Teaching an Undergraduate Unix Course

BY BENEDICT REUSCHLING

Backstory—Inheriting a Unix Course

Back in 2002, when I was an undergraduate computer science student, I made my first real contact with a Unix system. I vividly remember my first programming lecture when the professor opened both his ThinkPad laptop and the lecture itself with these words: "There are two types of professors in this department: those who use Unix — that is the group I belong to — and there is the group that uses Windows — and that's everyone else." He not only taught us programming, but he did it all in the command line, displaying programs with cat, editing them in vi, running make to compile, showing slides in X11. All very basic, yet somehow very appealing (maybe because I grew up with DOS, never knowing what I was missing in a terminal). After all, running Linux or any kind of Unix on your machine back then showed that you were hip and cool in computer science terms.

Running Linux or any kind of Unix on your machine back then showed that you were hip and cool in computer science terms.

I attended all the lectures this professor taught — the advanced programming class, then operating systems and distributed systems. By that time, I also had my own laptop loading Linux, and shortly after discovering FreeBSD, I ran that exclusively.

There was another course taught by the same professor — not a part of the main curriculum — called Unix for Developers. In this course, Unix was the primary learning objective. This was clearly old-school, but, nevertheless, interesting to me. We learned about Unix tools to edit files efficiently and wrote scripts in shell and awk to add missing functionality when needed. The course was challenging, but I was eager to learn and try out as much as possible on my own system.

As time went on, the professor eventually switched to a Mac (with many of the students doing the same), got involved in a lot more lectures and project work and no longer had time to teach Unix for Developers. This occurred after I had graduated and had worked at

the department as a lab engineer. He must have remembered my enthusiasm for the subject when he asked whether I would be willing to take over his Unix for Developers course. I agreed and he outlined how it was handled. It had become a bit outdated at the time, covering Linux kernel 2.4 when 2.6 was already out, so I set out not only to update the content, but to also put in my own bits of BSD here and there.

A few years after teaching the course in German, I was asked whether I would be willing to teach it in English for our exchange students. As many of the concepts are in English and don't translate very well into German, I went over the whole script again and translated it. I've been running it in English ever since, even occasionally using extracts as tutorials at conferences. But let's go back to the early days...

The Organizational Structure

So here I was, with a required lecture and a semester to fill with course content. The course is taught in the winter term, which means roughly 15 weeks between October and January with a Christmas break in between. The written exam is scheduled for February, which leaves about two weeks between the last lecture and the exam date for the students to prepare.

So here I was, with a required lecture and a semester to fill with course content.

There is a weekly, 90-minute lecture and a three-hour lab. Because of the popularity of the elective course, there are typically 40 students taking the course. This means that each lab group of roughly 16 students meets every 14 days, alternating with the other group so each have 5 lab dates in total.

The labs are not graded but need to be completed by each student group (typically done in pairs) to be permitted to take the exam at the end. There is also no midterm evaluation — the final grade is determined by the exam result. This scenario has been debated many times and whether the labs should be graded given the effort students put into them. Other classes in this German University follow the same structure (with a few exceptions), so it's pretty much established form.

Since this is an elective course, I only get students that are interested in the topic. No student is required take my course to get their degree. They must have a certain number of elective courses listed in their records, but the classes can be chosen freely. Not only does this reduce the overall number of students, but I get two types of students: those who want to learn Unix and know very little about it, and those who know Unix quite well already and want to learn even more (or want to get an easy grade). Later, we will see that balancing these two groups is not always easy...

The Lecture

As previously mentioned, when I took over the lecture it was outdated as it had not been offered for a while. I thought I could put some BSD content into it to make it more Unix agnostic since I needed to update slides anyway.

