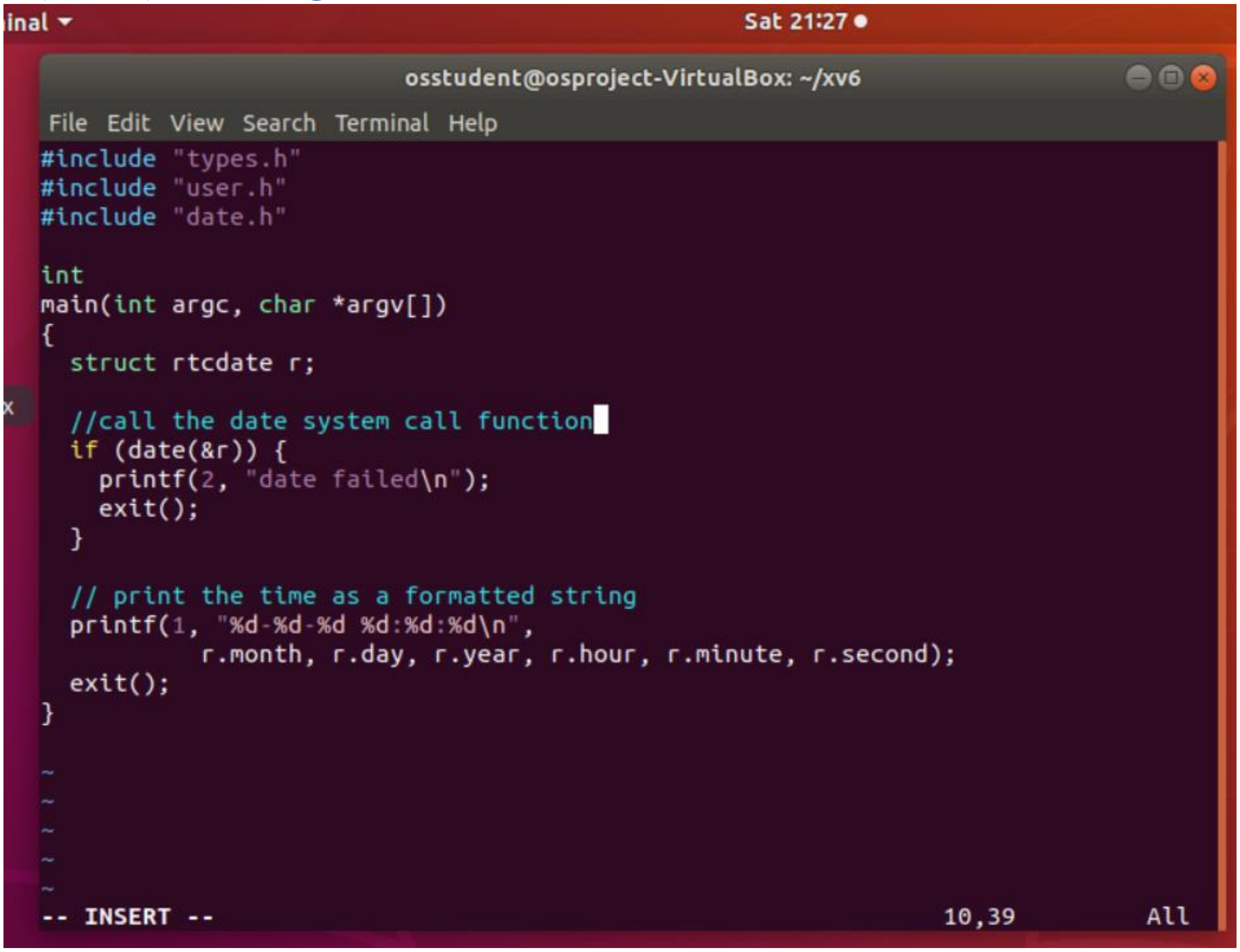
Student Name: Demetrius Johnson
Course: CIS-450-002
Professor: Dr. Jinhua Guo
Date: February 11, 2022
Due Date: February 11, 2022
Project 2 Report

Task 1: Using the *find* command



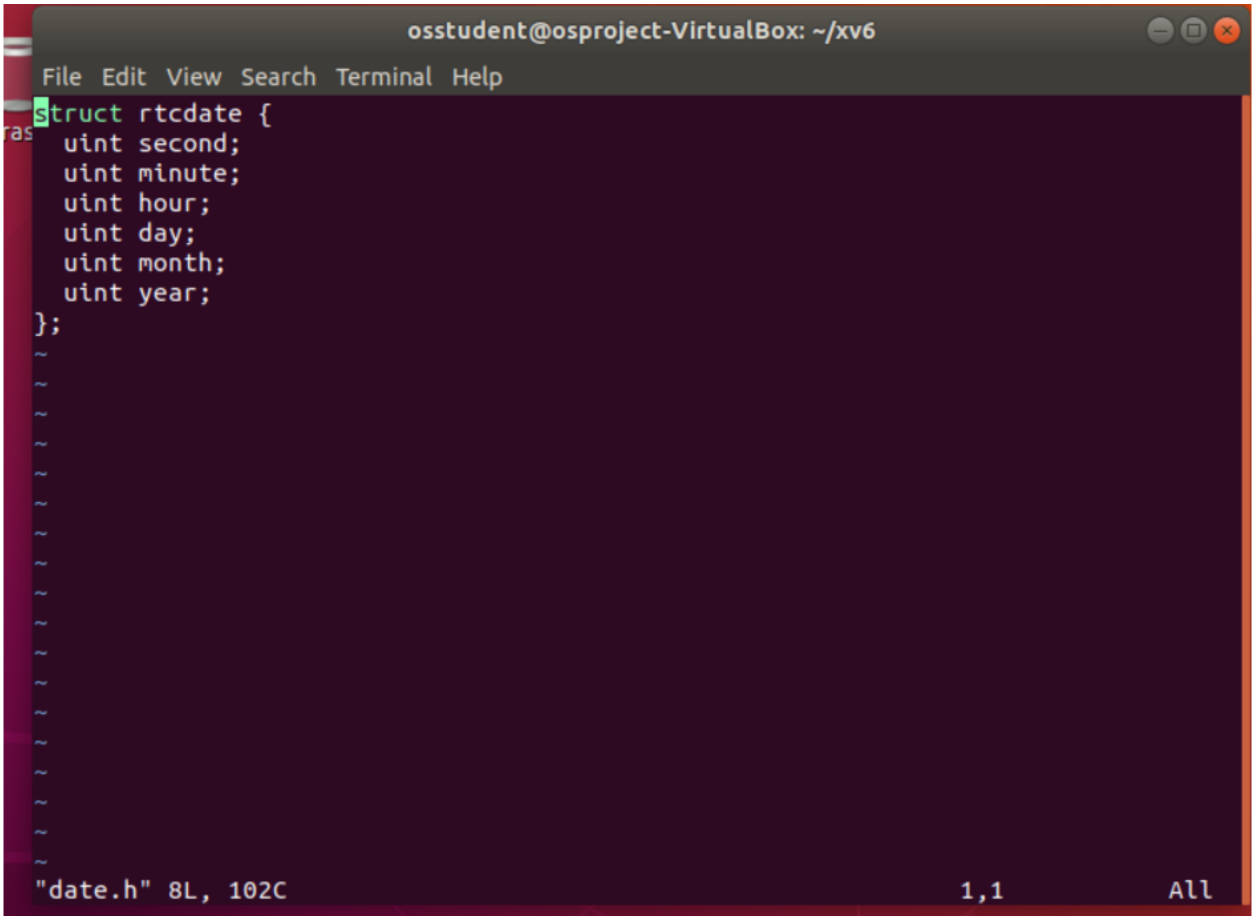
-
    - Above: I learned how to use the find command so that I can use that and the grep functions to effectively search for files and directories and text within files.
    - I have to type: find [file path to the file I want to search within, with the default being the current directory and all directories under that directory] [-name] [the name of my file or directory] [-type f == search for a file type; I can also specify directory type with -type d, and I can also not specify type and it will by default search for files and directories].

Task 2 (date.c): Creating *date.c* file

- 

    - Above: I have created a file called date.c inside xv6 directory and implemented the initial code that we are required to have in the user program.
    - I note from P2 discussion that the first parameter in the print function has integer value (1), and it tells the function to print/send output to the standard output.
    - If our system call function that we are creating – date() – fails it will return 0, indicating that the system call failed; then for the print function we use integer (2) for the first parameter (output parameter) in order to send error messages to cerr/stderr (standard error); note that by default stdoutput and stderr both stream output to the console window.
    - For xv6 and many other operating systems: 0 = kernel mode, 3 = user mode, and the levels in between have mixed privileges and are often not used.

