Building and Running an Open-Source Community—The FreeBSD Project

BY MARSHALL KIRK MCKUSICK

The Berkeley Software Distribution (BSD) started out as a part-time effort of Bill Joy, then a graduate student at UC Berkeley. He ran it out of the office he shared with four other students. Today it has thousands of developers and is used by untold millions of people around the world. This is the story of how we got there.

FreeBSD is an open-source operating system derived from 4.4BSD-Lite released in 1992 and 1994 by the University of California at Berkeley. FreeBSD is one of a very few open-source projects still active after more than 25 years.

The typical life cycle of an open-source project is that a person, or sometimes a small group of people, get excited about doing something and hack up a bunch of code for it. What typically ends up happening is that they finish with the idea, or the leader of the group gets bored and moves on, and the project dies. If you look at SourceForge or GitHub, 90% of those projects are "dark," meaning they have not had any commits to them for at least a year. An open-source project with longevity has to have some method for passing leadership from one person to another. Much of this article looks at FreeBSD's governance structure and how it has evolved over time.

Current Users of FreeBSD

Today, FreeBSD is most heavily used to provide core Internet support. FreeBSD is used to operate root name servers, major web hosts, routing infrastructure, and as the foundation for major commercial operating systems. Some examples include Netflix content distribution servers, which are the source of one third of all Internet traffic; New York Internet, which does much of the network infrastructure on the East Coast; and the BBC, whose servers are primarily running on FreeBSD. Juniper Networks routers and Network Appliance file servers are built on top of FreeBSD. FreeBSD is heavily used in the appliance and embedded operating system market where companies need to put their intellectual property inside the operating system so they cannot use Linux due to its GNU Public License (GPL). I discuss the GPL later in this article.

Verisign uses FreeBSD to handle their critical Internet name service lookup. They use FreeBSD, Linux, and commercial operating systems because they want a diversity of operating systems in case there is a zero-day exploit in one of the operating systems. When a zero-day exploit happens, they can take down the affected systems and not lose their service, as their service must be up and running at all times. They value FreeBSD because it is a small enough niche that there is not a lot of targeting of it. It is hardened enough that it is harder to crack it than to crack other operating systems and part of FreeBSD's security derives from its code base chang-

ing slowly. Less known is that FreeBSD is the operating system used on the Sony PlayStation 4 and later. Apple's Darwin, the base operating system in Mac OS X, is directly derived from FreeBSD.

CSRG at Berkeley

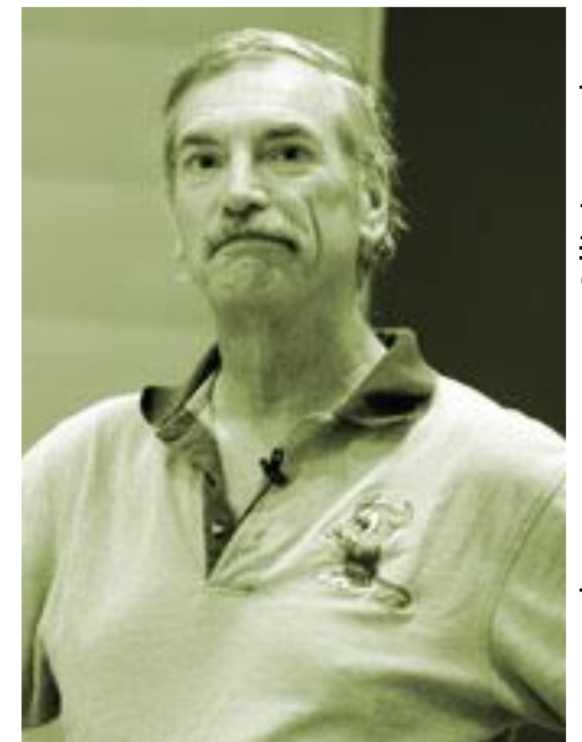
Photo courtesy Ollivier Robert

I will start with a bit of history, rolling back the clock to the 1970s and 1980s when BSD was being developed at the University of California at Berkeley. BSD started at Berkeley in 1977 as a one-man project run by Bill Joy (who went on to be one of the founders of Sun Microsystems).