During the very first introductory lecture each semester, it was typical for a student to ask: "which Linux distro are we going to use?" That question usually causes conflicts, as there

are a lot of distributions out there. If I'd say Ubuntu, then the Linux Mint folks are disappointed, and if I say Mint, then the Arch Linux users are appalled. To avoid this (and to give everyone an equal chance to learn something new without sacrificing some familiar concepts), I usually answer the question with: "All of them and none. We're using FreeBSD, but if someone wants to use their distro of choice, then I won't stop you. After all, this is an introduction and not an indoctrination course."

I explain to the students that if they are new, beginning with FreeBSD is as good as any Linux distribution and the concepts carry over well. For those who already know a Linux distro, they can opt to dip their toes into a BSD system and discover a lot of the basics are the same. This usually appeases everyone, and I avoid the distro wars entirely. However, if students chose to run on their own favorite distribution, they don't get any help from me. It's their system, their choice, their administrative work, and not mine. The popularity of certain Linux distros changes with each student generation, while FreeBSD has happily remained mostly the same--stable, free of surprises, and easy to get started with. Interestingly, a lot of the stuff I teach is distribution agnostic, so there is little to no difference between systems. We can sometimes compare differences in class when I do demos (more on that later), but in essence, the commands we use all do what we want them to.

Here is what the course offers:

- Unix Overview (basics like logging in, commands like ls, cp, etc.)
- Editors (vi/vim crash course)
- Shell (command history, tab completion, redirections, pipes, here documents, background jobs)
- Shell scripting (big part on variables, control structures, loops, debugging)
- Shell scripting II (cdialog programming, functions, and traps)
- grep, sed, awk (extracting data from files, manipulating it in various ways, awk-programming)
- Filesystems (ZFS gets introduced here)
- Ansible (Setup, running ad-hoc commands, writing playbooks and change multiple jails in parallel)

This is a lot for 15 weeks, even though it may not look like much. One might argue that Filesystems don't belong in a programming-oriented lecture. That was a remnant from when I took the course, and I thought replacing ZFS for the regular filesystem concepts was a good compromise. Other parts like shell scripting were extended after I learned that some (but not all) of the professors also teach it as part of the mandatory course on operating systems. I added concepts like dialog programming (think of the FreeBSD installer to get an idea of how this may look), functions and traps. They fit together nicely since traps need to call functions when they're executing.

Some of the content changes more often over time, depending on my own interests and based on student feedback. Realistically, one could only comfortably present roughly 40 slides in a 90-minute lecture, considering questions and spending 2-3 minutes per slide.

When the pandemic hit, that format was no longer doable, so most colleagues and I switched to the inverted classroom model. In this teaching format, students study the material up-front, and the lectures are used to address questions and discuss the material. The inverted classroom allows the teacher to provide more material up-front and use the lectures to gauge whether there are common problems that should be explained in class for everyone. It also requires more initiative from the students. If there are not many questions, I assume everything is understood (which can backfire for the shy students), and I do a couple

of demos by sharing my terminal on a projector or in a video call. I've found that students like this format as they can try out things right away on their own machines, they get to see me make errors (nobody's perfect), and it gives the lecture a more dynamic nature rather than going through the material slide by slide.

Content gets added and updated based on student feedback. When I see that students struggle with something, I create a couple of extra slides to help them grasp that concept. This also depends on whether the students have had prior experience with the subject or are completely new to Unix. Overall, I've found that there is something new to learn even for seasoned Unix users, so it does not matter too much if there are some students who have had prior exposure. Typically, ZFS and/or Ansible is both new and exciting to the students because of the capabilities they provide. This is especially true for ZFS. I have had students tell me later — when I see them again in our master's program — that they are glad I taught it and that they use it at home for their own NAS.

The Labs

Lab exercises are intended to have students demonstrate that they have understood a certain topic and can apply it to a given problem. They typically work in pairs and present their results to me for evaluation. They need to get all 5 labs completed to take the exam. The exercises follow the material being taught in the lecture, but there can also be parts that are only explained on the lab assignment sheet and not in the lecture. This can be because it is too small to cover in class or is a separate topic that does not fit into the current curriculum.