Task 3 (date.h): Viewing the *date.h* file

- 

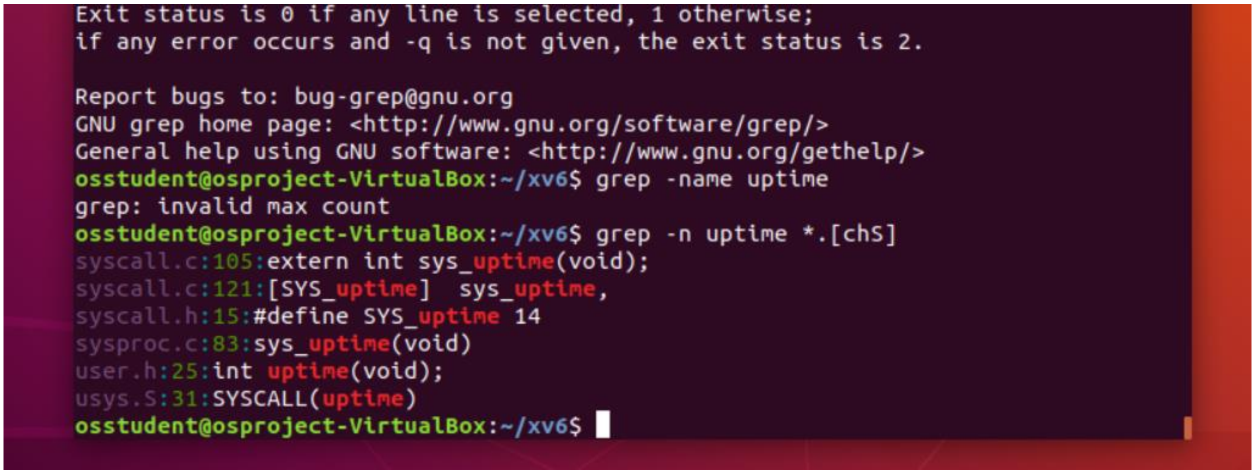
  osstudent@osproject-VirtualBox: ~/xv6

  File Edit View Search Terminal Help

  ```
  struct rtcdate {
    uint second;
    uint minute;
    uint hour;
    uint day;
    uint month;
    uint year;
  };
  "date.h" 8L, 102C                                   1,1          All
  ```

    - Above: showing the declaration and definition of the abstract data type: struct *rtcdate*.

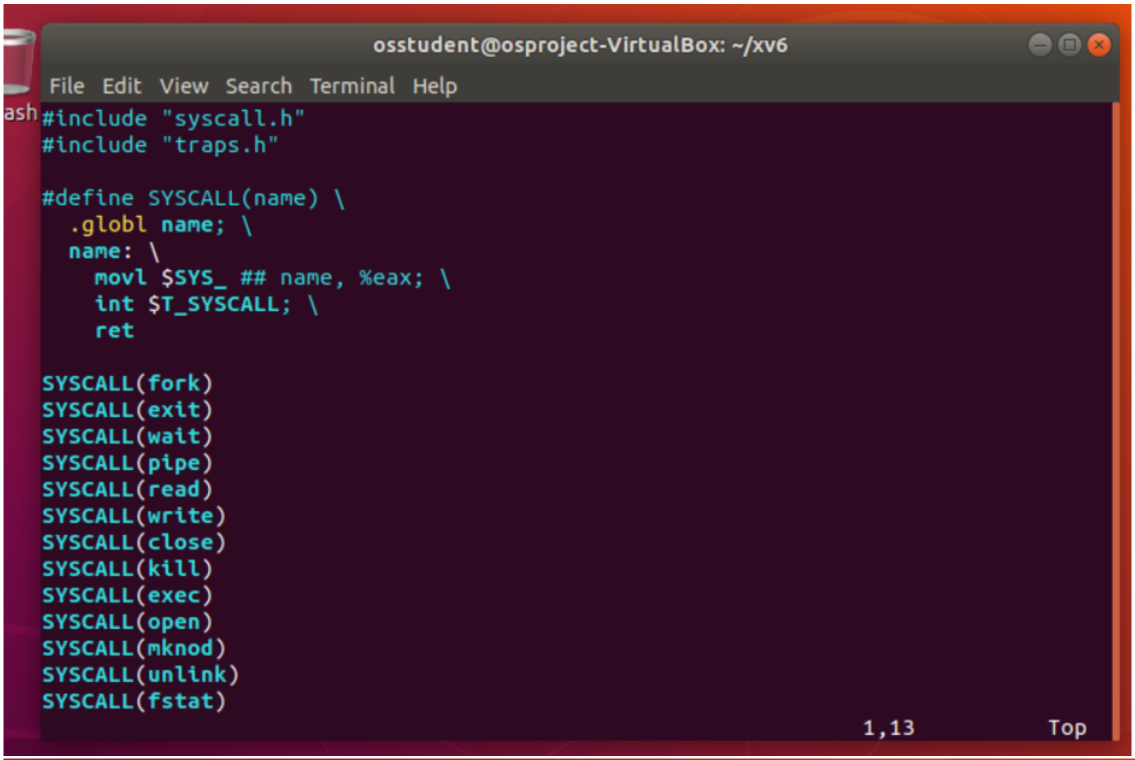
Task 4: Using the *grep* command

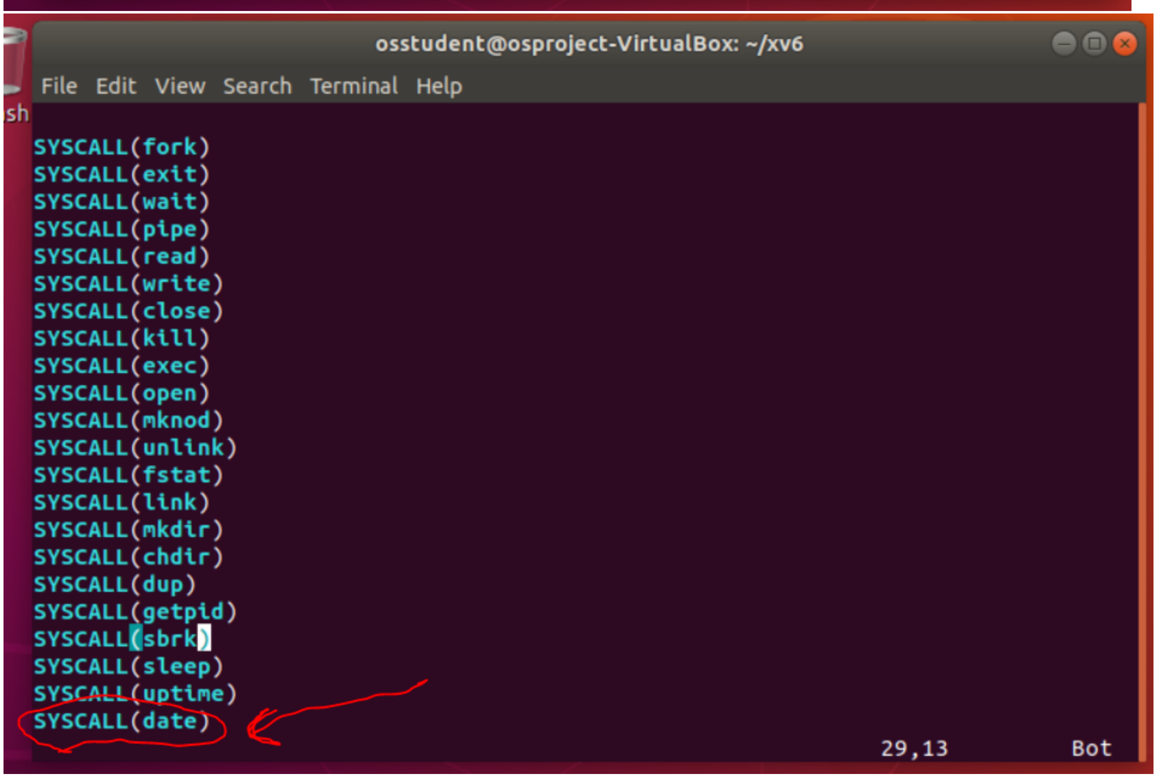
- 

```
Exit status is 0 if any line is selected, 1 otherwise;
if any error occurs and -q is not given, the exit status is 2.

Report bugs to: bug-grep@gnu.org
GNU grep home page: <http://www.gnu.org/software/grep/>
General help using GNU software: <http://www.gnu.org/gethelp/>
osstudent@osproject-VirtualBox:~/xv6$ grep -name uptime
grep: invalid max count
osstudent@osproject-VirtualBox:~/xv6$ grep -n uptime *.[chS]
syscall.c:105:extern int sys_uptime(void);
syscall.c:121:[SYS_uptime]  sys_uptime,
syscall.h:15:#define SYS_uptime 14
sysproc.c:83:sys_uptime(void)
user.h:25:int uptime(void);
usys.S:31:SYSCALL(uptime)
osstudent@osproject-VirtualBox:~/xv6$
```

