Plan 9 and Inferno Go to School

Brian L Stuart

ABSTRACT

Although not originally designed for educational purposes, Plan 9 and Inferno are excellent tools for use in Computer Science education. Here I discuss how I use them in my classes at Drexel University. The discussion includes CS314: Computing in the Small using Plan 9, the Operating Systems courses CS370 and CS543 using Inferno, and the computer I take to the classroom for presentations running Plan 9.

1. Introduction

Education in the realm of computing has long been influenced by the work at Bell Labs. The CARDIAC developed by David Hagelbarger and Saul Fingerman was the first exposure for many people to the inner workings of a computer.[3] My CS164: Introduction to Computer Science course still uses the CARDIAC as an architectural example, and students get their first taste of machine language programming on a CARDIAC simulator that runs in their web browser. An entire generation of Computer Science students learned how an operating system really works by studying John Lions' commentary on 6th Edition UNIX.[4]

Drexel University has a long history of innovation and contribution to Computer Science Education. In the early 1960s, when the name was Drexel Institute of Technology, one of the widely distributed FORTRAN manuals for the IBM 1620 was written by a Drexel faculty member.[2] In 1984, the university partnered with Apple Computer becoming one of the first schools to provide Macintoshes for its students. Today, the Computer Science Department as part of the College of Computing and Informatics is one of the strongest units in the university. Each year, we have hundreds of students enrolling as Computer Science and Software Engineering majors. Almost all of our students participate in cooperative education, gaining substantial industry experience prior to graduation, and our graduates are highly respected in the industry.

For all the courses discussed, one of the key principles that guides my teaching is that it is critical to understand how things work, and not just how to use them. One of the best expressions of this was found on the office chalkboard of Richard Feynman after his passing: "That which I cannot create, I do not understand." Without understanding the mechanisms under the covers, seeing example code that one just copies is no better than the children at Hogwarts memorizing spells. If I understand how something works, I can find lots of ways to use and apply it, but if I only know a way to use it, then I'm limited to that procedure.

2. CS314: Computing in the Small

Like most programs in Computer Science and Software Engineering, ours doesn't require our students to get any direct experience with hardware. Yet, there are more CPUs running in embedded systems than any other type of computer system. To provide students with an opportunity to have some experience with embedded systems, we offer CS314: Computing in the Small.

Historically, embedded systems ran on relatively small processors without a conventional operating system. This type of embedded design is still the right way to go for many applications. Today, however,

we frequently find systems being built around ARM or MIPS processors with enough horsepower to run operating systems like Linux. It is this latter approach to embedded applications that this course addresses.

For this class, I chose the Raspberry Pi as a good platform. Certainly the price point is ideal for students (except for the recent price-gouging). The Pi has a good variety of I/O support, giving students a chance to work with a number of different devices. It provides GPIO lines, SPI ports, and I^2C interfaces.

Perhaps the best reason to use the Raspberry Pi for this class is that there exists a solid Plan 9 port for the Pi. There are several reasons to prefer Plan 9 over Linux in this sort of class. One reason is that the Plan 9 kernel is written in a much clearer and more straightforward way. This makes it much easier for the student to understand how a user-land operation makes its way to setting the device registers. Being able to see that connection is critical for the level of understanding that I'm trying to get across.

Another important benefit of using Plan 9 is the fact that it is the path less taken. This often seems contradictory to students who believe that learning in programming is always about examples that they can basically copy and tweak. They expect to use web searches to find all possible lines of code that they just assemble. However, this does not represent a significant level of understanding. By requiring that the students construct their code using essentially only the man pages for reference, I can guide them to understanding what they are doing at a deeper level. Related to this is the fact that unlike what is often presented for Pis in the Linux and Python world, programming is not all about using a library that does everything for you.

The third reason I prefer Plan 9 for this course is to expose students to some of the newer and cleaner ideas found in Plan 9 over Linux and other UNIX-likes. For example, the file-oriented interfaces provided by the drivers in Plan 9 using ASCII text make it much easier to demonstrate things in class. They help the students see that APIs with complex structures and many arguments aren't the only way to do things. The hope is that they will come to understand that just because they see something in industry doesn't mean it's the right way to do it. Far too much of what is done under the label of "best practice" is, in fact, quite bad practice.

