**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

* Graphical user interface, text

  Description automatically generated
  + 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

* Text

  Description automatically generated
  + 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

* Text

  Description automatically generated
  + Above: showing the declaration and definition of the abstract data type: struct *rtcdate*.

# Task 4: Using the *grep* command

* Text

  Description automatically generated
  + 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

* A screenshot of a computer

  Description automatically generated
* Text

  Description automatically generated
  + 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:
    - Text

      Description automatically generated
    - 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:

Text

Description automatically generated

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

* Text

  Description automatically generated
* Text

  Description automatically generated
  + 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

* Text

  Description automatically generated
  + 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

* Text

  Description automatically generated
  + 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):
    - A screenshot of a computer

      Description automatically generated
    - 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:
    - Text

      Description automatically generated

# Task 9 (syscall.c): Add sys\_date(void) external function and add function pointer to sys\_date function in the syscall.c file

* Text

  Description automatically generated
  + 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.
* Text

  Description automatically generated
  + 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.
* Text

  Description automatically generated
  + 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

* Text

  Description automatically generated
  + 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

* Text

  Description automatically generated
* Text

  Description automatically generated
  + 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:
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + 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:
  + Text

    Description automatically generated
  + 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)

Graphical user interface, text

Description automatically generated

* 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)

Text

Description automatically generated

* + Text

    Description automatically generated
  + Text

    Description automatically generated
  + 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)

* Text

  Description automatically generated
  + 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:

Text

Description automatically generated

* 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:
  + Table

    Description automatically generated
  + Note: the trap frame is also the kernel stack (a special running kernel stack).