  - Above, calling the grep command, searching for the string “uptime” in any file with either .c, .h, or .S file extension in their file name (assembly language file extension types).
  - The actual implementation of my system code for my new system call function date() should be in the system process c file: sysproc.c

Task 5 (usys.S): Add SYSCALL(date) to usys.S file

- 

- 

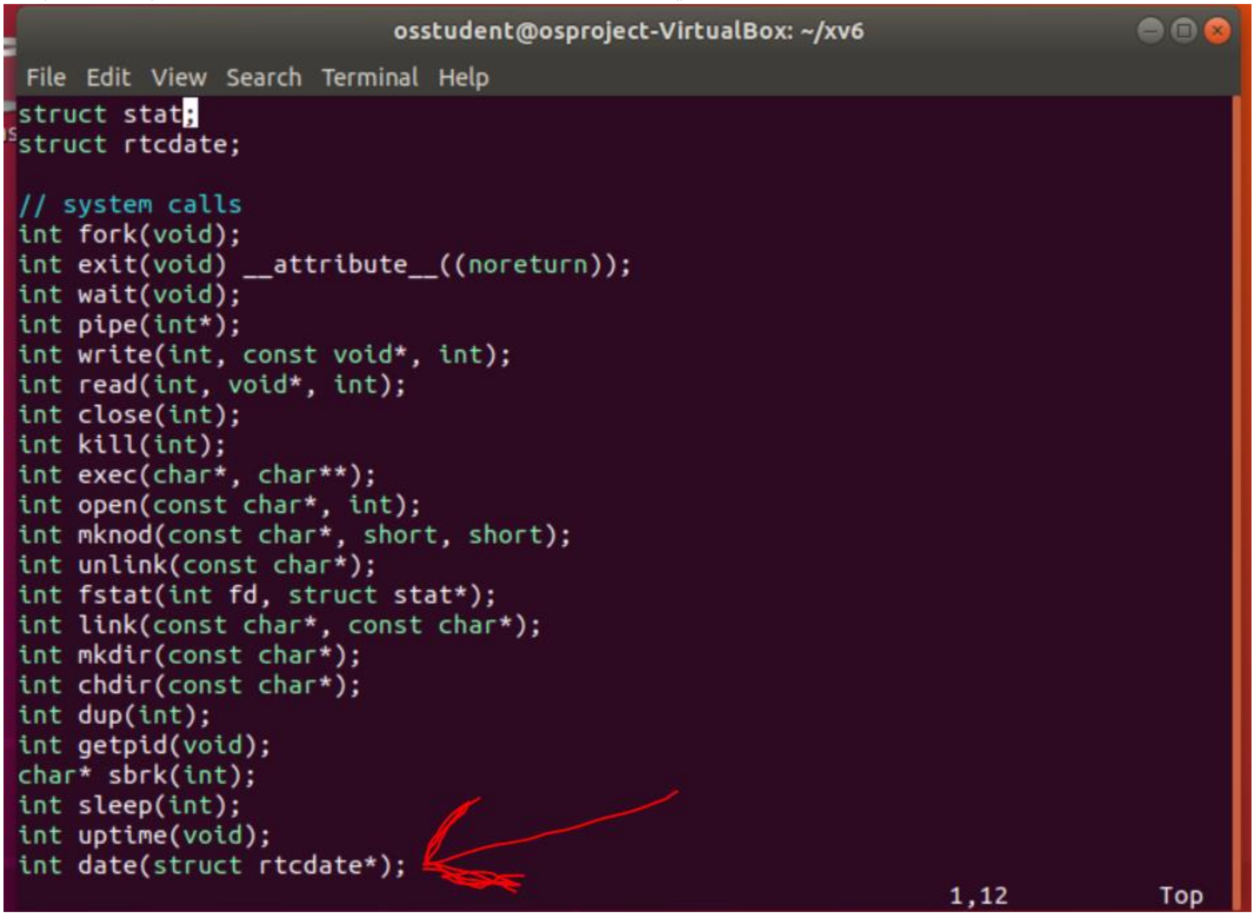
    - Above we look inside the usys.S (user system call file) assembly language file; SYSCALL is a macro function which is defined in our usys.S function; see examples the expansion of a few of the macro function implemented with specific names below after we compile usys.S code:

- ```
  .globl write; write: movl $16, %eax; int $64; ret
  .globl close; close: movl $21, %eax; int $64; ret
  .globl kill; kill: movl $6, %eax; int $64; ret
  ```
- Notice the identification integer value for write, close, and kill being passed into SYSCALL macro function is 16, 21, and 6, respectively. We will add an identifier to our date function system call in another file (syscall.h), and assign it a unique integer value.

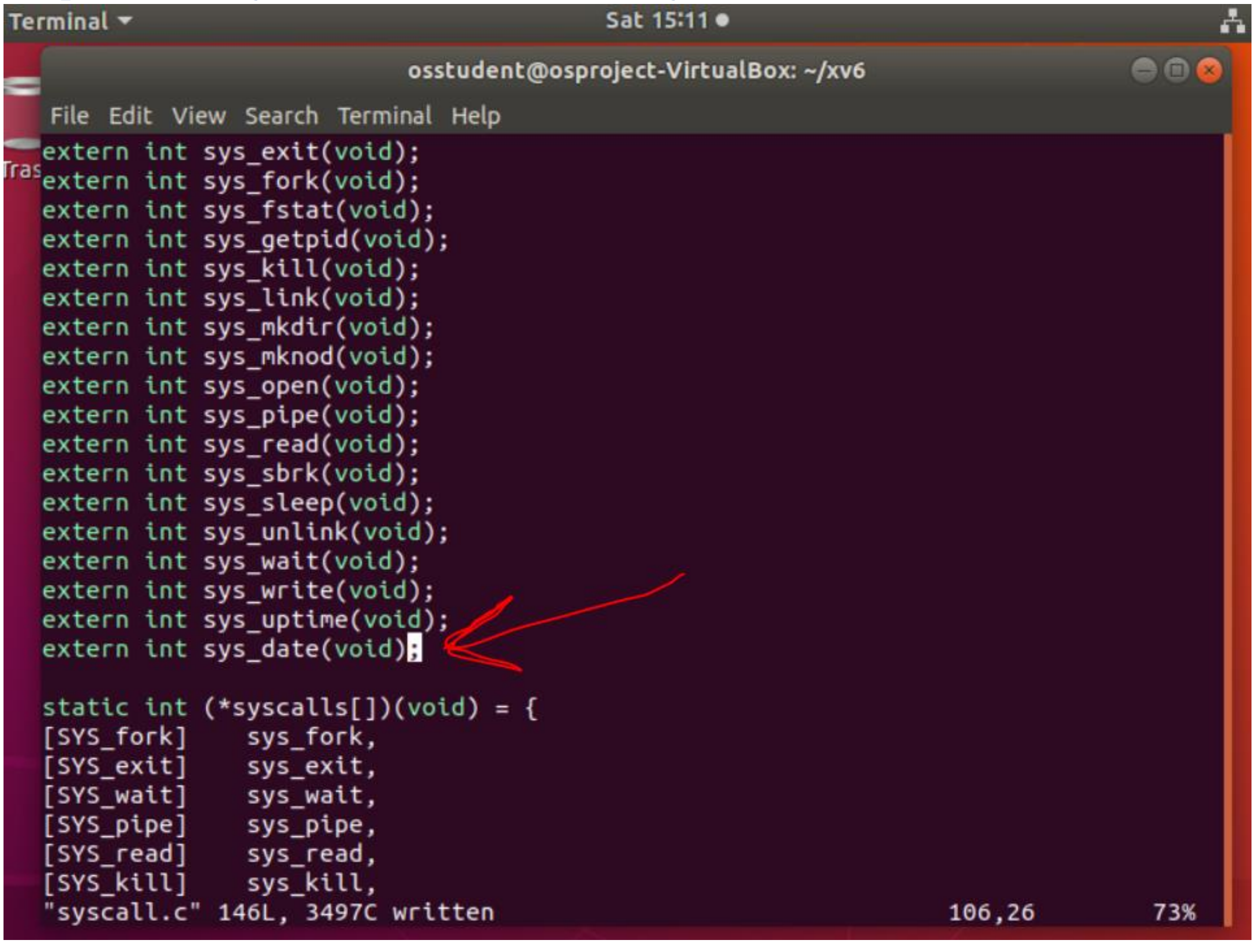
- movl simply moves a value in the eax register to do a specific system call based on the value placed into the register; T_SYSCALL is an integer constant (64) → int in assembly language is not "integer" but "interrupt"; so we do a system interrupt 64 to interrupt the system and call the given system call placed into the eax register.
- In xv6 this is how we implement a system call: we do an interrupt; we know that the interrupt is normally a hardware interrupt – (such as when you have a disk operation complete) hardware will generate an interrupt; they are between 32 and 63. 0 to 31 are the software interrupts (such as divide by 0 error or segmentation fork; voluntarily give up CPU); 64 is the system call interrupt (also known as trap) and is also technically a software interrupt. The system interrupt such as syscall allows us to get into kernel mode (and implement the system call code; such as the uptime function, or the new date system call function we are implementing).
- Finally, we add the date system call that we are implementing in this project.
- Note to self: Here is how CPU privilege is raised after the interrupt 64 (int 64) is executed, as described from the github website article:

Kernel Side: Trap Tables

Once the **int** instruction is executed, the hardware takes over and does a bunch of work on behalf of the caller. One important thing the hardware does is to raise the *privilege level* of the CPU to kernel mode; on x86 this is usually means moving from a *CPL (Current Privilege Level)* of 3 (the level at which user applications run) to CPL 0 (in which the kernel runs). Yes, there are a couple of in-between privilege levels, but most systems do not make use of these.

The second important thing the hardware does is to transfer control to the *trap vectors* of the system. To enable the hardware to know what code to run when a particular trap occurs, the OS, when booting, must make sure to inform the hardware of the location of the code to run when such traps take place. This is done in **main.c** as follows:
  ```

Task 6: Compile usys.S to view all the macro functions generated

- ```
  osstudent@osproject-VirtualBox: ~/xv6
  File Edit View Search Terminal Help
  usys.S:31:SYSCALL(uptime)
  osstudent@osproject-VirtualBox:~/xv6$ vim usys.S
  osstudent@osproject-VirtualBox:~/xv6$ vim usys.S
  osstudent@osproject-VirtualBox:~/xv6$ gcc -S usys.S
  # 1 "usys.S"
  # 1 "<built-in>"
  # 1 "<command-line>"
  # 31 "<command-line>"
  # 1 "/usr/include/stdc-predef.h" 1 3 4
  # 32 "<command-line>" 2
  # 1 "usys.S"
  # 1 "syscall.h" 1
  # 2 "usys.S" 2
  # 1 "traps.h" 1
  # 3 "usys.S" 2
  # 11 "usys.S"
  .globl fork; fork: movl $1, %eax; int $64; ret
  .globl exit; exit: movl $2, %eax; int $64; ret
  .globl wait; wait: movl $3, %eax; int $64; ret
  .globl pipe; pipe: movl $4, %eax; int $64; ret
  .globl read; read: movl $5, %eax; int $64; ret
  .globl write; write: movl $16, %eax; int $64; ret
  .globl close; close: movl $21, %eax; int $64; ret
  .globl kill; kill: movl $6, %eax; int $64; ret
  ```
- ```
  .globl pipe; pipe: movl $4, %eax; int $64; ret
  .globl read; read: movl $5, %eax; int $64; ret
  .globl write; write: movl $16, %eax; int $64; ret
  .globl close; close: movl $21, %eax; int $64; ret
  .globl kill; kill: movl $6, %eax; int $64; ret
  .globl exec; exec: movl $7, %eax; int $64; ret
  .globl open; open: movl $15, %eax; int $64; ret
  .globl mknod; mknod: movl $17, %eax; int $64; ret
  .globl unlink; unlink: movl $18, %eax; int $64; ret
  .globl fstat; fstat: movl $8, %eax; int $64; ret
  .globl link; link: movl $19, %eax; int $64; ret
  .globl mkdir; mkdir: movl $20, %eax; int $64; ret
  .globl chdir; chdir: movl $9, %eax; int $64; ret
  .globl dup; dup: movl $10, %eax; int $64; ret
  .globl getpid; getpid: movl $11, %eax; int $64; ret
  .globl sbrk; sbrk: movl $12, %eax; int $64; ret
  .globl sleep; sleep: movl $13, %eax; int $64; ret
  .globl uptime; uptime: movl $14, %eax; int $64; ret
  .globl date; date: movl $SYS_date, %eax; int $64; ret
  osstudent@osproject-VirtualBox:~/xv6$
  ```
  - Above, using gcc compiler with option S to compile the usys.S assembly language file, showing the code defines a macro function from the defined macro template for each particular system call instance.
  - As an example: for the uptime syscall function, $14 is the specific number used to identify a specific system call such as uptime and is placed into the eax register; then int $64 is the SYSCALL interrupt that is generated to capture the trapped value and allow uptime to get into kernel mode.

- Note to self: In assemblers, symbol **$** usually means two different things: Prefixing a number, means that this number is written in hexadecimal. By itself, $ is a numeric expression that evaluates as "the current position", that is, the address where the next instruction/data would be assembled. The movl means “move long”; as in an integer of size long.
- The eax register not only stored the user system call value, but it also stores the return value of the system call (0 = success, 1 or -1 = fail, usually in this format since integer value 1 is used for standard output, and -1 is used for stderr output.)

Task 7 (user.h): Add declaration for the date() function in the user.h file

- 

    - Above, adding the declaration for the date() function in the user.h header file, so that the signature exists for any function that calls date() during compilation; otherwise, compiler will complain there is no existing function defined called date(); of course, definition will be done in another file. So, this is basically a function signature interfaced in the system call with the user applications.
    - NOTE to self: uptime() and date() are system call functions. They interface the user functions with system calls, defined in user.h.
    -

Task 8 (syscall.h): Define another integer, named SYS_date in the syscall.h file

