

# The GNU Project

---

GNU philosophy

The original version of this essay was published in *Open Sources: Voices from the Open Source Revolution*, by Chris DiBona and others (Sebastopol: O'Reilly Media, 1999), under the title “The GNU Operating System and the Free Software Movement.”

This document is part of GNU philosophy, the GNU Project's exhaustive collection of articles and essays about free software and related matters.

Copyright © 1998, 2001, 2002, 2005, 2006, 2007, 2008, 2010 Richard Stallman

Verbatim copying and distribution of this entire document are permitted worldwide, without royalty, in any medium, provided this notice is preserved.

---

# The GNU Project

## The First Software-Sharing Community

When I started working at the MIT Artificial Intelligence Lab in 1971, I became part of a software-sharing community that had existed for many years. Sharing of software was not limited to our particular community; it is as old as computers, just as sharing of recipes is as old as cooking. But we did it more than most.

The AI Lab used a timesharing operating system called ITS (the Incompatible Timesharing System) that the lab's staff hackers<sup>1</sup> had designed and written in assembler language for the Digital PDP-10, one of the large computers of the era. As a member of this community, an AI Lab staff system hacker, my job was to improve this system.

We did not call our software “free software,” because that term did not yet exist; but that is what it was. Whenever people from another university or a company wanted to port and use a program, we gladly let them. If you saw someone using an unfamiliar and interesting program, you could always ask to see the source code, so that you could read it, change it, or cannibalize parts of it to make a new program.

## The Collapse of the Community

The situation changed drastically in the early 1980s when Digital discontinued the PDP-10 series. Its architecture, elegant and powerful in the 60s, could not extend naturally to the larger address spaces that were becoming feasible in the 80s. This meant that nearly all of the programs composing ITS were obsolete.

The AI Lab hacker community had already collapsed, not long before. In 1981, the spin-off company Symbolics had hired away nearly all of the hackers from the AI Lab, and the depopulated community was unable to maintain itself. (The book *Hackers*, by Steve Levy, describes these events, as well as giving a clear picture of this community in its prime.) When the AI Lab bought a new PDP-10 in 1982, its administrators decided to use Digital's nonfree timesharing system instead of ITS.

The modern computers of the era, such as the VAX or the 68020, had their own operating systems, but none of them were free software: you had to sign a nondisclosure agreement even to get an executable copy.

This meant that the first step in using a computer was to promise not to help your neighbor. A cooperating community was forbidden. The rule made by the owners of proprietary software was, “If you share with your neighbor, you are a pirate. If you want any changes, beg us to make them.”

The idea that the proprietary software social system—the system that says you are not allowed to share or change software—is antisocial, that it is unethical, that it is simply wrong, may come as a surprise to some readers. But what else could we say about a system based on dividing the public and keeping users helpless? Readers who find the idea

---

<sup>1</sup> The use of “hacker” to mean “security breaker” is a confusion on the part of the mass media. We hackers refuse to recognize that meaning, and continue using the word to mean someone who loves to program, someone who enjoys playful cleverness, or the combination of the two. See my article, “On Hacking,” at <http://stallman.org/articles/on-hacking.html>.

surprising may have taken the proprietary software social system as a given, or judged it on the terms suggested by proprietary software businesses. Software publishers have worked long and hard to convince people that there is only one way to look at the issue.

When software publishers talk about “enforcing” their “rights” or “stopping piracy,” what they actually *say* is secondary. The real message of these statements is in the unstated assumptions they take for granted, which the public is asked to accept without examination. Let’s therefore examine them.

One assumption is that software companies have an unquestionable natural right to own software and thus have power over all its users. (If this were a natural right, then no matter how much harm it does to the public, we could not object.) Interestingly, the US Constitution and legal tradition reject this view; copyright is not a natural right, but an artificial government-imposed monopoly that limits the users’ natural right to copy.

Another unstated assumption is that the only important thing about software is what jobs it allows you to do—that we computer users should not care what kind of society we are allowed to have.

A third assumption is that we would have no usable software (or would never have a program to do this or that particular job) if we did not offer a company power over the users of the program. This assumption may have seemed plausible, before the free software movement demonstrated that we can make plenty of useful software without putting chains on it.