The first distribution was called the BSD distribution. It was released in early 1977 and consisted of just three utilities that Bill wrote that would run on UNIX:

- the C shell,
- the ex/vi editor,
- the Pascal compiler/interpreter

You would just add these utilities to your existing version of UNIX. Typically, you were running Version 6 UNIX, also known as Sixth Edition UNIX or V6. As users converted to Version 7, which was released in 1979, they could take the BSD utilities with them. There were additional utilities that were added to the 2BSD release in May 1979.

Up to this point, UC Berkeley was running UNIX on Digital Equipment Corporation PDP-11 machines. The PDP-11 had a 16-bit address space and thus had a maximum of 64K bytes of text plus 64K bytes of data. UNIX was a swap-based system. UNIX read programs into memory to run them and freed the memory when they were done.

In 1978, Digital Equipment Corporation released their 32-bit VAX computer (VAX stood for Virtual Address eXtension). Berkeley was an early adopter, receiving a VAX with serial number 7 with its maximum memory capacity of four megabytes. The VAX was the first hardware on which UNIX ran that supported virtual memory. The vendor-supplied operating system for the VAX was VMS, which was mostly a batch system. Its initial front end was a primitive command-line interface for interactive use that was not liked by people who were used to UNIX.

The initial port of UNIX to the VAX (32/V) was done at Bell Labs by John Reiser and Tom London and did not use the virtual memory hardware. It ran as a swap-based system just like the PDP-11. So, you could not run any program larger than physical memory.

Users at UCB were working with Franz Lisp, which needed far more address space than the four megabytes of physical memory on Berkeley's VAX. VMS did support virtual memory, but its interface was not to their liking. So, Bill Joy ported Ozalp Babaoglu's VM system into UNIX /32V over the four-week Christmas break. The port was stable enough by a couple of weeks after the Christmas break, so it permanently replaced VMS usage at Berkeley.

The release containing the VM system became 3BSD and was released in 1979. The 3BSD distribution was the first Berkeley distribution containing the whole kernel, the utilities and libraries, etc. Distributing complete systems became the model of the BSD distributions still used to this day.

The Defense Advanced Research Projects Agency (DARPA) funded a lot of early computer-infrastructure research. DARPA was interested in having a common machine and operating system that would be used across all the DARPA projects so they could easily move code developed by their supported groups between their projects.

DARPA wanted to have other features added, most importantly an implementation of TCP/IP to replace their earlier NCP networking infrastructure as NCP supported only 254 hosts.

Berkeley got one of the DARPA contracts to ''harden'' the BSD kernel and help integrate the networking code. This contract led to the formation of the Berkeley Computer Systems Research Group (CSRG).

After its humble start in Bill Joy's office, the CSRG eventually grew to a staggering four people. They would spend a lot of time talking with other people finding out what they were doing with the system, getting their improvements, their new utilities, and bringing them back and getting them incorporated into the system.

In the 1970s, the network was composed mainly of dialup links running at 1200 baud. Most of the contributions to the CSRG came on a nine-track tape, though occasionally it was sent as an email attachment across a painfully slow link.

In the early 1980s, CRSG started using a source code control system from Bell Labs called SCCS. For the first time, SCCS allowed them to keep track of all the changes that were being made to the system, who made them, and the reasons that they were made. Using a source-code control system was a novel concept back in the early 1980s.

As a DARPA contractor, the CSRG was given access to the ARPAnet (which is what eventually became the Internet). The ARPAnet made it possible to expand the set of people who could work on the system. In particular, the CSRG had network connectivity that allowed others to actually log into their machines remotely, thus avoiding expensive long-distance charges when using modems to connect cross-country or further. Most importantly they had a source code control system so they could avoid collisions and keep track of who was doing what.