- ```
  osstudent@osproject-VirtualBox: ~/xv6
  File Edit View Search Terminal Help
  // 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
  ~
  ~
  ~
  "syscall.h" 23L, 507C written                    23,18         All
  ```
    - Above, we enter the syscall.h assembly language header file. We define another integer, named SYS_date, and assign it an unused syscall identification number; for this case, we assign SYS_date the integer value 22 (decimal by default; remember I could write 22d to specify decimal).
    - Now, SYSCALL macro function from the usys.S file will know what to replace the defined but unassigned identifying integer SYS_ ## name (name = date, ## = 22):
        - ```
          #define SYSCALL(name) \
            .globl name; \
            name: \
              movl $SYS_ ## name, %eax; \
              int $T_SYSCALL; \
              ret
          ```
        - So now the template macro function will look like this:
        - .globl date;   \
        - movl $SYS_date, %eax; \
        - int $T_SYSCALL; \
        - Ret
        - ;so for the above code, int = interrupt in assembly, T_SYSCALL = integer value 64, SYS_date = 22. Also note: .globl means this is a global function.
        - Here is a screenshot from how it looks after compilation, as shown earlier in tasks 4 and 5:
        - ```
          .globl date; date: movl $SYS_date, %eax; int $64; ret
          osstudent@osproject-VirtualBox:~/xv6$
          ```
          ```

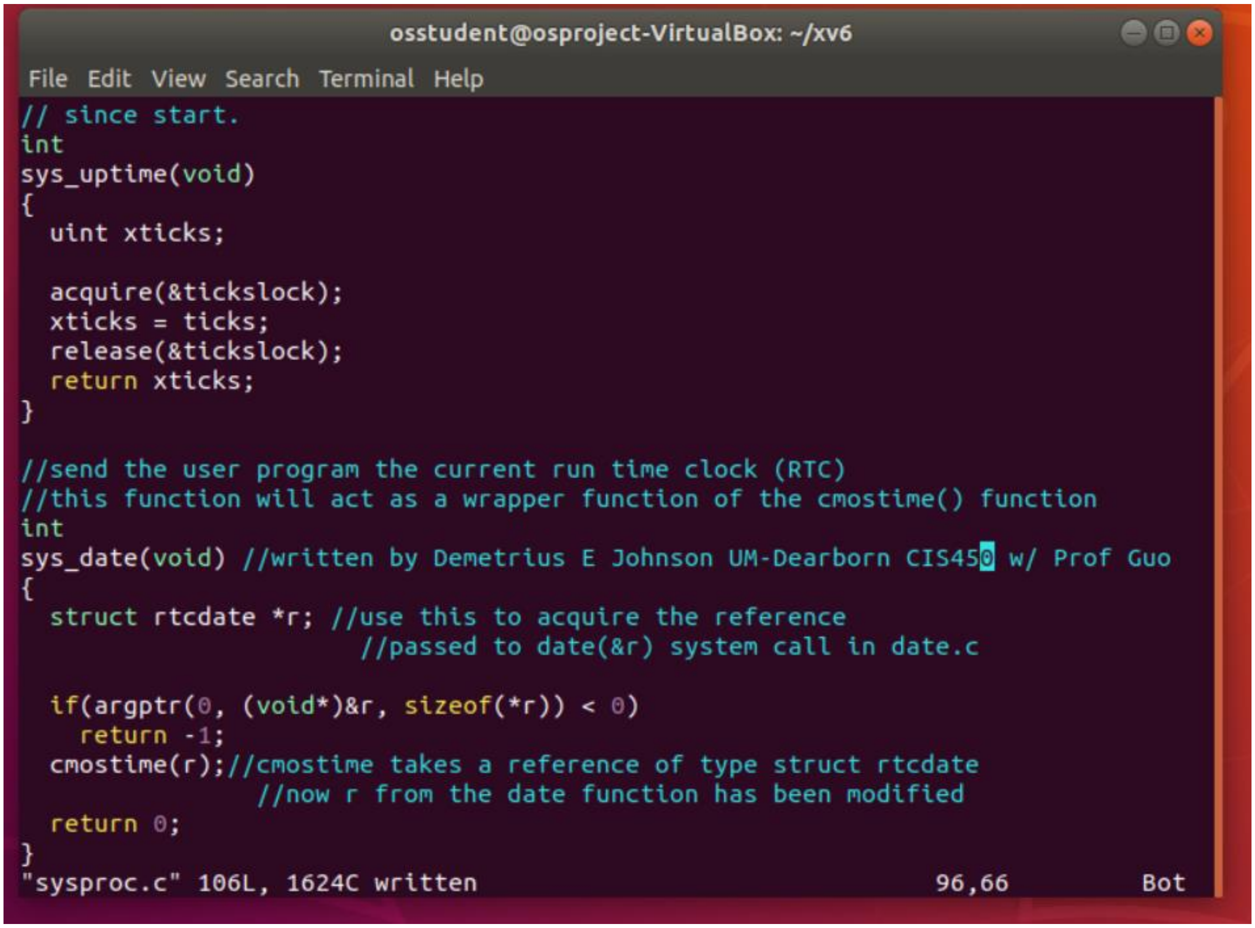
Task 9 (syscall.c): Add sys_date(void) external function and add function pointer to sys_date function in the syscall.c file

- 

    - Above, adding the external function sys_date() so that syscall.c function can call the sys_date function, which is outside of the scope of syscall.c function.
    - It is a reference to an external function that we are going to implement in the sysproc.c file (system process file); all of the user system call functions are implemented in the sysproc.c and systemfile.c files.
    - Note: sysproc.c is similar to the sysfile.c file, they both implement system code.

- ```
  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,
  };
                                                              112,23        87%
  ```
    - Above, adding the sys_date to the function pointer array.
    - Notice above, syscalls[]. This data variable is an array of function pointers that point the functions shown below it inside the implementation; the left side are the integers that were defined in syscall.h file, and the right side are all the function pointers to the external functions.
- ```
  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;
    }
  }
  "syscall.c" 146L, 3497C                                      140,1         Bot
  ```
    - Above, is how a user program invokes a system call when a SYSCALL interrupt is generated; the interrupt will call the syscall() function shown above. So, for an interrupt to get to a c function, it calls the c function syscall() (implemented in the sysproc.c file). This is the function that checks the SYSCALL (interrupt function from the usys.S file) value, which was stored in the eax register before we reach this syscall() c function. The array of function pointers variable syscalls[] is used, where num = eax value; thus it is similar to doing this: syscalls[num] == syscalls[eax].

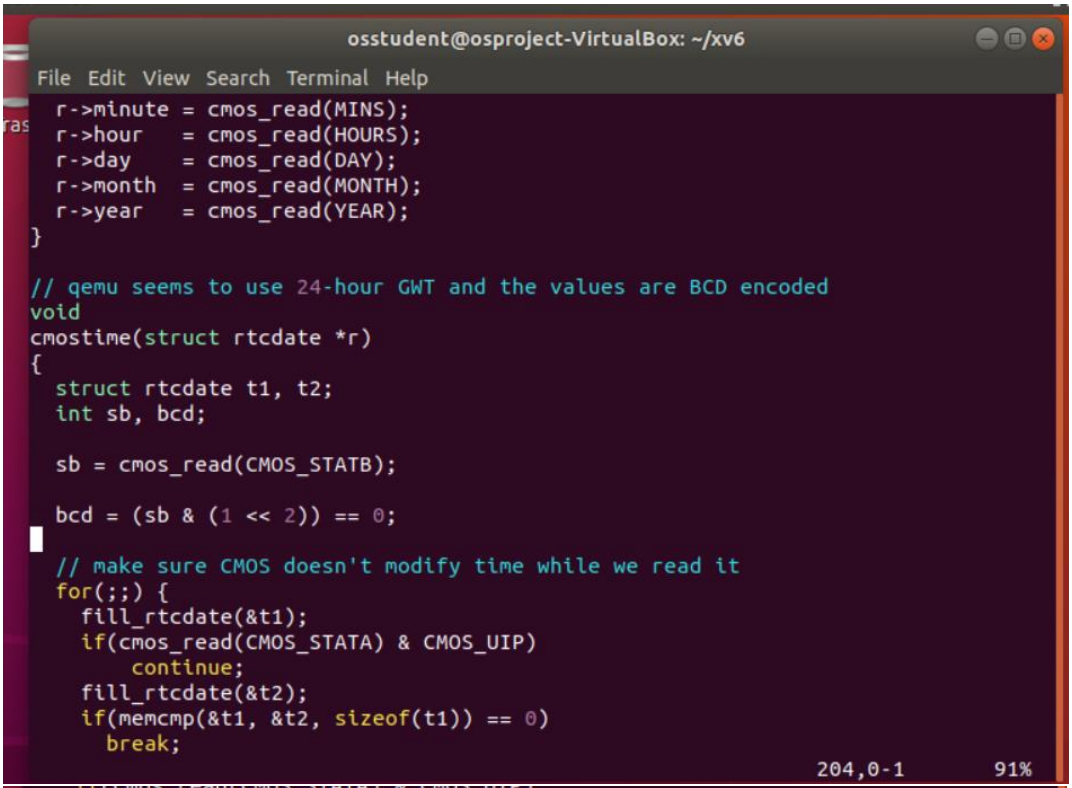
Task 10 (sysproc.c): Add/define and implement sys_date(void) function to the sysproc.c file

- 

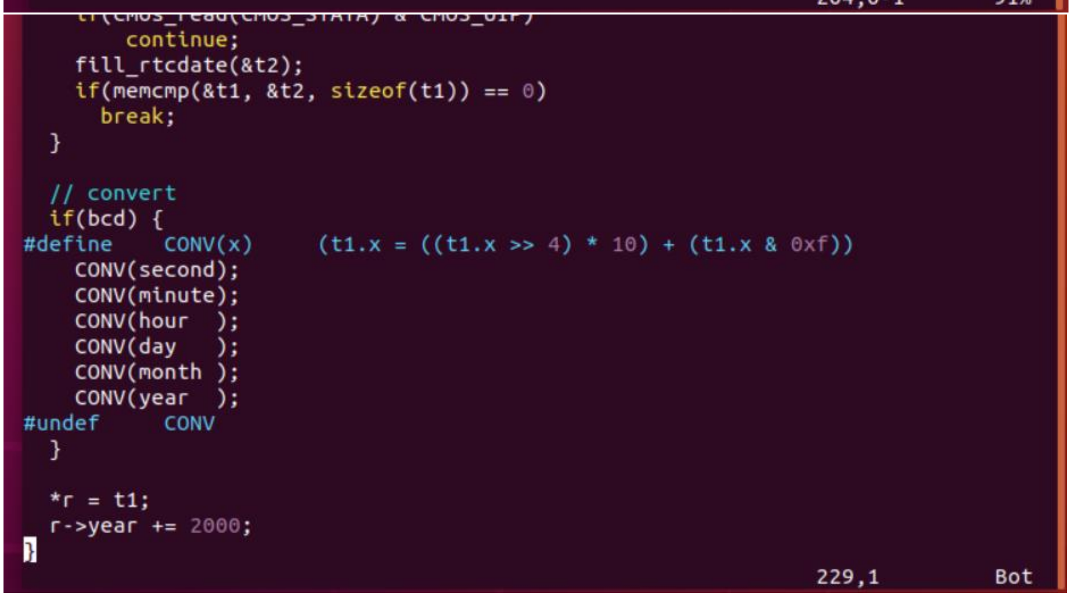
```
osstudent@osproject-VirtualBox: ~/xv6
File Edit View Search Terminal Help
// since start.
int
sys_uptime(void)
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}

