On Writing a ComputerCraft OS

Jack Bruienne
Michigan State University
bruienne@msu.edu

Abstract

One of the most common projects I've seen for ComputerCraft is to write an operating system. People look at the limited command-line interface that CraftOS provides, and think, "I want this to work like my normal computer does!" Time and time again, a new post pops up on the ComputerCraft forums or Discord either announcing an OS, or asking for help with an OS, or releasing an OS. Usually, there are some very obvious flaws in these "OS"es, ranging from poor design choices, to overstating what they are and underdelivering. There are many common misunderstandings and undersights that newbie developers run into when writing an operating system, and these end up creating mediocre products at best.

1 A Critical Distinction

The term "OS" is thrown around a lot, and in my opinion it's very overused. According to Wikipedia: "An operating system (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs" [7]. However, much of this is not really possible to accomplish in ComputerCraft. Hardware (peripherals) is automatically handled inside the mod through the `peripheral` API, and hardware is exposed through a simple object-oriented-like interface. Contrast this to a real system, in which hardware has to be memory mapped or controlled through arbitrary I/O ports; components can have various interfaces that are not compatible with other versions of components of the same type; and access has to be abstracted for effective use. ComputerCraft's simplistic peripheral interfacing API removes most of the need to manage hardware manually. CraftOS does add in some extra code to allow accessing networked peripherals through the base `peripheral` API, but other than that, what the system already provides is enough for pretty much all use cases.

In addition, since CC uses Lua, a dynamically-allocated, garbage collected language, memory allocation is handled by the internals of the language, and the user does not have to, and cannot, manually manage these resources

(outside of `collectgarbage`, which is not exposed in standard CC). Input and output operations are all handled through the event system, and there is no such thing as probing of input devices or setting up interrupts - every event that could be triggered externally is handled through the coroutine layer, and inputs are automatically translated into a usable format for the computer.

Software resources are slightly more representable in ComputerCraft. The filesystem provided by CC is pretty basic, only giving access to the raw filesystem representation on disk, and not supporting more advanced features such as file permissions and mounts (besides disk drives, which are automatically handled anyway). The terminal interface is also simplistic, and does not natively handle things such as newlines or control codes. CraftOS supplements this by providing `write` and `print` functions that handle newlines and the ability to redirect the terminal output to another output "device"/object. Process control is notably missing from CraftOS, since Lua is a natively single-tasking language (though advanced computers do have `multishell`, which lets you run multiple programs in separate tabs of the shell). However, the existence of coroutines makes it possible to have cooperative multitasking, and through debug hooks it's even possible to set up preemptive multitasking. Even if these services are provided by ComputerCraft, an operating system could override these services and add extra functionality that is missing from the base APIs.

The main part of an operating system that makes sense on ComputerCraft is the presence of "common services for computer programs" - or, as ComputerCraft refers to them, APIs. CraftOS comes with a number of APIs built-in that are loaded on boot, providing services such as value serialization and unserialization, simple painting functions, and a universal settings database. Other OSes may provide extra functionality, including cryptography, UI libraries, automatic event handling, etc. These sorts of APIs can be very useful for developers, and I'd argue this is the most valuable part of a ComputerCraft OS.

However, in practice, many programs billed as "operating systems" provide *none* of these services. The main goal of developers who make these is to create a desktop environment. They rely on the services provided by CraftOS, and do not extend from it further, besides possibly adding their own globals to access the UI components. These are not operating systems. These are called **shells**, and should be distinct from actual OSes. A shell is a program that just provides the services of the base OS (in this case, CraftOS) in a human-usable form. The term "shell" is usually used to refer to command-line applications, such as, well,

`shell`. But GUI applications can be considered shells too! Usually, a GUI shell is split into multiple components: a window manager, a desktop manager, a file browser, etc. However, it's still completely valid for a shell to be one application that provides a UI.

There's also a large number of "OS"es that actually *are* shells - no nitpicking required. These are just replacements for the default CraftOS shell that have a dollar sign instead of a bracket, and maybe some extra programs to make it feel more "Unix-like". At least for GUI "OS"es, a different user interface is implemented - CLI "OS"es are pretty much always shells. (There are some that are not, but we'll get to that later.)

If you decide to make an "OS", consider whether it's actually an operating system. Does it change how the computer is interfaced with? Does it provide any extra APIs/libraries for programs to use? Does it do anything besides change how the user interacts with the computer? If not, your program is a *shell*, not an OS, and it should not be marketed as such.