Initially, the CSRG gave accounts on CSRG machines to 10 other trusted people who could log on and update sources. This approach was far more efficient than the previous receipt of tapes and email that had to be integrated by CSRG staff. The selected people were those who had been making a lot of contributions and understood the way that CSRG liked to have things done. They would coordinate some set of people with whom they were working, thus increasing the total number of active contributors to about 100 people.

CSRG staff used SCCS to track changes and verify them before doing distributions. A key role of the CSRG was to ensure that the system as a whole remained consistent. Before each distribution, the CSRG team would have the source code control system list everything that had been changed since the last distribution. In those days, the list of changes could be printed out because it was not too huge. The CSRG team would go through and look at every single one of the changes that had been made, make sure that it fit in, and that it didn't break something that was going to affect some other program. When a change was found, say, in the "diff" program that broke the "patch" program, they would contact the folks that made the changes so they could sort out the program that needed to be fixed up.

This structure formed the basis for the current BSD-based projects having:

- a core group overseeing the development,
- a larger set of people who are doing commits into the system, and
- an even larger set of people that are feeding the people doing the commits.

FreeBSD Project Structure

The FreeBSD Project currently uses Subversion for its source-code-control repository. The Project is in the process of moving to using Git as Subversion as Git is more widely used and understood. Anyone can download the repository, but only a subset of the people involved with the project are allowed to update it.

A FreeBSD distribution consists of the core system of about 100 libraries and 750 utilities plus the symmetric-multithreaded kernel that runs on Intel/AMD 32/64-bit, ARM 32/64 bit, MIPS, PowerPC, RISC-V, and other processors.

All other software is maintained in the ports collection that currently has over 32,000 packages. The ports are organized based on different functionality: software development, web

hosting, routing, etc. The ports collection has a database that allows users to look up what they want. The ports come in two flavors: users can either download the source code and build it themselves, or there is a package where it is maintained in a compiled state for the various architectures, so they can just download the binaries.

Releases

FreeBSD has a formal structure for the way that it releases software. The top of the development tree is known as Current and that is where ongoing development is done.

Periodically, FreeBSD branches off from Current and builds a new base distribution referred to as a Stable branch. Bug fixes are first made to Current, then tested. After they have proven to be reasonable and stable, those changes get migrated down into the Stable branches through something called an MFC (move from Current). The checkin to the Stable branch from Current references the particular change that was made to the Current tree. The application binary interface (ABI) is never changed during the lifetime of a Stable branch.

Many projects just have the current release because they do not want to support the older releases as it takes a fair amount of effort to keep maintaining them. Because FreeBSD is used by a lot of companies that build embedded systems, it is important that FreeBSD continue some maintenance of earlier releases for a significant period of time. Historically FreeBSD has attempted to continue providing at least security-level maintenance of the older releases for up to five years.

The FreeBSD Community

Most contributors to FreeBSD are volunteers. A relatively recent phenomenon is that some companies hire people to liaise with the FreeBSD Project and make sure that problems fixed by the company get upstreamed.

The key to understanding an open-source project is to understand how to make the project appealing to volunteers. Volunteers only do what they want to do. They are not working for you as they would have been in a traditional software development company where you have a manager, you are paying your employees, and the manager tells the employees what they are going to do. It may be some task that is not particularly desirable, but somebody will be assigned to do that task, and they will get it done because that is their job.

In an open-source project, the workforce is people volunteering their time. They do not tend to volunteer to do stuff they are not interested in doing. They only do the things they want to do. Their open-source project contributions are their lowest priority (after work, family, recreation, etc.). So, if some crisis occurs at work, or they have family issues, or they want to go on vacation, all those things take precedence over their development. The best of intentions can be preempted by other priorities. The effect of being the lowest priority is that you do not really have a schedule.

Volunteers are transient. Most people who work on an open-source project get excited, they work on it for a while, then they lose interest, and they go off and do other things. Most people do not make a lifetime career working on one particular project. So, projects need to plan for and deal with transience. If they fail to do so, they end up having a lot of deadwood, which eventually weighs the entire project down. Failing to eliminate dead wood is one of the common reasons that projects fail or go dark. The people who are deadwood need to leave.