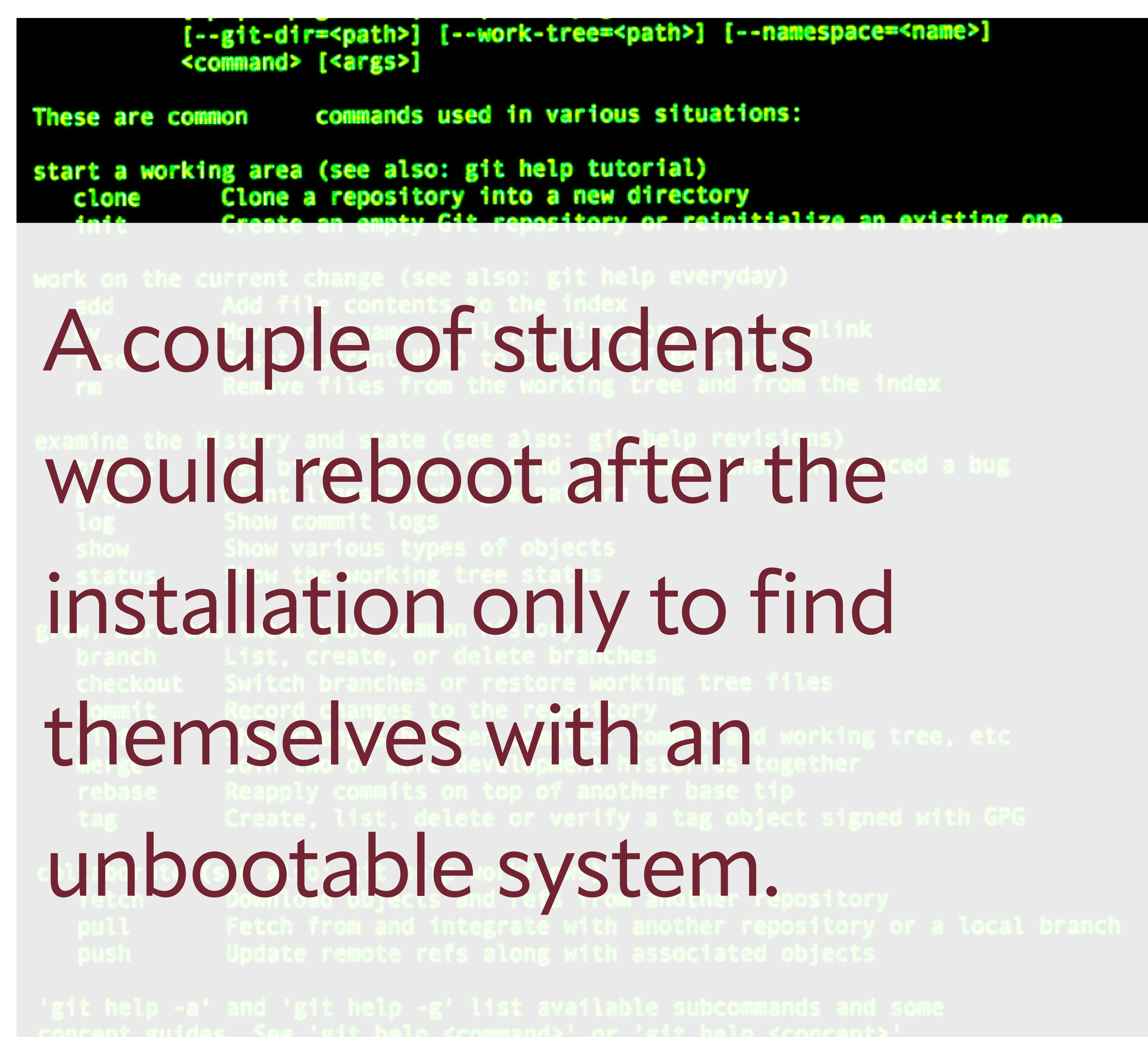
Lab exercises are intended to have students demonstrate that they have understood a certain topic and can apply it to a given problem.