//send the user program the current run time clock (RTC)
//this function will act as a wrapper function of the cmostime() function
int
sys_date(void) //written by Demetrius E Johnson UM-Dearborn CIS450 w/ Prof Guo
{
  struct rtcdate *r; //use this to acquire the reference
                       //passed to date(&r) system call in date.c

  if(argptr(0, (void*)&r, sizeof(*r)) < 0)
    return -1;
  cmostime(r);//cmostime takes a reference of type struct rtcdate
                //now r from the date function has been modified
  return 0;
}
"sysproc.c" 106L, 1624C written                              96,66          Bot
```

  - Above, I have added the sys_date() function definition to the sysproc.c file, under uptime function as shown. Now I have to write the implementation inside of this function definition so that it is defined to return run time clock using the cmostime() function.
  - We need to get the rtc struct reference so that we can pass it to cmostime() function and use the return of the cmostime() function in our sys_date() function.
  - Essentially, my system call function will act as a wrapper function.

Task 11 (lapic.c): Viewing the cmostime() function in the lapic.c file

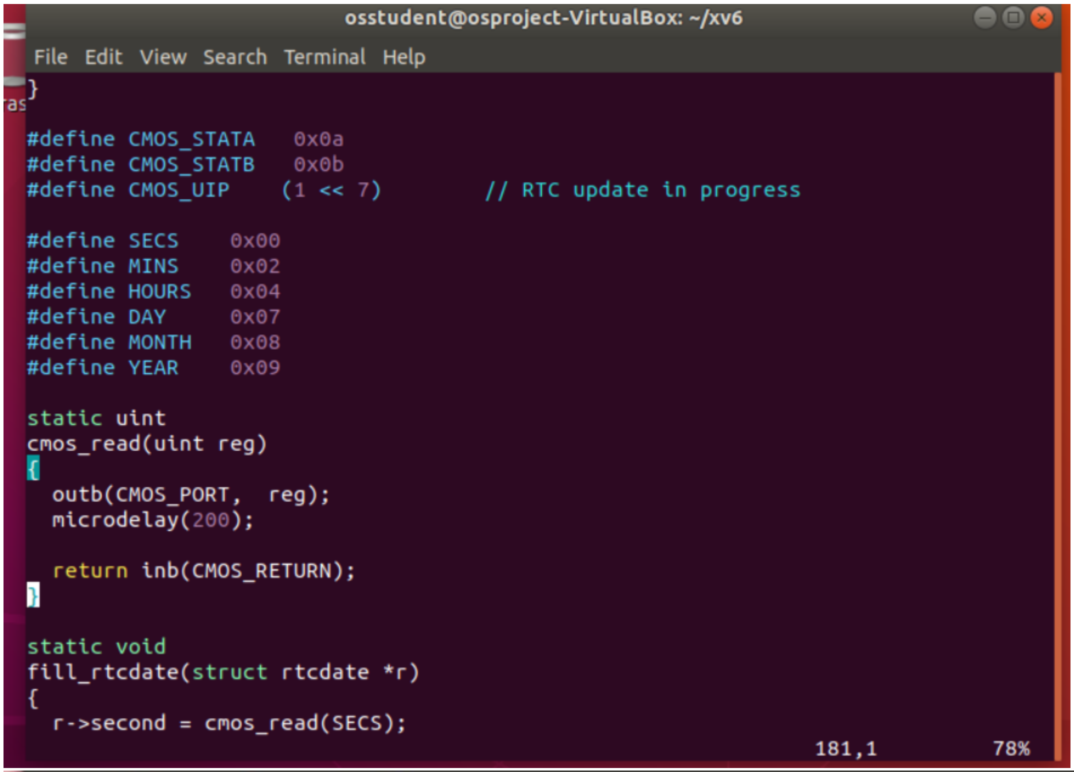
- 

```
osstudent@osproject-VirtualBox: ~/xv6
File  Edit  View  Search  Terminal  Help
  r->minute = cmos_read(MINS);
  r->hour   = cmos_read(HOURS);
  r->day    = cmos_read(DAY);
  r->month  = cmos_read(MONTH);
  r->year   = cmos_read(YEAR);
}

// qemu seems to use 24-hour GWT and the values are BCD encoded
void
cmostime(struct rtcdate *r)
{
  struct rtcdate t1, t2;
  int sb, bcd;

  sb = cmos_read(CMOS_STATB);

  bcd = (sb & (1 << 2)) == 0;

  // make sure CMOS doesn't modify time while we read it
  for(;;) {
    fill_rtcdate(&t1);
    if(cmos_read(CMOS_STATA) & CMOS_UIP)
        continue;
    fill_rtcdate(&t2);
    if(memcmp(&t1, &t2, sizeof(t1)) == 0)
      break;
                                                   204,0-1       91%
```

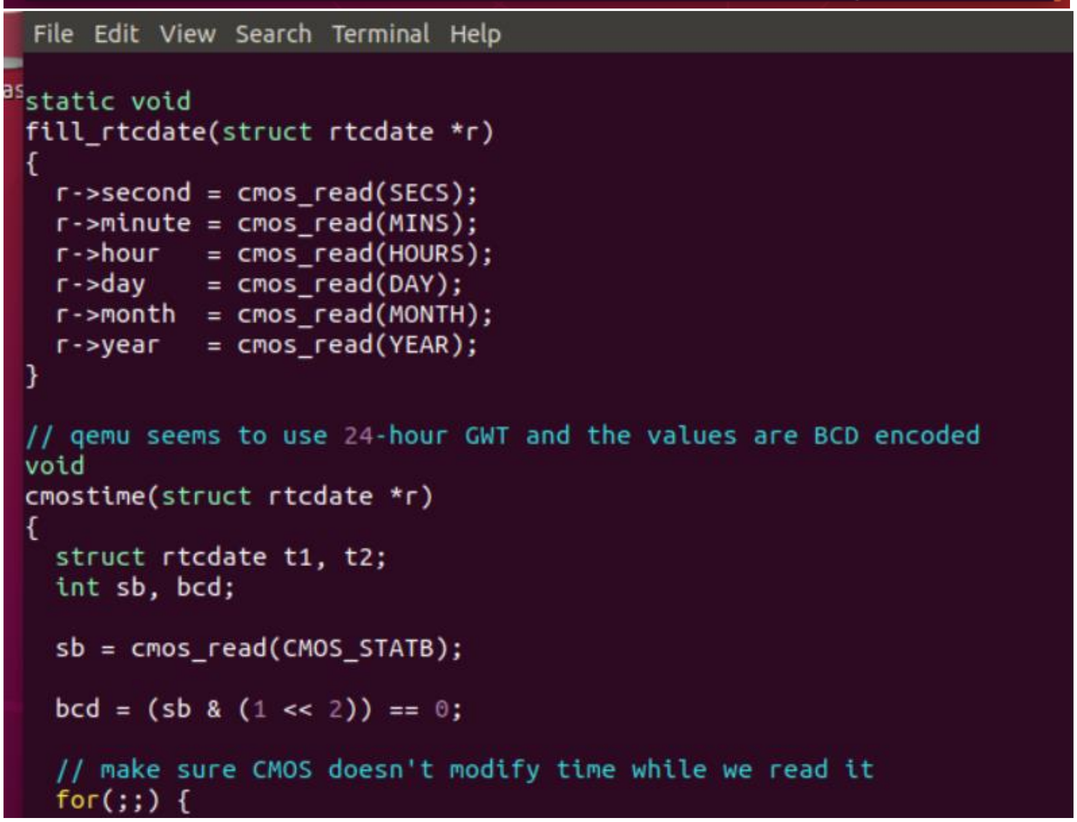
- 

```
        continue;
    fill_rtcdate(&t2);
    if(memcmp(&t1, &t2, sizeof(t1)) == 0)
      break;
  }