2 You Are Not Original

If you think you've got some neat idea for an OS that's never been done before: throw that thinking in the garbage. If you can think of it, it's (more likely than not) been done before by someone else. Don't start an OS project thinking that it will be special in some way. Uniqueness in an OS is determined by a) its themeing of user interface, and b) the set of features it provides. There's tens to hundreds of "OS"es that provide a user login system - if your selling point is that it has multiuser support, reconsider how you're advertising the OS.

2.1 Linux Clones Are Overdone

Adding on to the previous part, one particular type of OS that has been done way too much are Linux "clones". These attempt to make a shell that looks like Bash, and add programs that are named like GNU programs. However, in 99% of these "OS"es, they *do not* actually clone anything else about Linux, and in most cases they're just reskins of CraftOS. Having two-letter programs, following the FHS, and putting a dollar sign in the shell does not make your "OS" (read: shell) anywhere near Linux.

For an OS to be an actual Linux clone, it would need to have things such as preemptive multitasking, kernel-userspace separation, a syscall layer, virtual filesystem mounts, a full permission system (possibly including ACLs), multiple users, etc. And this is just kernel features - for a proper GNU/Linux userspace you'd need to implement many other utilities that would be expected in `/bin`, `/sbin`, etc., as well as a proper Bourne shell (or any other POSIX shell) that supports scripting. This is a big task, and the type of developer that would naively write a "Linux clone" would likely not

have the skills to properly accomplish this. They'd probably end up with a mediocre shell that pretends to look like Linux - just like the rest of the Linux clones on the CC forums.

Do note that it *is* possible to pull off a proper *nix-like OS. VorbaniOS is one that I've seen that appears to do it right. This has many of the features I described above, but it's quite a large project, and would likely take a developer months to get into a properly working state, let alone complete.

3 Writing An OS Is Hard

Taking on an operating system project requires a lot of time to complete. Since 2014, I have started multiple OS projects thinking that I would be able to get them finished. Perhaps this is a personal issue of motivation, but I believe this is more because I think of a cool idea, but once I get started on it I realize how much time it'll actually take.

To make an OS (and not a shell as previously described), a lot of components need to be assembled. The kernel is the largest part, and you'll likely spend a lot of time working on this, depending on what kinds of features the kernel has. (If the kernel just provides process management, then it won't take long - but you won't have a very good base to build on.) The kernel is also the first thing to work on, since everything builds upon it. Next is adding user libraries, which will consist of UI libraries (if present) and other utilities you'd want to provide to program. The other big portion (at least for OSes with a GUI) will be writing a proper windowing system, which will need to be able to support connections from multiple processes. Finally, you'll need to write programs that can run on your OS, as having an empty desktop or shell won't really be interesting to users.

All of this will require writing tens of thousands of lines of code across many files. In addition to writing code, a significant portion of time will also be spent planning the OS - figuring out how the kernel works, what kinds of services it will provide, what interfaces will be provided to programs, how to manage windows, etc. It might actually take longer to plan the OS than to actually write it. While writing code, you'll likely also realize that some of your earlier decisions were bad, and you want to change how something works. This will take time to rewrite whatever you want to change, as well as rearchitecting anything else that may be affected by the change.

I see very little of this actually done by developers of "OS"es - they just jump in and start writing code, but then they either drop or pause the project, or kind of throw whatever they have together so it looks like it works, but a lot of things are broken or missing. Before you start writing an OS, consider the amount of time it will take. Do you have the

motivation to keep working on it, and will you continue to be motivated until it's done? Do you have the time to work on it? How committed are you to having it done in a reasonable amount of time? I've had a very loose idea of an OS for over two years that I've scrapped and rethought multiple times. I know I don't have the attention span to keep working on it continuously, and I don't have any sort of expectation on when I'll have anything done. Consider your ambition for completion before starting.

4 Do You Really Need A GUI? (aka Why?)

One of the most common features of "OS"es is a GUI, and many users of ComputerCraft have complained about the lack of a windowing system. Therefore, the next logical step for those users is "Well, I'll just make my own!" As stated above, they often underestimate the requirements of such a system, which requires multiprocessing, window management, layering, mouse hit detection, user interface libraries, and much more. But the real question to ask is, "Why do I want a GUI?"