Open-source projects need to be self-organizing since they typically have no paid staff or managers. To have long-term success, projects need to be democratic and all members should be able to advance up the project hierarchy based on their contributions to the project. Linux is the obvious exception to this rule where Linus Torvalds started and still runs the project. But

even Linux is going to have to find a path to new leadership.

The usual cause of failure for an open-source projects is that the grand leader and their set of friends control the project. A newcomer who wants to get involved can rise up only so far. They can never move into the leadership role, or, often, even become a direct report to the leader. A successful project has to be able to change the leadership. Otherwise the leadership becomes deadwood, which leaves the project rudderless.

A project must anticipate turnover and manage it gracefully. The problem gets back to the whole issue of transients. The project needs to have a system for getting rid of the deadwood. It has to be a very clear and transparent system, where it is obvious that it is not the people in charge who are picking on people. There must be criteria for remaining involved with the project. Everybody has to meet those criteria and when you no longer meet those criteria, it is time to leave. In the FreeBSD Project, the criterion for remaining a project committer is that you must make a commit once every 12 months.

Organization

The organization of the FreeBSD Project is made up of four concentric rings.

Users

The outermost ring is the users. The FreeBSD users can send and receive feedback for bugs and interact on FreeBSD run mailing lists. Most users are not involved in the development, but some will send and receive feedback for bugs.

Figuring out the number of users of an open-source project is difficult. Often the number is based on the count of downloads of the project's software. Projects like to count downloads, then have some multiplicative factor to decide how many users that represents. Since the software is free, many people will download it but never end up using it. At the other extreme, one download might be for an entire company like Netflix that then deploys it across thousands of machines. Figuring out the balance between these extremes is all but impossible. So, the FreeBSD Project just publishes the number of downloads and lets others choose the factor to use.

Developers

Inside the user ring are the developers, who number about 6,000. It is hard to get an exact handle on the number of developers. This estimate is based on how many unique names have posted to bug lists, how many unique names have posted to various mailing lists, and how many people are credited with code coming into the source tree.

The developers are more directly involved in the Project. They are writing code, sending in bug reports, interacting with the Project to get stuff done. Developers have read access to the source-code repository. If they have changes that they want to put back into the system, then they need to find someone who has the right to commit to the source code tree. They then submit their changes to those committers (committers are described in the next subsection). Making a direct connection with a committer is the best and usually quickest way to get a change made though the change can be submitted by sending a pull request.

Committers

Committers are the people permitted to make changes to the system. In recent years, the number of committers has varied between 350 and 400. Most committers are authorized to commit changes to specific parts of the system. Most commonly, the committer will be associated with a particular set of ports in the ports tree for which they agree to assume responsibility. They will have commit privileges for that set of ports. If they want to make changes in other parts of the system, then they need to get their commit privileges expanded.

Ports typically do not have as much oversight as parts of the core system and the kernel. Typically the person in charge of dealing with the port does not need to check with any other committers. The community from which that port comes, for example Apache, already has a lot of oversight going on. So the job of the committer is to keep FreeBSD up-to-date with the current release of Apache. The committer is welcome to go get other opinions, but they are not required to do so, and they generally do not need to do so.

All changes to the core system and the kernel require review by at least one other committer. The commit message must list who did the review. All commit changes are mailed to all committers, so anyone listed as having done a review can speak up if they feel that their feedback had not been resolved.

Committers are the gateway to feed in the changes from the developers. Developers find committers who are working in their area of interest. They typically find the relevant committer by looking at who has been actively making changes in the code that they are interested in changing and contacting them directly.

They work together to get their changes put in. When one of these developers begins to become more involved, the committer can nominate that developer to become a committer. The process of becoming a committer is somewhat involved. Once a developer gets nominated by a committer, the Core Team (described in the next section) will make a decision.