  // convert
  if(bcd) {
#define     CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
    CONV(second);
    CONV(minute);
    CONV(hour  );
    CONV(day   );
    CONV(month );
    CONV(year  );
#undef      CONV
  }

  *r = t1;
  r->year += 2000;
}
                                                   229,1         Bot
```

  - In the above two screenshots: showing the cmostime() function from the lapic.c xv6 system file. This is the function (along with the functions defined above it) that actually does the work/implementation of acquiring the system run time clock value and converting it to integer form at. Note: cmostime is just a function. But, it must be called in the kernel mode.
  - Here are the functions above it that also assist the cmostime() function:

- 

```
osstudent@osproject-VirtualBox: ~/xv6
File  Edit  View  Search  Terminal  Help
}

#define CMOS_STATA   0x0a
#define CMOS_STATB   0x0b
#define CMOS_UIP    (1 << 7)        // RTC update in progress

#define SECS    0x00
#define MINS    0x02
#define HOURS   0x04
#define DAY     0x07
#define MONTH   0x08
#define YEAR    0x09

static uint
cmos_read(uint reg)
{
  outb(CMOS_PORT,  reg);
  microdelay(200);

  return inb(CMOS_RETURN);
}

static void
fill_rtcdate(struct rtcdate *r)
{
  r->second = cmos_read(SECS);
                                                              181,1          78%
```

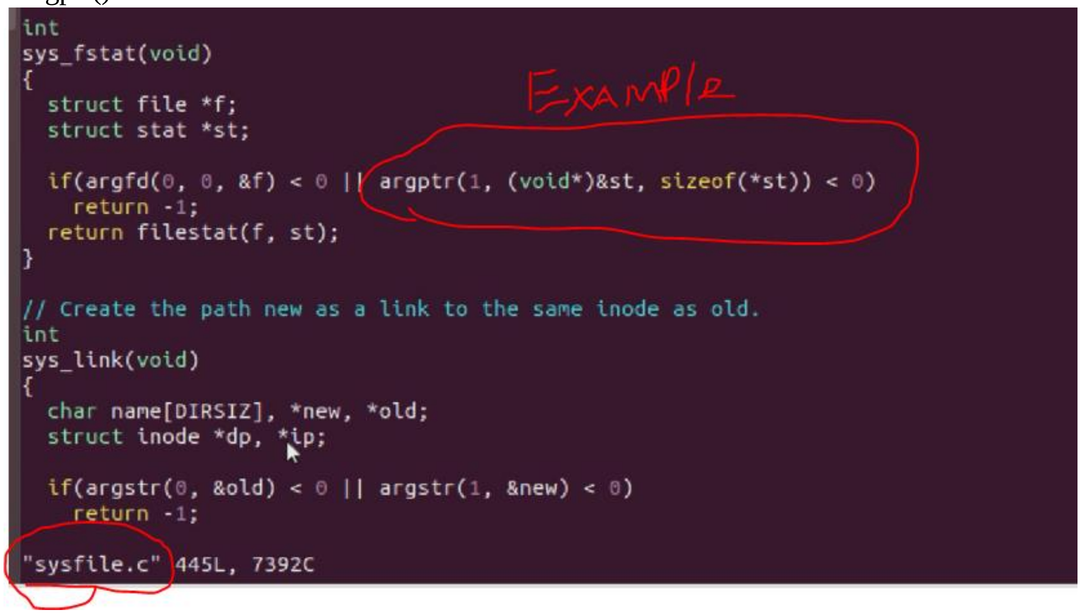
- 

```
File  Edit  View  Search  Terminal  Help

static void
fill_rtcdate(struct rtcdate *r)
{
  r->second = cmos_read(SECS);
  r->minute = cmos_read(MINS);
  r->hour   = cmos_read(HOURS);
  r->day    = cmos_read(DAY);
  r->month  = cmos_read(MONTH);
  r->year   = cmos_read(YEAR);
}

// qemu seems to use 24-hour GWT and the values are BCD encoded
void
cmostime(struct rtcdate *r)
{
  struct rtcdate t1, t2;
  int sb, bcd;

  sb = cmos_read(CMOS_STATB);

  bcd = (sb & (1 << 2)) == 0;

  // make sure CMOS doesn't modify time while we read it
  for(;;) {
```

- Remember, this code is apart of the OS code, so when xv6 runs, the main function (main.c), which can be acquired using the argptr() function. This is what we will use to acquire the rtcdate struct reference in the cmostime() function, and use it in our sys_date() system call function implementation.

Task 12: Using argptr() helper function to fetch a *pointer* argument in a system call

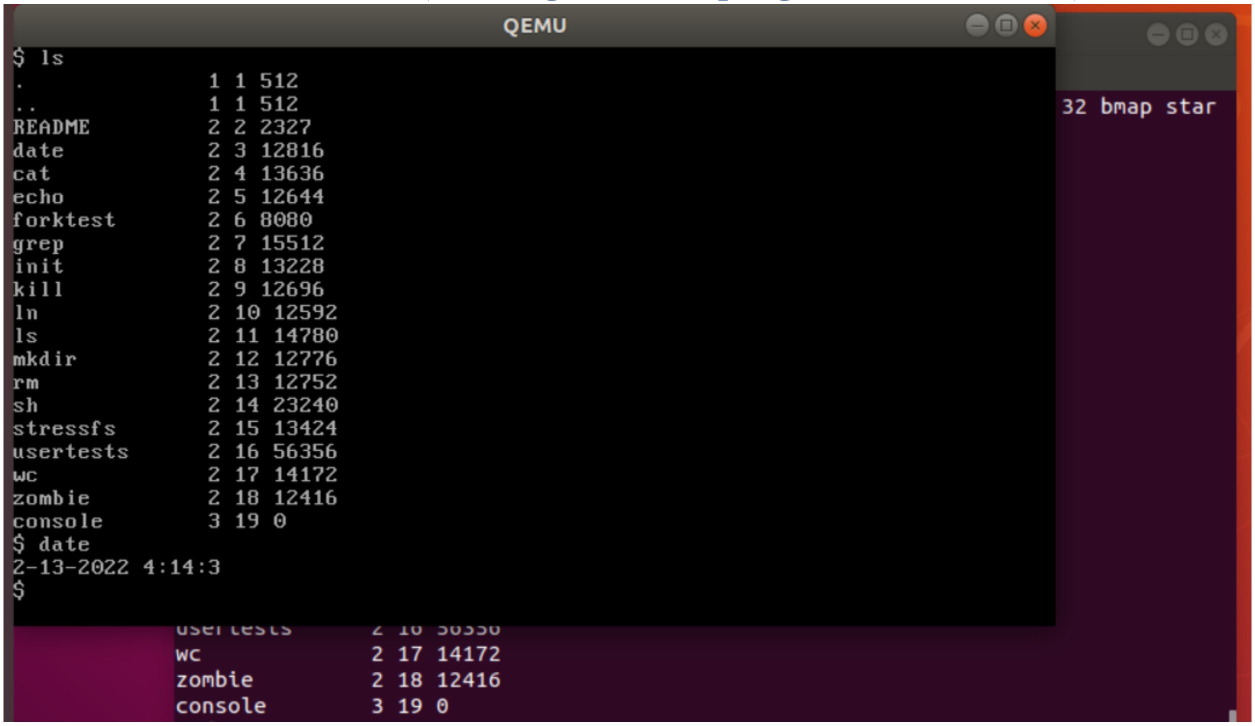
- In the system code of sysfile.c, there are examples as mentioned from the project 2 instructions of using the argptr() function:




```
int
sys_fstat(void)
{
  struct file *f;
  struct stat *st;

  if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
    return -1;
  return filestat(f, st);
}

// Create the path new as a link to the same inode as old.
int
sys_link(void)
{
  char name[DIRSIZ], *new, *old;
  struct inode *dp, *ip;

  if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
    return -1;