If we decline to accept these assumptions, and judge these issues based on ordinary commonsense morality while placing the users first, we arrive at very different conclusions. Computer users should be free to modify programs to fit their needs, and free to share software, because helping other people is the basis of society.

There is no room here for an extensive statement of the reasoning behind this conclusion, so I refer the reader to the article *Why Software Should Not Have Owners*.

## A Stark Moral Choice

With my community gone, to continue as before was impossible. Instead, I faced a stark moral choice.

The easy choice was to join the proprietary software world, signing nondisclosure agreements and promising not to help my fellow hacker. Most likely I would also be developing software that was released under nondisclosure agreements, thus adding to the pressure on other people to betray their fellows too.

I could have made money this way, and perhaps amused myself writing code. But I knew that at the end of my career, I would look back on years of building walls to divide people, and feel I had spent my life making the world a worse place.

I had already experienced being on the receiving end of a nondisclosure agreement, when someone refused to give me and the MIT AI Lab the source code for the control program for our printer. (The lack of certain features in this program made use of the printer extremely frustrating.) So I could not tell myself that nondisclosure agreements were innocent. I was very angry when he refused to share with us; I could not turn around and do the same thing to everyone else.

Another choice, straightforward but unpleasant, was to leave the computer field. That way my skills would not be misused, but they would still be wasted. I would not be culpable for dividing and restricting computer users, but it would happen nonetheless.

So I looked for a way that a programmer could do something for the good. I asked myself, was there a program or programs that I could write, so as to make a community possible once again?

The answer was clear: what was needed first was an operating system. That is the crucial software for starting to use a computer. With an operating system, you can do many things; without one, you cannot run the computer at all. With a free operating system, we could again have a community of cooperating hackers—and invite anyone to join. And anyone would be able to use a computer without starting out by conspiring to deprive his or her friends.

As an operating system developer, I had the right skills for this job. So even though I could not take success for granted, I realized that I was elected to do the job. I chose to make the system compatible with Unix so that it would be portable, and so that Unix users could easily switch to it. The name GNU was chosen, following a hacker tradition, as a recursive acronym for “GNU’s Not Unix.”

An operating system does not mean just a kernel, barely enough to run other programs. In the 1970s, every operating system worthy of the name included command processors, assemblers, compilers, interpreters, debuggers, text editors, mailers, and much more. ITS had them, Multics had them, VMS had them, and Unix had them. The GNU operating system would include them too.

Later I heard these words, attributed to Hillel:<sup>2</sup>

If I am not for myself, who will be for me?  
 If I am only for myself, what am I?  
 If not now, when?

The decision to start the GNU Project was based on a similar spirit.

## Free as in Freedom

The term “free software” is sometimes misunderstood—it has nothing to do with price. It is about freedom. Here, therefore, is the definition of free software.

A program is free software, for you, a particular user, if:

- You have the freedom to run the program as you wish, for any purpose.
- You have the freedom to modify the program to suit your needs. (To make this freedom effective in practice, you must have access to the source code, since making changes in a program without having the source code is exceedingly difficult.)
- You have the freedom to redistribute copies, either gratis or for a fee.
- You have the freedom to distribute modified versions of the program, so that the community can benefit from your improvements.

---

<sup>2</sup> As an Atheist, I don’t follow any religious leaders, but I sometimes find I admire something one of them has said.

Since “free” refers to freedom, not to price, there is no contradiction between selling copies and free software. In fact, the freedom to sell copies is crucial: collections of free software sold on CD-ROMs are important for the community, and selling them is an important way to raise funds for free software development. Therefore, a program which people are not free to include on these collections is not free software.

Because of the ambiguity of “free,” people have long looked for alternatives, but no one has found a better term. The English language has more words and nuances than any other, but it lacks a simple, unambiguous, word that means “free,” as in freedom—“unfettered” being the word that comes closest in meaning. Such alternatives as “liberated,” “freedom,” and “open” have either the wrong meaning or some other disadvantage.

## GNU Software and the GNU System

Developing a whole system is a very large project. To bring it into reach, I decided to adapt and use existing pieces of free software wherever that was possible. For example, I decided at the very beginning to use  $\text{\TeX}$  as the principal text formatter; a few years later, I decided to use the X Window System rather than writing another window system for GNU.