Lab assignments typically involve getting to know something more about the system, doing a programming exercise (or before that, creating useful shell pipelines), text processing, making configuration changes in the system and similar tasks. The most difficult lab for me — the teacher — is always the first one — setting up the Unix system. Remember that we have two types of students. While some struggle with even the most basic installer, others bring a perfectly set up system to the lab and leave after 5 minutes of showing it to me. It's easy if you've done it with at least one distribution (learning about the FreeBSD specifics is typically easy enough), but if it's the first time, it can be difficult for a newcomer. The overall goal of this lab is to have a running system at the end for everyone to use and follow in class. Once that has been accomplished, the subsequent labs are much easier for participants. They can use their installed system and are basically all on the same level as far as the system is concerned.

I've also tried out different formats over the years to see which works best in getting everyone on the same page — so as not to overwhelm the newcomers and not to bore the experienced students. At the beginning, using the projector, I walked through the installer in a VirtualBox VM with the students, explaining concepts and terms as they came up. That worked somewhat, but the advanced students were moving ahead to the next screen and the explanations turned into a lecture of their own.

Then I switched to providing instructions as to what the installed system should have at the end — a certain partition layout, a local user separate from root, and a running network. This tended to create a lot of different results, even though they all used the same virtual hardware platform. Some didn't remember their passwords in the lab 14 days later or had made their partitions too small and couldn't install any software on it. In addition, cheating was much easier, as one student could pass around a finished VM image, and the other students simply imported that on their machines. The students did not learn much from simply running through the installer and hitting enter a couple of times with no clue as to what was going on behind the scenes (partitioning, DHCP calls to the network, etc.).

To prevent cheating and to give students a bit more information about what was going on, I provided instructions on how to do the installation manually. They would drop to the shell and do everything by hand: set up partitions, extract FreeBSD source archives, make basic settings for the network, and install the boot loader. All of this was accompanied by instructions on what the commands they were using did. To prevent cheating, I gave instructions to label their partitions in gpart after the uniquely generated disk ID from VirtualBox. That way, each system had its own ID and I could easily compare them.



A couple of students would reboot after the installation only to find themselves with an unbootable system.

That worked well to a certain extent. A couple of students would reboot after the installation only to find themselves with an unbootable system. They must have written the boot code to the wrong partition — like the one used for swap by not getting the ID in gpart right. I also had a few students stop the installation in between, suspend the VM to do something else, and later try to continue, only to find that the virtual CD provided by the FreeBSD ISO image would no longer be mounted, making all inputs result in "command not found" errors. Yet other students booted their systems just fine, worked with them and then in a later lab would reboot for the first time (suspending the VM all the time) and find themselves in an unbootable system — with all the solutions inside the unbootable partition. Not good, especially since those students would ask me what to do and have me figure out the particulars of their install from months ago.

Although I refined my manual installation instructions to include regular VM snapshots at certain points to go back to, other problems remained. Students did not read my explanations but would simply look for the next command to enter in a 12-page document (including images). That, of course, defeats the purpose of trying to teach them a little about how a Unix system like FreeBSD is installed and what components are involved. Again, the newcomers struggled with this more than the seasoned Unix users. Luckily, the number of struggling students was limited to only a few, while the rest did fine with this lab.

I'm currently doing a separate project with a small student group with the goal of providing course participants with a ready-made machine (jail) running some application. They have to keep this machine with the application running while I inject certain errors that the students have to find and fix. A global hiscore list displays how quickly each team solved it based on points given by a check program that runs over these systems to figure out if and when an injected error is no longer present. Of course, I could inject different errors for

each group or even multiple ones. From shutting down services to removing execute bits or whole files — the possibilities are endless (at least in my mind). Students learn how to keep things running, they don't have to deal with installing it properly in the first place (which is what they typically find at a company), and learn skills to find and fix common errors. We're still fleshing out the details, but I think it will be engaging for students.