"sysfile.c" 445L, 7392C
```

  -
  - Remember: the &rtc reference is pushed into the stack from the cmostime() function; thus we can go to that location in the stack to acquire that reference, using argptr() function.

Task 13: Final Solution (running the date program in xv6 shell)



- Notice above: I was finally able to finish the program at 2-12-2022 23:14:03 EST, which translates to 2-13-2022 4:14:03 UTC.

Task 14: Summary/Reflection (>100 words)

Overall, this project was extremely difficult for me. Once I finished, I realized the amount of code we needed to add was very minimal; however, trying to understand everything about how system calls work made it very difficult to understand why exactly we made changes to all the files in the way that was required. I have learned a great deal, however, about how the CPU switches states, and how its privilege level changes based on interrupts. I learned a bit about traps – the method that is used to go from a user program down into kernel mode so that kernel-level code can be executed. I also learned that the data on the CPU of the original caller of the function is placed in a trap frame above the kernel, and that we can access the arguments passed into the caller of the system call function. Finally, figuring out how to wrap the cmostime() function into my sys_date() function was simple, but as I mentioned before, it was difficult to understand so that is why it took me so long to finally figure out what I needed to do with the argptr() function; to understand the argptr() function and why it was necessary, you necessarily have to understand how exactly the trap function works and how exactly xv6 handles interrupts and switches into kernel mode.

Additional Notes

Notes to self (main.c)

- Kernel side: Trap Tables (using xv6 as the example of what the ideas that the github article was talking about)

Kernel Side: Trap Tables

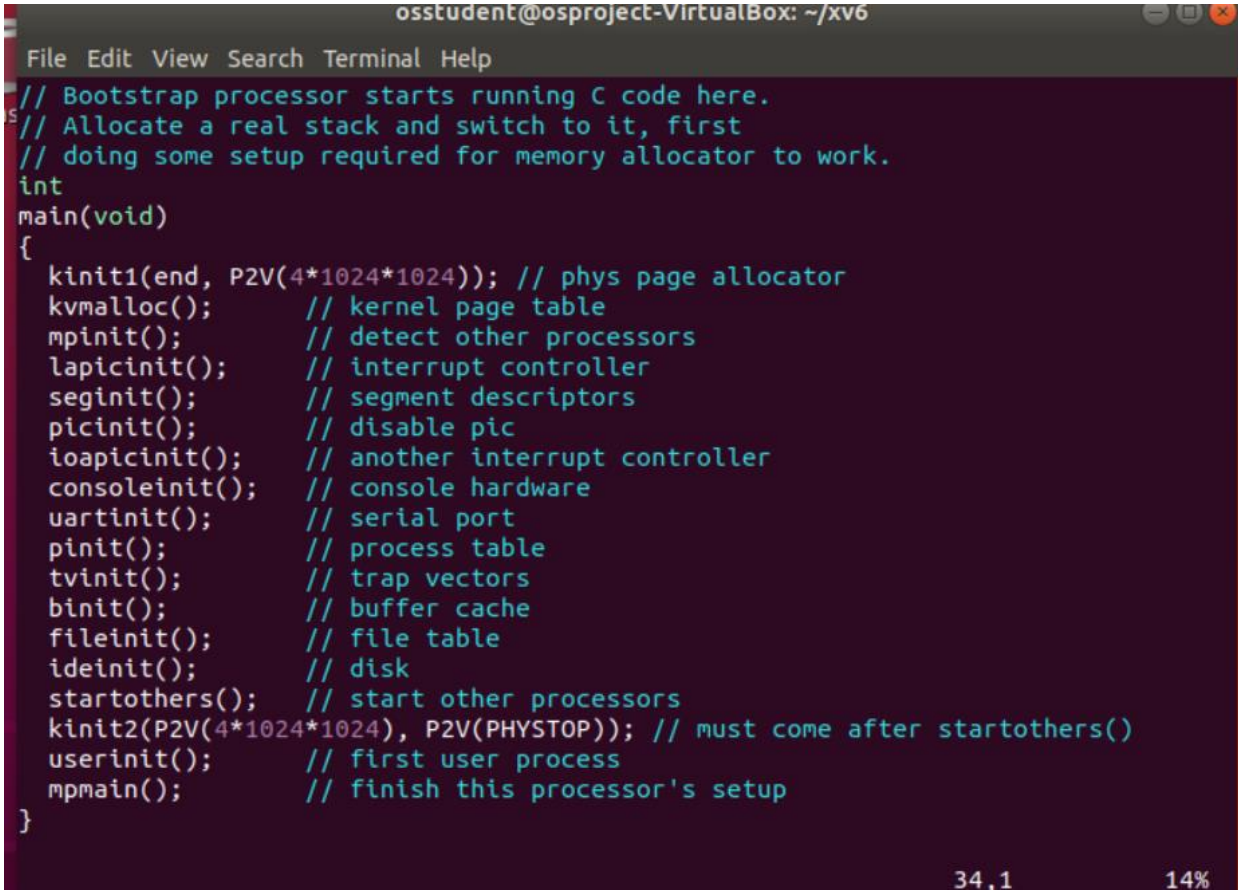
Once the **int** instruction is executed, the hardware takes over and does a bunch of work on behalf of the caller. One important thing the hardware does is to raise the *privilege level* of the CPU to kernel mode; on x86 this is usually means moving from a *CPL (Current Privilege Level)* of 3 (the level at which user applications run) to CPL 0 (in which the kernel runs). Yes, there are a couple of in-between privilege levels, but most systems do not make use of these.

The second important thing the hardware does is to transfer control to the *trap vectors* of the system. To enable the hardware to know what code to run when a particular trap occurs, the OS, when booting, must make sure to inform the hardware of the location of the code to run when such traps take place. This is done in **main.c** as follows:

osstudent@osproject-VirtualBox: ~/xv6

File Edit View Search Terminal Help

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"

static void startothers(void);
static void mpmain(void)  __attribute__((noreturn));
extern pde_t *kpgdir;
extern char end[]; // first address after kernel loaded from ELF file

// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
                                                    1,1           Top
```



```
osstudent@osproject-VirtualBox: ~/xv6
File  Edit  View  Search  Terminal  Help
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
                                                    34,1          14%
```

- Notice above, xv6 OS program starts (of course) in the main file (main.c) just like all programs. Here, many initializations occur including informing the hardware of the locations of where traps take place. Notice the line that says “trap vectors”:

Notes to self (trap.c)

- osstudent@osproject-VirtualBox: ~/xv6

  File Edit View Search Terminal Help

  ```
  #include "types.h"
  #include "defs.h"
  #include "param.h"
  #include "memlayout.h"
  #include "mmu.h"
  #include "proc.h"
  #include "x86.h"
  #include "traps.h"
  #include "spinlock.h"

  // Interrupt descriptor table (shared by all CPUs).
  struct gatedesc idt[256];
  extern uint vectors[];  // in vectors.S: array of 256 entry pointers
  struct spinlock tickslock;
  uint ticks;

  void
  tvinit(void)
  {
    int i;

    for(i = 0; i < 256; i++)
      SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

    initlock(&tickslock, "time");
  "trap.c" 112L, 2657C                                          1,17          Top
  ```

  - Above, in the trap.c file, each interrupt handler is initialized in the tvinit() function. Notice I goes from 0 to 256 because you can have up to 256 interrupts (each interrupt id value can be between 0 and 255 = 256 values when you include integer 0).
  - So when you make a system call, the initializer function shown above will tell which interrupt handler belongs to the systemcall function (which for xv6, it is 64); thus, the above function serves to make sure that the interrupt vector (vectors[i]) is initialized so that when we generate an interrupt, the system will know which interrupt handler to use.

- Note from the github website on the SETGATE function; set up interrupt table so correct interrupt handler will be called when there is an interrupt:

The **SETGATE()** macro is the relevant code here. It is used to set the **idt** array to point to the proper code to execute when various traps and interrupts occur. For system calls, the single **SETGATE()** call (which comes after the loop) is the one we're interested in. Here is what the macro does (as well as the gate descriptor it sets):

```
// Gate descriptors for interrupts and traps
struct gatedesc {
  uint off_15_0 : 16;   // low 16 bits of offset in segment
  uint cs : 16;         // code segment selector
  uint args : 5;        // # args, 0 for interrupt/trap gates
  uint rsv1 : 3;        // reserved(should be zero I guess)
  uint type : 4;        // type(STS_{TG,IG32,TG32})
  uint s : 1;           // must be 0 (system)
  uint dpl : 2;         // descriptor(meaning new) privilege level
  uint p : 1;           // Present
  uint off_31_16 : 16;  // high bits of offset in segment
};
```

- Remember after an interrupt from a system call, the current state of registers are saved so we don't lose the information about the user program that made the call; like we learned in class, interrupts cause a context switch (in system call case, to allow CPU to get kernel privileges and execute code from another function that requires kernel mode), and then the context will switch back to the calling function (and back to CPU user level privileges) when the system call makes a return. The state is saved via the struct trapframe shown below:

```
struct trapframe {
  // 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 es;
  ushort padding1;
  ushort ds;
  ushort padding2;
  uint trapno;
```

  - Note: the trap frame is also the kernel stack (a special running kernel stack).