The following subsections discuss the assignments I give in this class. As a matter of full disclosure, I did have a hand in developing these drivers as part of the original course development. Much of the driver testing was done on devices that are listed as recommended for the students to use.

2.1. GPIO

Giving an initial assignment that just asks for a simple input and output using GPIO lines provides the students with an easy assignment that makes sure they have the system up and running on their Pi, that they understand how to connect devices, and that they understand how to use the compiler. The actual assignment is generally simply to connect a switch of some sort and an LED of some form to a pair of GPIO lines and to control the LED with the switch. I encourage them to implement a three-state machine with states of on, off, and flashing, but if they want to just implement a flashlight, that's okay for this assignment.

Although it's a rather easy assignment on the surface, there are a few details that the student must think about and understand. On the output side, the student needs to first understand whether they have a raw LED that needs a current limiting resistor or a device that integrates the resistor with the LED. There was a time when we could assume that all Computer Science majors knew Ohm's Law and could understand the role of the resistor, but that's no longer the case. As a result, I do explain that in class.

On the input side of the GPIO assignment, there are also questions that the student must address. One of the devices that students might purchase from Adafruit and use provides a logic output level based on a capacitive touch sensor. If they use a device like that, then the input GPIO line is best configured without either an internal pullup or pulldown. On the other hand, if the student has a simple mechanical switch, they need to understand the role of the pullup or pulldown and how to decide which

to configure. Overall, this relatively simple assignment gives the student an early and quick taste of the world of embedded systems.

2.2. SPI

SPI is one of two commonly used synchronous serial interfaces with hardware support on the Pi. I typically give them an assignment to write code that uses an LCD panel driven with SPI. Particularly with the small color LCD panels, most of the controller chips they use are quite similar. To use them, the student must understand the protocol for accessing a device's internal registers. Once the students understand how to do that, they are then faced with a myriad of pixel formats to choose from. This gives them a chance to understand the tradeoff between color detail and performance. Using a display device also exposes the student to the effect of the SPI clock rate. Particularly when writing a whole screen to clear it, the difference made by different clock rates is quite evident. It's also a good exercise for them to see how a maximum clock rate specified in a datasheet can often be exceeded, but usually not relied on.

Of course, SPI is used for other types of devices besides just LCD screens. I discuss in class one of my more recent uses of the SPI interface. A number of LED arrays are currently available that are constructed with 8×8 LED modules attached to a controller board. The controller board drives the LEDs with what amounts to a large shift register. SPI is a particularly convenient interface for devices like this as its different modes allow one to select the combination of rising and falling edges appropriate to the device. I recently used one of these as part of an ENIAC simulation running on a Plan 9 Raspberry Pi.

2.3. I^2C

Most of the I^2C devices the students use are sensors of various sorts. Among the more popular ones are various light sensors and accelerometers. One part of I^2C that is especially challenging from all perspectives is the subaddressing. Devices often use this feature to identify which device register is being accessed. However, the details of its implementation are quite tricky, and the mechanism exposed to applications is perhaps unintuitive. In particular, the offset in a pread or pwrite call is used (some might say abused) as the subaddress. The justification for this design choice is that if we view the register file of the device as exactly that, a file, then the idea that the offset is another way to specify the address of the register is natural. This functionality gives the student a chance to better understand the role of the offset, and the use of pread and pwrite.

2.4. 1-Wire

The 1-Wire interface is an especially interesting one. It is so named because it uses a single line for both input, output, and power. Of course, there still must be a ground connection, so calling it "one" wire is a little euphemistic. The Pi hardware doesn't have direct hardware support for the 1-Wire interface, but by playing with the GPIO line function, it is possible to implement.