I've seen a lot of users say they want their ComputerCraft computer to work like their real computer. But those users are missing the point of the mod itself. CC's goal is to provide a fantasy terminal that is similar to 80s microcomputers without the complexity of programming that they often had (though BASIC was pretty, well, basic). Even with this goal, CC is quite overpowered for this goal, and it already supports much more than any of those old computers could, even without any sort of graphics mode. Trying to get something like Windows 10 working in CC kind of defeats the point of the limitations. Certainly, it's cool to see stuff like this actually working in such an environment, but arguably it would be easier to use the computer in a basic CLI shell. Perhaps users that have never touched the command line would have an easier time, but in the end it may be better for them to just bite the bullet and learn. (At least CC's design is much easier to get used to than a plain Linux shell.)

Due to the limitations of CC (notably, the lack of proper graphics), it's not very easy to make a proper GUI. At the very least, you end up with a small 51x19 display area to try to show graphical content. If you use the drawing characters from `\x80-\x9F`, at least you get more resolution and square pixels - but with the caveat that you can only have two colors in any 2x3 area. Navigating this requires some smart techniques and algorithms that are beyond the typical type of user that would want to write a desktop environment.

Desktop-like GUIs aren't the only imitation that users desire - as previously stated, a lot of users try to make Linux "clones" because that's

the only CLI they know. Just because a system has a CLI doesn't mean it has to work exactly like popular systems. Sticking to the design of some other system often causes there to be limitations in the design of what you're writing. This is especially true when comparing Linux, which is written in C and compiled to machine code, with CC, which uses Lua and has a lot more reflection and metaprogramming capabilities than compiled languages like C. Instead of trying to make a clone of something, try making something completely new. Use the benefits of the language you're writing with to your advantage! This will make your system much more robust, as programs will be able to use the system to the fullest.

Consider how important it is that you have a GUI. Is it worth the months it will likely take? Critically, do you know enough about how a desktop works that you can reasonably implement one yourself? There's many GUIs out there already too that will probably be better than yours - my favorite one recently being LevelOS [4], which does the desktop thing right (however, it's got a messy codebase, and it could be better designed - but it works!).

5 Just Do Something Else

As fun as it is to have an OS on your computer, there are many other things that could be more useful for ComputerCraft. And the types of programs you might include with an OS can, more often than not, be as useful *or more useful* than if they were only available in the OS. If you're new to programming in CC (or in general), you will pretty much always not be experienced enough to pursue an OS project. If you do have the experience, there are other types of programs that will be more useful. Try making something that works in CraftOS first, as every user of CC will be able to use it, and not just those who have your OS.

For new developers, turtle programs are likely the best first step for getting used to Lua and ComputerCraft. They're pretty simple to navigate, and tasks like moving a distance will introduce you to loops and other constructs. Make a farming script that automatically harvests a plot, collects the crops, replants it, and puts the crops in a chest. Or set up an automatic miner that can automatically collect only ores and returns to a chest when full. If you have a modpack, make an automatic reactor controller script, possibly with a cool UI for touch control on monitors. Try to make something that improves your or someone else's survival gameplay instead of something that's just going to be eye candy at best.

If you're more experienced with CC, and maybe not as interested in the Minecraft side of it (like me), try using the resources available

in CC and push them to the limit. Make a music player that automatically manages speakers and can run outside of the locked tick speed of CC - maybe even with stereo support. Create a package manager (of which there are also many) that can handle deep dependency resolution and conflicts, with expandable repository support. My best recommendation is to make a cool game, of which I have not seen very many of. I'm still waiting for someone to create a game using CraftOS-PC's graphics mode [3], so if you're interested, try giving it a shot - I'll promote the first good one on the front page. (UPDATE: This promotion has now been taken, but I still encourage you to make a cool graphics mode game!)

There's plenty of other projects you could try making that do not have the complexity of an operating system, and are much more useful. Before diving head-first into an OS, make something that's actually useful for other people. It'll help the community much more than yet another sloppily-built UI system.

6 If You Say So...

To be frank, there are some valid reasons you might want to build an operating system. If you aren't concerned about quality, don't care about releasing it, and just want to give it a shot, then try it out! If it's an epic failure, so be it, and if you end up with a great product, then congratulations. In addition, if you know how to structure an operating system, have the skills required, and can stick to a goal, then making an OS is a fine long-term project. Most of my worries are about people who just think it up and want to jump right in without any preparations. Honestly, a lot of those people are just kids, who like Minecraft, like computers, and have big dreams about the kinds of things they want to make. I don't necessarily want to rain on their parade, but they are trying to compete in a market with those who are much older and wiser than they are, and if they continue without having a reality check, they will get a reality check in the form of comments on a forum post. I'd hope that those comments aren't hateful, but you can never really be sure about it.