A new committer is required to have a mentor who is typically the committer that nominated them. The mentor is responsible for ensuring that the committer is not shooting themselves in the foot. The mentor helps the developer learn what they need to know: how the source-code control workflow is managed; how to use Phabricator (the forum that provides pre-commit code review workflows); how to use Bugzilla (the forum that tracks bug reports); where to find out the Project's coding styles; on which mailing lists to post review requests; which changes require a broader review; even small details such as on which IRC channel they should engage.

Mentorship generally ranges from a few months to more than a year. Somebody who is very active in the Project can get through the mentoring process much more quickly than someone who is working more intermittently. Most of the committers appreciate having a mentor as they do not want to do something that is going to be viewed as wildly stupid or out of track with the way the Project works. Often even after the mentor releases them to be on their own, they still continue to request the mentor to keep reviewing their changes.

The Project needs to deal with the unfortunate but important task of identifying and retiring commit privileges from the committers who are no longer contributing. The Project needs to have a well-defined and uniformly applied policy in place to retire inactive committers.

The FreeBSD retirement policy is the automatic suspension of commit privileges after one year of non-use. For the six months after suspension, commit privileges can be restored by simply requesting that they be reactivated. After 18 months of non-use and six months of being suspended, a retired committer has to go through the whole nomination and mentoring process again.

The reason for repeating the nomination and mentoring process is that the way the Project works changes over time. New rules and procedures get put into place. Major tools may have changed, such as a switch in the source code control system. New tools such as the Phabricator review system may have been added. There may have been changes in the bug-tracking system. Often there has been a reorganization of the mailing lists, IRC channels, etc. The returning committer needs somebody to get them back up to speed. The mentoring period is often short for people returning to the Project as committers.

Avoiding the loss of commit privileges is really a minor threshold to get over. Retaining commit privileges requires only one commit per year and can be done by updating your personal

information (what people can see when they pull up "who is this person"). So, the hurdle is about a millimeter high.

A recent study of the Project demographics found that the average age of the committers is 39; the median age is 37; the youngest is 25; the oldest is 68; and the biggest group is concentrated in the 31 to 40 age range.

Most of the people coming onto the Project have been doing software development for at least a decade. They typically start in their teens so by their mid-to-late 20s they have 10 years under their belts. Many of the people come from the Linux world where they have risen up as far as they can go. They are feeling as if they would like to be able to move up further. They find out about FreeBSD, they see how it is set up and they like it, so they come over and join the Project.

While the Project has had committers in their teens, the benefit of a slightly older demographic is that they are past their "youth angst," which means that they have gotten over squabbling on mailing lists. They tend to be a little more mature in their discussions with far fewer ad hominem attacks.

It is important to avoid fights on the mailing list, particularly ad hominem attacks. So the mailing lists are closely monitored. If people start getting into too much of a rat hole or just personal attacks, they will be pulled aside and told to take that off the list. It does not take much to poison a list and it is very hard to get it back on track once that has happened.

Core

The Project is headed up by the nine-member Core Team. Their role is to oversee and manage the FreeBSD Project.

The history of who is in charge of the Project evolved over time. When the BSD Project started at Berkeley it used "the lord and master" approach. A single person made the final decisions. The BSD Project was started and run by Bill Joy until he left to go to Sun Microsystems. Mike Karels took over and then later Kirk McKusick (your humble narrator and author of this article).

When the BSD Project was spun off by Berkeley and it passed to the outside world, several distributions sprang up, including those by Bill Jolitz and later NetBSD and FreeBSD. This article focuses on the evolution of FreeBSD.

When the FreeBSD organization was set up, the organizers decided to set up a group of seven people called the Core group that were in charge of overseeing the Project. The original Core group was self-selected. The people who set up the Project deputized themselves onto the Core Team. They were "Tsars for life."

As is common in open-source projects, some began to lose interest. There was no mechanism to replace them and most did not want to leave because of the prestige of being on the Core Team.

A group of people got together and decided to write a set of bylaws that included a provision to make Core an elected position. Core was also expanded to nine people. The entire Core is elected every two years. Core members are nominated from and elected by the committers. Any active committer can run for Core. Candidates are self-selecting and no nomination is required.