Because of this decision, the GNU system is not the same as the collection of all GNU software. The GNU system includes programs that are not GNU software, programs that were developed by other people and projects for their own purposes, but which we can use because they are free software.

## Commencing the Project

In January 1984 I quit my job at MIT and began writing GNU software. Leaving MIT was necessary so that MIT would not be able to interfere with distributing GNU as free software. If I had remained on the staff, MIT could have claimed to own the work, and could have imposed their own distribution terms, or even turned the work into a proprietary software package. I had no intention of doing a large amount of work only to see it become useless for its intended purpose: creating a new software-sharing community.

However, Professor Winston, then the head of the MIT AI Lab, kindly invited me to keep using the lab’s facilities.

## The First Steps

Shortly before beginning the GNU Project, I heard about the Free University Compiler Kit, also known as VUCK. (The Dutch word for “free” is written with a *v*.) This was a compiler designed to handle multiple languages, including C and Pascal, and to support multiple target machines. I wrote to its author asking if GNU could use it.

He responded derisively, stating that the university was free but the compiler was not. I therefore decided that my first program for the GNU Project would be a multilanguage, multiplatform compiler.

Hoping to avoid the need to write the whole compiler myself, I obtained the source code for the Pastel compiler, which was a multiplatform compiler developed at Lawrence Livermore Lab. It supported, and was written in, an extended version of Pascal, designed to be a system-programming language. I added a C front end, and began porting it to the Motorola 68000 computer. But I had to give that up when I discovered that the compiler

needed many megabytes of stack space, while the available 68000 Unix system would only allow 64k.

I then realized that the Pastel compiler functioned by parsing the entire input file into a syntax tree, converting the whole syntax tree into a chain of “instructions,” and then generating the whole output file, without ever freeing any storage. At this point, I concluded I would have to write a new compiler from scratch. That new compiler is now known as GCC; none of the Pastel compiler is used in it, but I managed to adapt and use the C front end that I had written. But that was some years later; first, I worked on GNU Emacs.

## GNU Emacs

I began work on GNU Emacs in September 1984, and in early 1985 it was beginning to be usable. This enabled me to begin using Unix systems to do editing; having no interest in learning to use vi or ed, I had done my editing on other kinds of machines until then.

At this point, people began wanting to use GNU Emacs, which raised the question of how to distribute it. Of course, I put it on the anonymous ftp server on the MIT computer that I used. (This computer, `prep.ai.mit.edu`, thus became the principal GNU ftp distribution site; when it was decommissioned a few years later, we transferred the name to our new ftp server.) But at that time, many of the interested people were not on the Internet and could not get a copy by ftp. So the question was, what would I say to them?

I could have said, “Find a friend who is on the net and who will make a copy for you.” Or I could have done what I did with the original PDP-10 Emacs: tell them, “Mail me a tape and a SASE (self-addressed stamped envelope), and I will mail it back with Emacs on it.” But I had no job, and I was looking for ways to make money from free software. So I announced that I would mail a tape to whoever wanted one, for a fee of \$150. In this way, I started a free software distribution business, the precursor of the companies that today distribute entire Linux-based GNU systems.

## Is a Program Free for Every User?

If a program is free software when it leaves the hands of its author, this does not necessarily mean it will be free software for everyone who has a copy of it. For example, public domain software (software that is not copyrighted) is free software; but anyone can make a proprietary modified version of it. Likewise, many free programs are copyrighted but distributed under simple permissive licenses which allow proprietary modified versions.

The paradigmatic example of this problem is the X Window System. Developed at MIT, and released as free software with a permissive license, it was soon adopted by various computer companies. They added X to their proprietary Unix systems, in binary form only, and covered by the same nondisclosure agreement. These copies of X were no more free software than Unix was.

The developers of the X Window System did not consider this a problem—they expected and intended this to happen. Their goal was not freedom, just “success,” defined as “having many users.” They did not care whether these users had freedom, only about having many of them.