If, after reading all of my demotivational ramblings, you still want to try making an OS, there are a few basic rules you should follow to make sure it doesn't suck like most of the other "OS"es out there.

6.1 Give It A Structure

For an OS project to be successful, it has to have a structure to it. Having multiple different modules for each part of the system is a must for a proper OS. Keep code that is not related separate, so do not have your scroll bar code inside the kernel (unlike Windows [2]). If using source control like Git (which I **strongly suggest**), place each module in a different

repository so you can keep track of each thing individually. Having organization in your code keeps things easy to maintain so you can find code when you need to access it.

6.2 Have A Plan

Just jumping into an OS without actually knowing what you're doing is a 100% guaranteed path to failure. Given the size of an operating system, there's a lot of details that you need to be sure about before you even start writing code. The interfaces provided by each module (and not just user interfaces, but programming interfaces) need to be set in stone before you can move on to dependent modules; otherwise, you end up going back and forth between modules trying to figure out how things work and what things are supposed to do. This is a big time suck, and will prolong the development process greatly. Think of how you want to complete each part before actually starting, as this will remove much hassle from writing the code.

6.3 Don't Build The Roof First

A common mistake I see from "OS" developers is that they start writing the first thing they want to see from their "OS" - the user interface. They write the windowing system first without even thinking about multiprocessing, libraries, etc., taking a do-it-as-you-go approach. This is not an effective programming strategy. Doing this leaves you with spaghetti code that does... something...?, and lots of dead code that you wrote for some purpose before, but don't actually need anymore and now it's just kind of sitting there. This strategy also leads to breaking best practices and structure, as you need to go back and forth to build the floor below you - eventually leading to the roof collapsing under the weak walls that were built to support the things you needed from the system.

6.4 Don't Pretend To Be Cool

So many new "OS" developers want to make their "OS" look cool, but without doing any of the work. They accomplish this by adding fake loading screens and pauses to make it look like the "OS" is doing something. ***Don't do this.*** Fake loading screens and the like may look cool to you, but they cripple usability and make you look like a fool when the user realizes their time is being wasted for the purpose of "realism". ComputerCraft OSes should pretty much never have any sort of loading time, since you have the full speed of the server's CPU, memory, and disk, but you're limited to 1 MB of disk space. (Some tasks do take a lot of processing, though; and HTTP requests can take a long time too.) In short, if your code contains even a single `sleep` call, it is a waste of time.

6.5 Think About The User (And Developer)

An effective OS should be easy to use, not only for the end user, but also for developers who want to write programs for your OS. Having a beautiful UI will draw people in, but poor UX will push them away. Make sure everything in your OS is understandable for users so they don't pull their hair out trying to figure out how things work. Label things well, use proper English (for OSes that target English-speaking audiences), and test every part of your OS to make sure it makes sense to you. If possible, have other people try it out before releasing, and get feedback on what strengths and weaknesses your interface has. The US government has a website at usability.gov [1] that has tips on how to improve user experience. (Even though this targets websites, it can still be a good resource for other types of interfaces.)

Ensuring developers can use your OS's services is important if you want to have an ecosystem of programs. A poor structure will hurt developer usability, but having hard-to-understand code or a lack of documentation will surely cripple anyone who might want to make applications. Providing documentation for every API exposed to the developer is a crucial requirement for developer usability. Otherwise, the developer will have to pick through your code to figure out what things mean. Document not only what APIs do, but also what file formats they use, how they work, and give tips and example code on using the APIs. Think of each API exposed in the OS as if it were independent, and the developer did not know about the rest of the OS. If different APIs interact with each other/the kernel, provide links to documentation on those parts. The developer's experience is as important as the user's experience as long as you want to allow growth on your platform.

6.6 Consider Ditching CraftOS

One common failure point I see in various "OS"es and libraries/expansions is that they try to add a bunch of new and useful features while also sticking to the CraftOS APIs. Unfortunately, due to CraftOS's (poor) API design, a lot of things that may seem simple to add are actually not that simple. Take, for example, setting load paths for `require`. In normal Lua, this is as simple as setting `package.path`; however, since the shell creates a new copy of `package.path` and `require` that does not inherit old settings, and that any changes you make are discarded on program exit, there is no viable way to add a folder to the path without some *really* dirty hacks. Another thing I see is an attempt to add file permissions to `fs`, which is something I've done myself [5], and it often ends up with more dirty code that also breaks API compatibility.

