Reflection

Known Bugs

Running two programs that both take keyboard input crashes very quickly (e.g. running shell from inside shell).

Backspacing does not update the display across multiple lines, but still updates the command buffer.

Entering more than 10 space separated arguments in the shell will almost always result in an invalid command, since the address of the 11th argument in the argv overwrites the argc counter.

Special Features

Clear screen command - Clears the screen, and resets background color and display page. Used by typing “clear” in the shell.

Background color - Changes the background color to one of 16 values, represented as a hex character (0-F). Used by typing “bgcol <color>”, e.g. “bgcol 4”.

Display paging - Allows the shell to maintain separate “windows” that maintain unique output, however all printing occurs on the active page. Used by typing “pg1” or “pg2”.

Graphics - A simple graphics sandbox, where you can draw “pixels” with the spacebar, change color with the number keys and a-f, change the cursor size with + and -, and move the cursor with the arrow keys or ijkl. Started with the “draw” command, and exits to shell when you press escape.

Command line arguments - Command line arguments can be passed to executables. A simple test program was made for this, printargs (shortened to printa in the OS), which will print argc and the argv array that it is passed. Can be tested with “execute printa [args]”, though passing more than 10 total arguments, including execute, will result in an invalid command.

Command tab completion - Searches a list of known shell commands and matches with the current buffer, printing all matches, or completing the command if only one match. Pressing tab when the prompt is empty will print all known commands.

Command history - Stores the last 20 commands, including the current command, and history can be browsed/used with the up and down arrow keys. Previous commands can be reused and added onto, but this will not change the entry in the history. Only the initial prompt will save its value when cycling through the history.

Exit - exits the shell, and additionally shuts down qemu when all processes have terminated.

Interesting Techniques

One technique that we explored during the project was #define macros. We quickly got tired of having to remember the interrupt numbers for each function, so we created define statements so we could make calls like this: interrupt(PRINTSTRING, “stuff”, 0, 0). As the project evolved, we explored just how much we could do with macros, and eventually discovered function-like macros, which allowed us to pass arguments into the macro and populate the interrupt with that. This in allowed us to entirely eliminate the interrupt function from the code we typed, while still turning into the same thing at compile time, and not adding any additional overhead from calling actual functions. In the interest of condensing our shell and kernel code, we defined our macros and some of our heavily used functions in a separate file “syscall.h”.

Lessons Learned

(Trey)

My biggest takeaway from this project experience is that when you start working on tasks well in advance, and allow plenty of time to analyze and come up with a good approach to solve problems, the quality of work done is much better than trying to beat a deadline in one sitting. This is the first project group I have been a part of where the group successfully started and completed each milestone multiple days in advance. The results of our work showed that this is an effective method to complete group projects successfully. I also had a positive experience with pair programming. We pair programmed every milestone, and the efficiency in which the work was done really stood out to me versus how some of my previous projects have gone in the past. I also think the way the milestones were broken up helped in understanding the material ahead of putting together the entire operating system and made it easier to be successful with the project. Overall, this has been one of the best project experiences in my time here at Rose.

As for the technical aspect, I came into the project with no experience in writing code for an operating system and only minimal exposure to how an OS works, so I learned about the core aspects of how an OS operates. The operating system uses interrupts for basically every interaction between the OS and the user. This could be seen from milestone 2 (where interrupts were used to write to and read from the console) onward. I also learned that the data for files are stored in different sectors on the disk, and to execute a program the files must be loaded from the disk and transferred into memory to be executed. The shell that we created during the project makes it easier for the user to execute command such as writing and deleting a file, executing a program, or showing the directory contents. Finally, I learned how basic multitasking works. The page table entries for each process involved in the multitasking must be set appropriately to allow for a process to run and block any other process from being scheduled until the process relinquished the CPU. Overall, my understanding of operating systems and how they work have been enhanced greatly after having done this project.

(Chris)

My biggest lesson learned from this project is the amount of control you can have when dealing with low level languages and assembly code. I am a huge fan of higher level languages like Python, Java, C#, and sometimes Javascript. I'm used to seemingly insane amounts of abstractions of logic when solving problems. It's great for implementing fun personal projects, or school projects, but that kind of thinking does not help when writing rudimentary code for an operating system. Writing in C was mostly review (except for threading), as I had taken CSSE132 previously, but having to write almost everything from scratch really showed me how much you can take into your own hands when designing systems. You don't have to adhere to existing standards in common APIs or frameworks--you write the specs yourself. It's quite a different perspective, and while frustrating to adapt to at times, it was a worthwhile experience.

From the technical aspect of the project, I learned more about kernel interrupts and just how much can be accomplished through them. I knew about hardware interrupts from keyboards and other USB devices, but I was not aware how many types of interrupts existed. I learned about the types of process queueing first during class, but implementing our own scheduling system really reinforced that knowledge well. In hindsight, I wished I had utilized a multi-level feedback queue, but that was extremely low on our list of priorities.

(Zac)

Personally, I thought this project was a great opportunity to take the concepts presented in class and really learn them and apply them to a functional project. It's easy to understand what a scheduler does, but significantly harder to implement even a rudimentary scheduler. I also learned/realized just how much of computing works just because that's the way we decided it will. Like command line arguments, which from the command line just looks like an alternate syntax for calling a function, but from a process perspective, aren't called like a function at all. But because it's defined that those arguments show up at a certain place in the process memory, we can just directly put the arguments there and trust that the process will check that address if it wants to use the arguments.

I think the most important technical detail that I learned while working on this project is just how much you can finagle things in C and assembly. I've had a fairly decent grasp on the concept of pointers for a while now, but have always been able to work comfortably in the abstraction layer of "Oh, this is an address but I don't need to know what the value is." While working on command line arguments, I had to dig into not only finding and setting argc, but building the entire argv array from scratch. I think this really helped cement how pointers work, and how arrays are represented.