The Pi acts as the bus master and all devices share a single line with a pullup resistor. Whether reading or writing, the bus master controls the clocking of the data. It does so by driving the bus line low for a short period (approximately 2μs). If neither the bus master nor any device drives the bus line low, the pullup pulls the bus high. When the Pi is transmitting, it initiates a bit by driving the line low. If transmitting a 0, the line is kept low for longer (typically 20 to 60 μs). If transmitting a 1, the bus is set to high impedance, letting the pullup pull it to a high logic level. When the Pi is reading data, it still initiates each bit transmission by driving the line low for a short time. The Pi then waits about 10 to 15μs and samples the bus line. If the device is sending a 0, it drives the line low, and if a 1 the bus is left high impedance. By setting the output to a 0, we can switch the GPIO line function between input and output to speak the 1-Wire protocol. When the line is set to output, the 0 output drives the line low, but when set to input, it goes to high impedance. Although we (my students and myself) have been able to run the 1-Wire protocol strictly from userspace, I've added an additional function to

the GPIO driver to support 1-Wire. Writing "function n pulse" to /dev/gpio switches the line to input, delays 2μs, and switches it back to output.

2.5. Device Driver

Because of the importance of understanding how everything works and all the layers of abstraction, I do make a point of teaching the structure of drivers in Plan 9. It gives the students a sense of what's involved with doing embedded programming without an operating system supporting them. It also gives the student a chance to see an exemplary system design.

Given the exposure students receive to drivers, it makes sense to give them a chance to experience modifying and rebuilding the kernel. Most often for this project, I give them a relatively simple modification to an existing driver, rather than expect them to create a new driver. Both the 1-Wire assignment and the device driver assignment depend on available time in any given term.

2.6. Final Project

Because I like to give students a chance to stretch their wings and show creativity, I give them the opportunity to specify their own final project, subject to my approval. It's always quite interesting to see what ideas students come up with. One that particularly comes to mind is the student who connected some sensors to his child's wooden train set and then drove motors to create gates at crossings. Another project that I remember was a student who purchased a cell phone module and connected it to the Pi's serial port. By the end of CS314, he was able to send text messages. Because he was interested in taking it further, he followed the course up with an independent study, adding more cell phone functionality.

3. CS370/CS543: Operating Systems

The first thing to address when talking about an operating systems course is whether it's taught from the perspective of applications with a focus on using system calls, or whether it's taught from an internals perspective. Both the undergraduate and graduate operating systems courses I teach at Drexel are very much internals courses. While many courses are taught with a primary focus on standard algorithms, my preferred approach is strongly influenced by Lions' commentary on 6th Edition UNIX.[4] I believe that students get the most out of a course that provides in-depth exposure and experience with one well-designed system.

There are, of course, several options to choose from when teaching a course centered around the internals of a particular system. Like many, I have taken and TA'ed a graduate course based on XINU, and I've taught using MINIX a number of times. When Vita Nuova open sourced the code to Inferno, the stage was set for the way I really wanted to teach the course. This development resulted in the textbook *Principles of Operating Systems: Design and Applications*.[5] Five of the chapters in the book attempt to present aspects of Inferno internals in a way inspired by Lions.

Inferno has a number of characteristics that I want in a system for teaching operating systems. When teaching with MINIX in the '90s, I could expect students to partition their drives and install MINIX so that they could dual boot, but that quickly became too much to ask. Although running MINIX on a simulator would have been an option, Inferno's ability to run hosted by another OS made it even more convenient. However, Inferno is not an artificial system that only runs in simulation. The fact that it also runs natively on bare hardware means that we have device drivers and their interrupt handlers that can be studied. Another important aspect of Inferno is that it is written in the beautiful and elegant style of Bell Labs, giving students a chance to see very well-written code. Finally, Inferno incorporates ideas such as per-process name spaces that are newer than what is usually seen in operating systems classes.

As with CS314, the following subsections discuss various programming assignments that I give in CS370 and CS543. Unless specified otherwise, the projects are ones that might be given in either the

undergraduate or the graduate version of the course. Even when the core of the assignment is the same, I generally require the graduate version of the class to work with native code, rather than hosted.

3.1. First Project

The first project in both these courses is generally a pretty easy one that is mostly about the students getting the system installed and them getting used to working in a larger code base and building the system. One of the most common things I ask them to do is print out the time that the kernel was built in hours, minutes, and seconds. I've stopped asking them to give month, day, and year, because they all just look up some algorithm online. As a hint, I tell them that they should look for KERNDATE in the output of the build process. The idea is that the student should recognize that KERNDATE is defined with a -D option to the compiler when compiling $CONF.c. If they look for how KERNDATE is used in that file, they will find that it's used as an initializer for a global variable kerndate. Finally, students need to recognize that to manipulate kerndate and print the results from a file like dis.c, they need to add an external declaration. Overall, it's an easy assignment, but it gets them to start working with the code.

