Team 1-D
05/07/2017

Known bugs

There are several known bugs that exist in our operating system that should be duly noted. The first is that of command input mismatching. If a user were to rapidly enter in commands such as asd, sdskf, sdoi, eijk, sdlk, in rapid succession, a subsequent valid command such as ‘dir’ may fail. We believe this is because of a mismatch between the shell program handling the interrupt that prints out ‘Bad Command!’ and polling for input on the subsequent key interrupt. Note we have not been able to formally replicate this bug, and may not still exist. Resolution: in order to fix this bug, simply restart the OS.

The shell’s ‘dir’ command has a known limitation with regards to sector size. As it stands, there cannot be more than 99 sectors for the dir command to execute successfully. This exists because the character array is statically allocated so many bits, which only allocates up to two character slots for displaying the size of a file as a number of sectors in the current directory.

Special features

We elected to augment our shell’s ‘dir’ command by including the number of sectors when listing the files in a directory. There are no special instructions for use outside of the default dir usage. However, you should see the number of sectors printed out in a right justified column next to each displayed file. For further details on dir’s implementation, see the below section.

We also implemented the clear screen command, which writes over the video memory with empty characters and resets the cursor to the top of the display window. This behavior is intended to mirror commands such as ‘cls’ in the Windows terminal and ‘clear’ in Linux systems. Note that since there is no command history stored in our shell, the previous display is lost permanently. The clear screen command does not reset the instance of the shell, and so background processes, the background and foreground colors, and so on will continue unmodified. In order to use the clear screen command, simply type in ‘cls’ and enter Return.

In addition to clear screen, we added further support for some of the BIOS interrupt 10h function calls by adding the scroll up window (AH = 06h). More specifically, we statically handle parameters AL, CH, CL, DH, and DL to clear the screen and maintain the terminal size, but allow for user input for BH, which determines background color from the top four bits and foreground color referencing the bottom four bits. Supported characters are listed in the BIOS Color Attributes Page and usage is as follows: cfg <foreground hex value> <backgroun hex value>.

The final special feature we chose to implement was exit. This will exit out of the foreground process, which by default is the first instantiation of the shell.

Interesting or clever implementation techniques

As we developed this project, we ran across several instances of basic functionality consuming a large number of lines both in shell and kernel. One particular offender were generic string operations. As a result, we elected to implement our own version of “strings.h” which included several standard string-based functions, such as strlen, strncmp, and so on. This prevented us from bloating our code with overly verbose string comparisons and allowed us to provide a standard interface to adhere to when working with our strings.

Another highlight of our project was a slight modification to the readFile function. We originally followed the spec but ran into several instances where we wanted a little bit more information on whether the file read succeeded. For instance, the distinction between a File not Found error and desiring a file not exist when making a new file from the shell could not be natively handled. For that reason, we added a return value of an integer, which allowed us to check if the file was found or not from the perspective of the caller. This was a standard return schema of 0 on success, -1 on error. Additionally, we added a third argument: int notDebug. This bit let us flip whether we desired the file to not exist, such as in the instance of unique naming conventions in a directory, or if we should desire the file exist, such as loading in a program to execute. This bit of information let us decide whether to print debug strings such as “File not found!” or “File already exists!”

A third design choice we made was how we handled printing out the directory’s contents using the dir command in our shell. We found counting the number of sectors to be trivially easy, as all we had to do was loop through the labeled sectors of our directory file, but when printing these counts we needed some way to support a standard atoi() not found using bcc. As a result, we were able to add 0x30 + our integer and store that in a char array. One benefit of this approach is each line would be relatively static: the filename would be at most six characters and the number of sectors would range from 1 to 26. Because of this predictable behavior, we were able to pre-instantiate an array with equal amounts of white space before doing our atoi() conversion, writing over the white space if an integer were to occupy the 10’s digit. This naturally lended itself to a very nice right justified display of the directory’s contents.

Lessons learned

Yuzong Gao

I learned lots of valuable lessons from working as a team in this project. First, it is always good to plan things ahead. We started each milestone very early, and even tried to work on the next

milestone although its deadline is quite late. Planning ahead gives us a lot of flexibility and time to do extra credit in milestone 5.

Another thing that I benefit from the group is learning from each other's coding behavior. We did a lot of pair programming to make sure everyone is on the same page. And it also enable us to observe each other's way of thinking through a problem. Personally I learned a lot from Zhou Zhou's clean coding style.

At last, I learned that it's important to keep communicating with teammates. We frequently ask each other's process in facebook group chat, and if anyone has any bugs in his/her code, all of us will try to help, so it guarantees the efficiency of our work.

Jake Patterson

While working as a team of four, I learned quite a number of important logistical lessons. Firstly, if the work can be easily split up into major components, it is almost always worth spending a couple of hours planning out a course of action and partitioning the work into manageable and logical chunks. When we did so, it allowed for us to split off into pairs of two, implement our particular deliverable, and reconvene to review and demonstrate the code we wrote.

I also found meeting and being in physical proximity to my team members to be very beneficial. Even if one of us didn't have a task at the time, being in the environment contributed to our overall camaraderie, work ethic, and success as a unit.

Lastly, we met early and communicated our progress often. We were very honest with how much work we had accomplished. This was very much a beneficial aspect of our working together as a team, as it prevented poor implementation from a panicked student who had put off and then tried to salvage one of their assignments at the last minute. It allowed us to cover for our mistakes and learn from one another in ways you don't typically see with teams that would otherwise procrastinate and fail to communicate.