This led to a paradoxical situation where two different ways of counting the amount of freedom gave different answers to the question, “Is this program free?” If you judged based

on the freedom provided by the distribution terms of the MIT release, you would say that X was free software. But if you measured the freedom of the average user of X, you would have to say it was proprietary software. Most X users were running the proprietary versions that came with Unix systems, not the free version.

## Copyleft and the GNU GPL

The goal of GNU was to give users freedom, not just to be popular. So we needed to use distribution terms that would prevent GNU software from being turned into proprietary software. The method we use is called “copyleft.”<sup>3</sup>

Copyleft uses copyright law, but flips it over to serve the opposite of its usual purpose: instead of a means for restricting a program, it becomes a means for keeping the program free.

The central idea of copyleft is that we give everyone permission to run the program, copy the program, modify the program, and distribute modified versions—but not permission to add restrictions of their own. Thus, the crucial freedoms that define “free software” are guaranteed to everyone who has a copy; they become inalienable rights.

For an effective copyleft, modified versions must also be free. This ensures that work based on ours becomes available to our community if it is published. When programmers who have jobs as programmers volunteer to improve GNU software, it is copyleft that prevents their employers from saying, “You can’t share those changes, because we are going to use them to make our proprietary version of the program.”

The requirement that changes must be free is essential if we want to ensure freedom for every user of the program. The companies that privatized the X Window System usually made some changes to port it to their systems and hardware. These changes were small compared with the great extent of X, but they were not trivial. If making changes were an excuse to deny the users freedom, it would be easy for anyone to take advantage of the excuse.

A related issue concerns combining a free program with nonfree code. Such a combination would inevitably be nonfree; whichever freedoms are lacking for the nonfree part would be lacking for the whole as well. To permit such combinations would open a hole big enough to sink a ship. Therefore, a crucial requirement for copyleft is to plug this hole: anything added to or combined with a copylefted program must be such that the larger combined version is also free and copylefted.

The specific implementation of copyleft that we use for most GNU software is the GNU General Public License, or GNU GPL for short. We have other kinds of copyleft that are used in specific circumstances. GNU manuals are copylefted also, but use a much simpler kind of copyleft, because the complexity of the GNU GPL is not necessary for manuals.<sup>4</sup>

## The Free Software Foundation

As interest in using Emacs was growing, other people became involved in the GNU Project, and we decided that it was time to seek funding once again. So in 1985 we created the Free

---

<sup>3</sup> In 1984 or 1985, Don Hopkins (a very imaginative fellow) mailed me a letter. On the envelope he had written several amusing sayings, including this one: “Copyleft—all rights reversed.” I used the word “copyleft” to name the distribution concept I was developing at the time.

<sup>4</sup> We now use the *GNU Free Documentation License* for documentation.

Software Foundation (FSF), a tax-exempt charity for free software development. The FSF also took over the Emacs tape distribution business; later it extended this by adding other free software (both GNU and non-GNU) to the tape, and by selling free manuals as well.

Most of the FSF's income used to come from sales of copies of free software and of other related services (CD-ROMs of source code, CD-ROMs with binaries, nicely printed manuals, all with the freedom to redistribute and modify), and Deluxe Distributions (distributions for which we built the whole collection of software for the customer's choice of platform). Today the FSF still sells manuals and other gear, but it gets the bulk of its funding from members' dues. You can join the FSF at <http://fsf.org/join>.

Free Software Foundation employees have written and maintained a number of GNU software packages. Two notable ones are the C library and the shell. The GNU C library is what every program running on a GNU/Linux system uses to communicate with Linux. It was developed by a member of the Free Software Foundation staff, Roland McGrath. The shell used on most GNU/Linux systems is BASH, the Bourne Again Shell,<sup>5</sup> which was developed by FSF employee Brian Fox.

We funded development of these programs because the GNU Project was not just about tools or a development environment. Our goal was a complete operating system, and these programs were needed for that goal.

## Free Software Support

The free software philosophy rejects a specific widespread business practice, but it is not against business. When businesses respect the users' freedom, we wish them success.

Selling copies of Emacs demonstrates one kind of free software business. When the FSF took over that business, I needed another way to make a living. I found it in selling services relating to the free software I had developed. This included teaching, for subjects such as how to program GNU Emacs and how to customize GCC, and software development, mostly porting GCC to new platforms.