3.2. Process Management

The second project is generally related to the process handling code of the system. There are two types of processes in Inferno, kernel processes and user processes that run compiled Limbo code in a Dis VM. In this course, we put most of our attention on the user processes.

Relatively easy forms of this assignment involved adding some form of instrumentation to the scheduler. More involved assignments ask the student to modify the scheduler. Sometimes I have asked for a UNIX-style priority scheduler. In other cases, I've asked the student to add mechanisms for adjusting the length of a quantum. One other process management project that I've sometimes assigned is to add a simplified suspend and resume mechanism roughly similar to the BSD stop and continue mechanism.

3.3. Memory Management

The Inferno memory management is built around variable sized allocation from a free block data structure consisting of a binary search tree where each node is a circular doubly linked list. This is a wonderful data structure to test whether students really understand how to deal with data structures. At the same time, it does open the door to experimenting with different structures here. To that end, the most common assignments I give on memory management involve replacing the tree of linked lists with something else. One common version is to have the students replace it with a single linked list. Another is to implement something like the slab allocator in Linux. The idea here is that some request sizes are very common, and it would be more efficient to just put blocks that get freed into lists of common sizes, rather than repeatedly splitting and coalescing them. The single list is more typical for the undergraduate section and the slab-type allocator is more typical for the graduate section. Although often frustrating to the student, one of the most educational experiences about this assignment is how bugs in the memory allocation can prevent the system from booting at all. Debugging that situation is often a new experience for many students.

3.4. I/O Devices

For the undergraduate section of the class, I generally settle for going over code from kb.c and sdata.c in class and explaining why talking directly to controllers is something that we only see in the native versions. However, because the undergraduate sections generally work with hosted builds of the system, there's not really an opportunity for them to work directly with devices. On top of that, because our terms at Drexel are 10-week quarters, four kernel assignments are probably enough for undergraduates.

For the graduate section, I do often add a fifth kernel project (but require fewer problem set submissions). The additional assignment is usually related to the keyboard driver, as that's something that's relatively

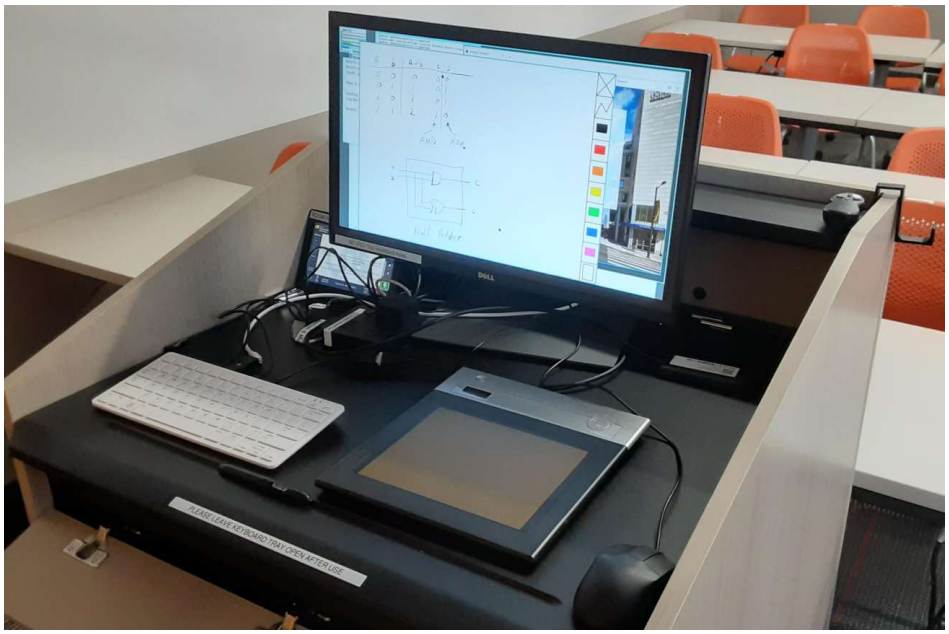

Figure 1: Raspberry Pi400 Presentation Set-Up

easy and can be done with a minimal image booted from a virtual floppy. Most often, I ask them to support multiple key mappings that can be switched with a special key sequence.