Each candidate puts up a statement as to why they should be elected to Core. In recent years, there have been a series of "office hours" where Core nominees show up to answer questions and expand on their vision for the Project.

After a prescribed number of weeks, the election is held. Every committer gets nine votes. The top nine vote-getters become the next Core. Though there have been efforts to install term limits, at present there are none. The electorate are very cognizant of who on Core is get-

ting things done and who works and plays well with others. The deadwood can continue to run, but they tend not to get reelected. Historically three to four Core members have turned over every two years.

A primary responsibility of the Core Team is to oversee the groups that run the Project. These include:

- the group that administers the Project servers,
- the group that does release engineering,
- the group that manages the QA and continuous integration testing,
- the people that organize and run FreeBSD summits,
- the group that handles security issues,
- the group that oversees the ports repository,
- and the group that oversees the system documentation.

An important task is to organize running the FreeBSD Summits where people actually using the system talk about what they are doing, what they need, and what they would like to see. A key takeaway from a summit is a list of tasks for which someone has volunteered to do something. Indeed, the summits often provide a wealth of ideas that become part of the roadmap that lays out the big vision of what needs to be done. Most important is getting the roadmap implemented, which involves encouraging people toward getting some of their ideas implemented.

Core is also responsible for adding and removing committers. As already described, requests for bringing in a new committer are submitted to Core. Core reviews the request, ensures that one or more suitable mentors have been found, and if all is in order approves the request. Committer removal typically occurs through the timeout mechanism described above.

Another important role for the Core Team is to resolve differences between committers. Most of the time, differences get hashed out on a mailing list or an IRC channel. Sometimes there is a difference in philosophy, and the parties are unable to sort it out. Occasionally, these devolve into "commit wars." Something is checked in, it gets ripped out, it is checked back in again, etc. Here, Core needs to step in and arbitrate.

The tool that Core has to enforce decisions is the ability to temporarily suspend commit privileges. For example, in resolving a commit war, Core can impose a cooling-down period. Neither party will be allowed to commit anything for two weeks. Very rarely, a committer's commit privileges can be permanently removed. Complete removal has only happened twice in the history of the Project. A very small number considering the many thousands of committers over the history of the Project.

Types of Contributors

There are many different types of contributions and it is important to have a broad view. There is not a hierarchy where kernel committers are better than utilities committers, are better than documentation committers, etc. Some other projects have hierarchies such as these but in the FreeBSD Project there is no higher value attributed to one type of contribution over another. People contribute in the areas in which they are skilled.

Some of the areas in which people contribute:

- Port maintainers (196 committers made 27,840 commits to Current in last 12 months). These commits do not count MFCs and other changes that would increase the total by perhaps 50%.
- Utility maintainers (193 committers made 4,334 commits to Current in last 12 months). They are maintaining 775 utilities and libraries. These commits do not count the one to two

additional MFCs for each of these commits.
- Kernel maintainers (71 committers made 6,056 commits to Current in last 12 months). These commits do not count the one to two additional MFCs for each of these commits.
- Documentation group (75 committers made 2,076 commits to Current in last 12 months). One commit is likely an entire section of a manual, or an entire manual (man) page, which are typically much larger than the coding commits. One of the strengths of the FreeBSD Project is that it has better documentation than most other open-source projects. Today the documentation is set up with multi-language support. Most documentation is available in about 10 languages.
- Security team (7 members). Most of the time, they do not have a lot to do, but when a security threat comes up, they have to jump on it now! For example, Heartbleed. Having seven team members allows coverage 24 hours per day. There are people in the U.S., Australia, Japan, India, Eastern Europe, and Western Europe.
- System administrators (7 members composed of 5 website and 2 email). A thankless but very necessary task. The Project has websites, a lot of online documentation, numerous mirrors that need to be maintained, mailing lists, build servers, development machines, etc.
- Release engineers (1 main, 10 helpers) During the actual release phase, the number of helpers jumps dramatically.
- Quality assurance (2 main, 5 helpers). This is the newest group at FreeBSD. Many open-source projects do not have such a group.