The Exam

What can I write about the exam without giving away too much of the content? Since I switched to an English-only lecture a couple of years ago, students fear that they will not understand the questions. But that turned out not to be a problem. The questions are typically programming related like "find the error in this short shell script," which bridges the human language gap quite nicely. There is multiple choice, fill in the blanks, write a short script on your own, or tell me what CoW in ZFS terms means. All are familiar question types for students at this point in their studies.

From the results, I can see that newcomers have an equal chance of getting a good grade in this course as those with prior exposure to a Unix system. I can't tell if the latter group studies at all for the exam, but I can certainly say that not studying at all does not guarantee a good grade. Since the exam typically contains material from the labs in different form, I can also tell afterwards which of the two in the lab group really did the exercises and who did not. That is a late revelation for the students and for me, but sometimes my intuition about which student is the better one is wrong.

Aftermath

Once grading is done and the students have had a chance to review their exams (which they rarely do), the class officially ends. But that does not mean the work is done for the teacher. Since this is a yearly course, I have time over the summer to relax and reflect on it. From the feedback and experiences in the lecture and labs, I refine or even completely rewrite certain parts — typically the ones that evoked a lot of questions during the labs or were small points raised by the group in the exam.

I also find cool new things in the Unix space that I want to teach in the future. During my sysadmin work, I occasionally come across a piece of code or a little problem that later becomes an exam question. Collecting these over the summer break refreshes the course content not only for me, but for the next generation of students. So, it is rare that two consecutive courses will be taught completely the same. That would be boring to me and the students and lab, and exam solutions from previous years would propagate over time.

Can I teach everything that Unix has to offer or that I think students should know? Certainly not. I can scratch the surface and hope that students find it sufficiently interesting to continue learning about it on their own after the course. Some of the more advanced topics are covered by colleagues who go deeper into subjects like managing cloud application development, systems programming in Rust, and similar topics offered as elective courses. Some students complain that I don't cover docker, but then I remind them that we're looking at jails which also have cool features.

Of course, you also have to address recent developments and trends. Whereas a couple of years ago, we'd still have to do basic HTML introductions in another course, we can now assume that many students already possess that knowledge from their school days or private dabbling. The same is true for hardware. A lot of students have never built their own

computer and have only used complete systems. Talking about interactions between components like CPU, RAM, and storage may seem new to those students, even though that is covered in the mandatory operating systems class. If students only bring a tablet or are only used to a graphical UI, it's difficult to introduce them to a text-based shell with a blinking cursor. This is not a FreeBSD-only problem, as each Unix eventually revolves around using the shell, even though it runs in a bells-and-whistles GUI.

I think students are happy to get an introduction to Unix that goes beyond what they learn in their operating systems classes. While those classes usually revolve around how a scheduler works, what the MMU does, and how system calls are good to know for programmers, my course is a more hands on, day-to-day use of Unix as an operating system for end users. It's certainly not perfect and has to constantly adapt to the changing times, but I like the current concept and students do as well.

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. Benedict is one of the hosts of the weekly bsdnow.tv podcast.

The FreeBSD Project is looking for

- Programmers
- Testers
- Researchers
- Tech writers
- Anyone who wants to get involved

Find out more by

Checking out our website
freebsd.org/projects/newbies.html

Downloading the Software
freebsd.org/where.html

We're a welcoming community looking for people like you to help continue developing this robust operating system. Join us!

Already involved?

Don't forget to check out the latest grant opportunities at **freebsdfoundation.org**

Help Create the Future. Join the FreeBSD Project!

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system.

Not only is FreeBSD easy to install, but it runs a huge number of applications, offers powerful solutions, and cutting edge features. The best part? It's FREE of charge and comes with full source code.

Did you know that working with a mature, open source project is an excellent way to gain new skills, network with other professionals, and differentiate yourself in a competitive job market? Don't miss this opportunity to work with a diverse and committed community bringing about a better world powered by FreeBSD.

The FreeBSD Community is proudly supported by