Luwen Zhang

I could not have asked for a better team to work on this project. Everyone is smart but humble and willing to share their knowledge with the rest of the team. We worked well when coding together. Usually one person types and other three people look for potential bugs and mistakes which minimizes the amount of time to debug later. When we ran into a bug that no one knew what went wrong, each of us would check out the project and run tests to try to fix the bug.

By doing 4-people pair programming, we did not really have many merge conflicts, which also saved some time. Since we always work together and update on progress, everyone was on the same page. If something does not work, one has enough knowledge to examine our code base for errors.

Zhou Zhou

Throughout the span of this project, I was grateful for the responsiveness and activeness of everybody on the project team. There was not a time when nobody would stand out to take charge of some components of the milestones or be not motivate themselves to produce satisfactory work. Without a motivated team, none of what I described below would actually work.

In this project experience, we attempted an entirely new meeting approach. For certain sequential parts of the milestone, we did four-people “pair” programming. One of us would be typing the code on the screen, without three others monitoring the code for potential errors, as well as think of solution whenever the coder gets stuck. This way, it saves the effort to synchronize each team member’s knowledge regarding basic design decisions in addition to system constraints. After the codes are committed, one of the observers would write the Readme documents, as well as committing the files to svn. This happened more often in the earlier milestones, where instructions were mostly sequential.

Later on, especially during milestone 5, we divided the team into groups of 2, so each group can perform pair programming on individual portions of the projects. The reason it applies well to milestone 5 is because its optional feature requirements are mostly separate from the main requirements of the project itself, making it ideal for parallel development. While this is a rather traditional approach, it benefit from earlier practises where the group observe one person coding, since each pairs now knows enough to finish their components.

Technical things learned

Yuzong Gao

Through each milestone of this project, I was quite surprised by the frequent use of interrupts, and how omnipotent it is. Starting from milestone 2, our handle interrupt function keeps growing and is able to handle more and more utilities. In later milestones, I was quite surprised that shell can perfectly does its job just by calling interrupts. I think this helps me better understanding the design of OS.

I also learned that how useful the system library is in C. I was very frustrated when I tried to debug by attempting to print some number. I had to create a char array and manually put 0x30+number into the array. And if the number is greater than 10, I’ll have to put each digit into char array, which is really painful.

Working with BCC is a little bit annoying. We have to initialize variables at the beginning of each function. And it's even more annoying if you want to print some messages. Luckily we had the stringMaker in examples to generate strings to make our life easier.

Jake Patterson

Prior to starting this project, I hadn't really considered that the switchboard of processes supported by an OS could be routed through an interrupt. The ax register providing an array of bits through which the particular function call made sense, but the 0x21 interrupt component in particular served to be quite interesting. Additionally, BCC being so limited in scope took me by surprise; lacking div and mod in particular was a bit of a scope adjustment to make. Finally, kind of a quirky issue was that of the carriage return character. I've only ever tangentially been bothered by differences between newlines across linux and windows, but having to include \r\n\0 for our lines firsthand allowed me to better understand the disparity between the pointer's location and a strict understanding of a line's contents.

It was also interesting how much of our code read through an entire max file size to load in a program of some size. That, in tandem with us using one directory pointing to one group of sectors to be read in on our image were both revealing of how much more aliasing, indexing, and partitioning could be done for larger file systems.

Luwen Zhang

I would have never thought that I could implement a simple OS with some basic commands before this class. When learning about char arrays in C, we were told that a special character is used to indicate the end of the string. In this project, we had to actually add '\0' to the end of the array despite the fact that we are used to having gcc handling this kind of things. Therefore, I appreciate gcc much more now than ever.

Double quoted strings gave us a hard time for m3. Everything seemed to be fine, but it just would not work. After reading the instructions again, we realized that the kernel code not use any kernel global variables and any double quoted strings are stored as globals, even they are declared in a function. I would have never thought about that if I did not see the warning in the instruction.

Zhou Zhou

Before I took this class, I had little idea that the system environment in which I code affects how I code programs. Then this project introduced me to a practical environment where many things can't be assumed although it's still called C. For example, in the processor project, we were told that global strings are highly discouraged because they take up the global space for variables and the space doesn't get reused. At first, we decided to ignore this problem, until we hit milestone 2 and the project starts to experience unfathomable failures. Then someone on the

team decided to convert the strings because he saw the piazza post, not knowing that would fix our code at all. It came to that after he converted all the strings, the project suddenly worked, and we moved on knowing we should make strings local everywhere in the project.

I've never appreciated system library that much, until we decide to make our own little string library to use across the project. It looked straight forward, but things get harder when we realized we had to handle wrong inputs in the code. For example, we first implemented strlen, strncmp and strncpy with naive approaches such as looking for '\0' as the string terminator. We then turned to use them in our project, and then realized there were edge cases such as unterminated strings. Then we had to use certain hacks to make the strings terminate so it wouldn't break the library. We can only imagine how the equivalent functions could be implemented by Linux authors and other implementers of the UNIX library, where they needed to be way more robust so the programmers don't need to know the inner workings and apply hacks like we had to in our project.

The restriction posted by BCC and the environment in general imposed several difficulties on us, such as no compiler errors when arguments don't match the function signature, and so on, but I'm glad we've made through it and seen a lot of similar problems that could as well occur in the professional environment. This experience would definitely prove useful when we go into work.