Advocacy and marketing group (2 main, 6 helpers). Around a decade ago, a member of this team was elected to Core. It led to a logo change and to a total website overhaul. These changes demonstrate that Core is about more than coding; it is also about running and promoting the Project.

Licensing

Most commercial software is developed using a traditional copyright. Usually the source is not available or is only available in very restricted ways often requiring the signing of a nondisclosure agreement. Recipients may not even be allowed to modify the source and certainly cannot pass it out to anybody else. Since open-source projects generally want their source code to be given away, they do not use traditional copyrights.

Open-source software commonly uses the GNU Public License (GPL), sometimes referred to as a ''Copyleft'' license. Software covered by the GPL version 2 (GPL 2) license must make source code available including any changes that have been made to it. With the more restrictive GPL version 3 (GPL 3) license, in addition to making your source code available, if any of that source code is covered by your patents, you have to make those patents available at no cost for use by anyone who uses your code. The GPL 3 license has caused a lot of consternation. Many companies got around the GPL 2 license by patenting the ideas in their code. They release their source code as required by the GPL 2 license but then require anyone using it to pay patent royalties. This approach was against the spirit of GPL which is why the GPL 3 license was created.

The Linux kernel is under a GPL 2 license, which means that any code written to run as part of the Linux kernel must be released. Some companies use their patents to control its use by others. Many companies avoid having to release their source code by creating loadable binary modules that are loaded into a Linux kernel. Loadable modules are a controversial way of avoiding the GPL terms.

Linux has not changed over to a GPL 3 license. But a lot of the other GNU software is under the GPL 3 license, including most of the rest of the software that is packaged around the Linux

kernel including the GCC compiler and its libraries. It is not yet clear if, when loading the GPL 3 libraries, your program becomes subject to the GPL. The GPL 3 license has made a lot of companies nervous, which has helped projects with a Berkeley license.

The other common approach is the Berkeley license. Sometimes referred to as a ''Copycenter'' license, as in take it down to the copy center and make as many copies as you want. Source and patent rights may or may not be provided, i.e., you can give back your changes, or not, as you choose.

FreeBSD uses a Berkeley license, which has played a big role in its success, particularly with companies that have their proprietary code in the kernel. In practice, the FreeBSD Project gets back about as much code as do GPL open-source projects. But it takes longer as there is a learning curve.

Company X builds their proprietary product on FreeBSD and they choose not to give anything back. Later, the time comes to upgrade the underlying FreeBSD to the next version. They need to port many bug fixes they have made, and they have to port all of their changes. It is a long and expensive process. They realize that if they had given back their bug fixes they would have just been there. So, this experience leads to an incremental amount of bug fixes coming back. Two years hence, they upgrade to the next release. They realize that many of their changes are not really proprietary. If they contribute them back, then someone else will maintain them. By the third upgrade cycle, the company is trying to give back code that is specific to their product. The result is that companies start hiring committers to give them a direct channel to what is changing in the system and to facilitate getting their bug fixes and enhancements merged back into FreeBSD.

Conclusions

Building an open-source community requires an ongoing effort. It is tempting to cling to doing things the way that they have always been done. To maintain its vitality, the Project must adapt to the needs of its users as they evolve over time. The community must be welcoming to newcomers and provide a path for their advancement. Newcomers need mentors to help them get integrated into the community. At the same time, the mentors need to accept the new approaches that are being brought to the Project. The FreeBSD Project has managed to successfully balance these needs for more than a quarter of a century. I am hopeful that they will continue to do so for many more years to come.

DR. MARSHALL KIRK MCKUSICK writes books and articles, teaches classes on UNIX- and BSD-related subjects, and provides expert-witness testimony on software patent, trade secret, and copyright issues. He has been a developer and committer to the FreeBSD Project since its founding in 1993. While at the University of California at Berkeley, he implemented the 4.2BSD fast filesystem and was the Research Computer Scientist at Berkeley overseeing the development and release of 4.3BSD and 4.4BSD.