Today each of these kinds of free software business is practiced by a number of corporations. Some distribute free software collections on CD-ROM; others sell support at levels ranging from answering user questions, to fixing bugs, to adding major new features. We are even beginning to see free software companies based on launching new free software products.

Watch out, though—a number of companies that associate themselves with the term “open source” actually base their business on nonfree software that works with free software. These are not free software companies, they are proprietary software companies whose products tempt users away from freedom. They call these programs “value-added packages,” which shows the values they would like us to adopt: convenience above freedom. If we value freedom more, we should call them “freedom-subtracted” packages.

## Technical Goals

The principal goal of GNU is to be free software. Even if GNU had no technical advantage over Unix, it would have a social advantage, allowing users to cooperate, and an ethical advantage, respecting the user's freedom.

---

<sup>5</sup> “Bourne Again Shell” is a play on the name “Bourne Shell,” which was the usual shell on Unix.



But it was natural to apply the known standards of good practice to the work—for example, dynamically allocating data structures to avoid arbitrary fixed size limits, and handling all the possible 8-bit codes wherever that made sense.

In addition, we rejected the Unix focus on small memory size, by deciding not to support 16-bit machines (it was clear that 32-bit machines would be the norm by the time the GNU system was finished), and to make no effort to reduce memory usage unless it exceeded a megabyte. In programs for which handling very large files was not crucial, we encouraged programmers to read an entire input file into core, then scan its contents without having to worry about I/O.

These decisions enabled many GNU programs to surpass their Unix counterparts in reliability and speed.

## Donated Computers

As the GNU Project's reputation grew, people began offering to donate machines running Unix to the project. These were very useful, because the easiest way to develop components of GNU was to do it on a Unix system, and replace the components of that system one by one. But they raised an ethical issue: whether it was right for us to have a copy of Unix at all.

Unix was (and is) proprietary software, and the GNU Project's philosophy said that we should not use proprietary software. But, applying the same reasoning that leads to the conclusion that violence in self defense is justified, I concluded that it was legitimate to use a proprietary package when that was crucial for developing a free replacement that would help others stop using the proprietary package.

But, even if this was a justifiable evil, it was still an evil. Today we no longer have any copies of Unix, because we have replaced them with free operating systems. If we could not replace a machine's operating system with a free one, we replaced the machine instead.

## The GNU Task List

As the GNU Project proceeded, and increasing numbers of system components were found or developed, eventually it became useful to make a list of the remaining gaps. We used it to recruit developers to write the missing pieces. This list became known as the GNU Task List. In addition to missing Unix components, we listed various other useful software and documentation projects that, we thought, a truly complete system ought to have.

Today,<sup>6</sup> hardly any Unix components are left in the GNU Task List—those jobs had been done, aside from a few inessential ones. But the list is full of projects that some might call “applications.” Any program that appeals to more than a narrow class of users would be a useful thing to add to an operating system.

Even games are included in the task list—and have been since the beginning. Unix included games, so naturally GNU should too. But compatibility was not an issue for games, so we did not follow the list of games that Unix had. Instead, we listed a spectrum of different kinds of games that users might like.

---

<sup>6</sup> That was written in 1998. In 2009 we no longer maintain a long task list. The community develops free software so fast that we can't even keep track of it all. Instead, we have a list of High Priority Projects, a much shorter list of projects we really want to encourage people to write.

## The GNU Library GPL

The GNU C library uses a special kind of copyleft called the GNU Library General Public License,<sup>7</sup> which gives permission to link proprietary software with the library. Why make this exception?

It is not a matter of principle; there is no principle that says proprietary software products are entitled to include our code. (Why contribute to a project predicated on refusing to share with us?) Using the LGPL for the C library, or for any library, is a matter of strategy.

The C library does a generic job; every proprietary system or compiler comes with a C library. Therefore, to make our C library available only to free software would not have given free software any advantage—it would only have discouraged use of our library.