3.5. File Systems

Although I do cover the implementation of kfs in class, I don't expect the students to come up to speed with Limbo enough to do a project with it. Instead, I have my students develop a new kernel server, and to make the job easier, I give them a devskel.c to work from. One of the simpler versions of this is a simple string manipulation. A string is written to the served file and when the file is read, some modification of the string is retrieved. Another form is a simplified encryption layer that encrypts and decrypts data as it's passed to a backing store. One of the more complex versions of the assignment is for the student to implement a RAID driver layer. Complete and thorough implementations of the encryption or RAID layers can be tested by running kfs on top of them.

4. Raspberry Pi400 Presentation Machine

The final use of Plan 9 discussed here is the machine that I take into the classroom to do my presentations. It is a Raspberry Pi400. This machine is basically a Pi4 built into a keyboard, making it very compact and convenient for use in the classroom.

All of the classrooms in our building are equipped with podiums that are connected to projectors and wall monitors. Each podium does have a computer running Windows installed, but many faculty bring their own laptops to use in the class. However, the podium gets a little crowded with a full laptop in addition to the monitor attached to the podium. On the other hand, using the existing monitor with the Pi400 is much easier to work with and leaves room for both a mouse and a drawing pad to use for a virtual white board.

The Pi400 I use in the classroom runs Plan 9, with both rio and acme configured to use larger fonts than usual to aid readability on the projector screen. Part of the reason for using Plan 9 is that it is, of course, a much more pleasant platform for developing tools for my classes. It also sets a good example

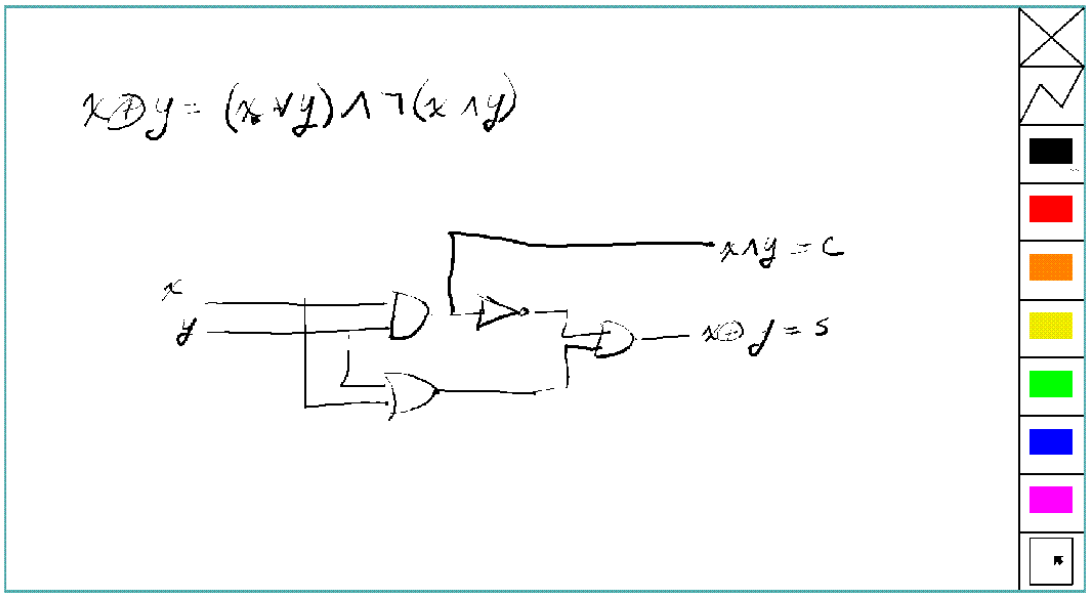

Figure 2: Virtual White Board Example

for students to see that not all of computing need be tied to mainstream and commercial systems. Figure 1 shows the system set up on one of our classroom podiums as I use it in class.