If you really want to make an operating system that is well-designed, you should not try to keep CraftOS compatibility. Creating a new API based around the features you want to support in your OS will make the design of the system *so* much more clean and stable (as long as you design it well enough). Even though this will limit the number of pre-existing programs you can use, you can always write a shim library to convert CraftOS API calls to your OS's calls - or even use something like YellowBox [6] to create a virtual machine with actual CraftOS. I'm following this design strategy in my upcoming operating system, which at the time of writing is TBA.

6.7 Follow Best Practices

Just like there's ettiquete in social interactions, there's a set of best practices that all programmers should follow when writing programs, including when making an operating system. In particular, ComputerCraft coding has certain rules that should be followed to avoid falling into specific pitfalls in CraftOS's design.

- Use local variables as much as possible, including local functions. You should avoid global variables as much as possible, as they are slower, aren't automatically garbage collected, and can cause funky errors when other code uses the same variable name. A good OS design will not use any global variables except for a single root namespace where all kernel services are provided - or in some cases, no globals, instead using the coroutine boundary (syscalls) to transfer data.
- Indent your code! The rule of thumb is that every time you write `then`, `do`, `function`, `repeat`, or `{`, increase the indentation level by 1, and when you write `end`, `until`, or `}`, decrease the level by 1. Your individual code style may dictate some exceptions, but this general rule should always be followed. Not indenting code makes it hard to read, like adding random line breaks or indentation in the middle of a paragraph. Being hard to read not only affects other people looking at your code, but yourself, as you traverse a large file and have to figure out where a block begins and ends.
- Give functions and variables descriptive names. For example, `button.position.x` is much more understandable than `b.px`. This not helps other people understand how things work, but also helps you understand what things do a month from now when you forget how everything works. Some things, like iterator variables, may be alright to shorten, but remember that less ambiguity will make it easier to

understand what the meaning of the variable actually is.

- Avoid `os.loadAPI` - use `require` instead. `os.loadAPI` is deprecated, creates unnecessary globals, and does not work with relative paths. It also does funky things with the environment of the API, and can cause a lot of things to break. You should replace all usages of `os.loadAPI` with `require`, which is much better and is compatible with normal Lua. To summarize how to do this, simply return a table of functions at the end of your API to make it compatible with `require` (and also convert any global functions to locals), and use `local myAPI = require "myAPI"` to load it.

7 Conclusion

Making a ComputerCraft operating system is one of the most popular projects in CC, but most of these "OS"es are bad for many reasons; some of those reasons being poor design, a lack of structure, or just plain not being an OS, but a shell. New users think of a cool idea that they want to try (an OS), not realizing the scope of the project and the kind of dedication it actually requires. Some just want to be able to have a system like their normal computer, but they never really question why they want it, and they don't find the proper OSes out there that accomplish this goal without them having to write a single line. And for those who just want to go ahead anyway, a list of guidelines that developers should follow when making an OS will keep them from falling into the same traps that other "OS" developers fall into. If you want to make an OS in ComputerCraft, think about how much you actually want to dedicate into writing it, and what your exact goals are. More often than not, it may be better for novice users to use an already existing pre-built system like LevelOS or Opus OS, or - hear me out - just plain old CraftOS.

References

[1] Affairs, A. S. for P. (2013, July 18). Usability.gov. Retrieved November 14, 2022, from https://www.usability.gov/

[2] Haglund, F., & Anders. (2017, January 1). *Why does windows handle scrollbars in kernel?* Stack Overflow. Retrieved November 14, 2022, from https://stackoverflow.com/questions/28532190/why-does-windows-handle-scrollbars-in-kernel

[3] JackMacWindows. (2021, January 6). *Graphics mode*. CraftOS-PC. Retrieved November 14, 2022, from https://www.craftos-pc.cc/docs/gfxmode

[4] *Join the Leveloper Software Discord Server!* Discord. (n.d.). Retrieved

November 14, 2022, from https://discord.gg/HtYwkHWpqN

[5] MCJack123. (2021, January 7). *MCJack123/CCKernel2: A kernel for ComputerCraft written in Lua.* GitHub. Retrieved November 14, 2022, from https://github.com/MCJack123/CCKernel2

[6] MCJack123. (2022, May 9). *YellowBox: Virtual Machines for ComputerCraft.* Gist. Retrieved November 14, 2022, from https://gist.github.com/MCJack123/e634347fe7a3025d19d9f7fcf7e01c24

[7] Wikimedia Foundation. (2022, October 12). *Operating system.* Wikipedia. Retrieved November 14, 2022, from https://en.wikipedia.org/wiki/Operating_system