There are a few details of the use of the Pi in that context that deserve some discussion. The first of these is the virtual white board mentioned above, an example of which is shown in Figure 2. The slate device I've been using is one that our IT group had, but no one had ever used. When I asked about recommendations for such a device, they said "take this one; nobody ever used it." It looks like a generic USB HID, and its protocol was fairly easy to discern. My initial approach was written as a program using the P9P library, running on a BSD UNIX sending output to an instance of wish to handle the graphics. When migrating to the Pi, there were two major shifts. First, usbd didn't automatically present a generic HID interface, but handling that was a simple matter of a small program using the USB library to locate and present the slate device. With it, reading from /dev/slate gets a single protocol message from the device.

Another change in the virtual white board highlights one of the best design decisions of Plan 9. In other environments, writing to the graphical display typically requires access to special system calls or special libraries, making it at best a nuisance when working in small experimental languages. In those setting, I have often gotten around the problem by running an instance of wish as a child process and sending Tk commands down a pipe. However, this still requires the existence of another program such as wish written in another language, reducing the stand-alone potential of the experimental language. On the other hand, by providing access to the draw device using ordinary file operations, a language's interpreter or run time environment aren't required to have access to these special interfaces. The relevance here is that the virtual white board is written in just such a small experimental language.

As second issue that must be dealt with when using Plan 9 in such a context arises from the unfortunate push toward the web browser as the universal UI. As with all other bureaucratic organizations, universities are far from immune to this phenomenon. In particular at Drexel, the primary vehicle of distributing material to students (syllabi, assignments, grades, etc.) is a Learning Management System (LMS) called Blackboard. Along similar lines, the mechanism by which both faculty and students learn of the final exam schedule is the Registrar's web site. As a result, there are days when I cannot escape

using a browser in the classroom. All of these web interfaces are heavily JavaScript-laden and often depend on the frameworks du jour. Although recent work on porting netsurf[1] shows promise, we are still a long way from a native Plan 9 web browser that can function with the assumptions made in the web development. To deal with this situation, I euphemistically "declare victory and depart the field of battle" by running a NetBSD machine in my office on which I run a web browser and to which I connect from the classroom using VNC.

The third issue that I've dealt with in my use of Plan 9 in this role is ssh. Currently, the SSHv2 implementation in the 9legacy distribution falls rather short of ideal, a situation for which I accept personal responsibility. It was originally written with the various cryptological algorithms that were required by the RFCs included, but not many beyond that. In the years since, several of these algorithms have been dropped from the default configuration of OpenSSH (and presumably others). When I first started using my Pi in this way, it was unable to connect with our departmental cluster because the key exchange algorithms were out of date. After adding newer versions of Diffie-Helman key exchange algorithms, I was able to use the system successfully during our fall term using ssh directly and as a carrier for sam -r. More recent updates to the cluster, however, have revealed that the original ssh-rsa and ssh-dss public key algorithms have now also been dropped, and I am in the process of adding replacements for them.

5. Conclusion

Without Plan 9 and Inferno, my approach to teaching, especially in systems courses, would be very different, and my students would be the worse for it. The tradition that Bell Labs established long ago of connecting their research to education and reaching out to the classroom has had an important impact on many of us. Although the Bell Labs of that time is no longer with us, its impact continues on through the influence of some of its later work on today's students. Those who continue the use and develop Plan 9 and related software are making a meaningful contribution to the improvement of the overall calibre of people entering the field.

References

[1] Jonas Amoson. Porting the netsurf web browser to Plan 9. In *Proceedings of the 9th International Workshop on Plan 9*, 2023.

[2] Decima M Anderson. *Basic Computer Programming: The IBM 1620 FORTRAN*. Appleton-Century-Crofts divison of Meredith Publish Company, New York, NY, USA, 1964.

[3] David Hagelbarger and Saul Fingerman. *An instructional manual for CARDIAC*. Bell Telephone Laboratories, 1968.

[4] John Lions. *A Commentary on the UNIX Operating System*. Republished 1996 by Peer-to-peer Communications, 1977.

[5] Brian L. Stuart. *Principles of Operating Systems: Design and Applications*. Cengage, 2009.