One system is an exception to this: on the GNU system (and this includes GNU/Linux), the GNU C library is the only C library. So the distribution terms of the GNU C library determine whether it is possible to compile a proprietary program for the GNU system. There is no ethical reason to allow proprietary applications on the GNU system, but strategically it seems that disallowing them would do more to discourage use of the GNU system than to encourage development of free applications. That is why using the Library GPL is a good strategy for the C library.

For other libraries, the strategic decision needs to be considered on a case-by-case basis. When a library does a special job that can help write certain kinds of programs, then releasing it under the GPL, limiting it to free programs only, is a way of helping other free software developers, giving them an advantage against proprietary software.

Consider GNU Readline, a library that was developed to provide command-line editing for BASH. Readline is released under the ordinary GNU GPL, not the Library GPL. This probably does reduce the amount Readline is used, but that is no loss for us. Meanwhile, at least one useful application has been made free software specifically so it could use Readline, and that is a real gain for the community.

Proprietary software developers have the advantages money provides; free software developers need to make advantages for each other. I hope some day we will have a large collection of GPL-covered libraries that have no parallel available to proprietary software, providing useful modules to serve as building blocks in new free software, and adding up to a major advantage for further free software development.

## Scratching an Itch?

Eric Raymond<sup>8</sup> says that “Every good work of software starts by scratching a developer’s personal itch.”<sup>9</sup> Maybe that happens sometimes, but many essential pieces of GNU software were developed in order to have a complete free operating system. They come from a vision and a plan, not from impulse.

---

<sup>7</sup> This license is now called the GNU Lesser General Public License, to avoid giving the idea that all libraries ought to use it.

<sup>8</sup> Eric Raymond is a prominent open source advocate; see *Why Open Source Misses the Point of Free Software*.

<sup>9</sup> Eric S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, rev. ed. (Sebastopol, Calif.: O’Reilly, 2001), p. 23.

For example, we developed the GNU C library because a Unix-like system needs a C library, BASH because a Unix-like system needs a shell, and GNU tar because a Unix-like system needs a tar program. The same is true for my own programs—the GNU C compiler, GNU Emacs, GDB and GNU Make.

Some GNU programs were developed to cope with specific threats to our freedom. Thus, we developed gzip to replace the Compress program, which had been lost to the community because of the LZW patents. We found people to develop LessTif, and more recently started GNOME and Harmony, to address the problems caused by certain proprietary libraries (see below). We are developing the GNU Privacy Guard to replace popular nonfree encryption software, because users should not have to choose between privacy and freedom.

Of course, the people writing these programs became interested in the work, and many features were added to them by various people for the sake of their own needs and interests. But that is not why the programs exist.

## Unexpected Developments

At the beginning of the GNU Project, I imagined that we would develop the whole GNU system, then release it as a whole. That is not how it happened.

Since each component of the GNU system was implemented on a Unix system, each component could run on Unix systems long before a complete GNU system existed. Some of these programs became popular, and users began extending them and porting them—to the various incompatible versions of Unix, and sometimes to other systems as well.

The process made these programs much more powerful, and attracted both funds and contributors to the GNU Project. But it probably also delayed completion of a minimal working system by several years, as GNU developers' time was put into maintaining these ports and adding features to the existing components, rather than moving on to write one missing component after another.

## The GNU Hurd

By 1990, the GNU system was almost complete; the only major missing component was the kernel. We had decided to implement our kernel as a collection of server processes running on top of Mach. Mach is a microkernel developed at Carnegie Mellon University and then at the University of Utah; the GNU Hurd is a collection of servers (i.e., a herd of GNUs) that run on top of Mach, and do the various jobs of the Unix kernel. The start of development was delayed as we waited for Mach to be released as free software, as had been promised.

One reason for choosing this design was to avoid what seemed to be the hardest part of the job: debugging a kernel program without a source-level debugger to do it with. This part of the job had been done already, in Mach, and we expected to debug the Hurd servers as user programs, with GDB. But it took a long time to make that possible, and the multithreaded servers that send messages to each other have turned out to be very hard to debug. Making the Hurd work solidly has stretched on for many years.

## Alix

The GNU kernel was not originally supposed to be called the Hurd. Its original name was Alix—named after the woman who was my sweetheart at the time. She, a Unix system

administrator, had pointed out how her name would fit a common naming pattern for Unix system versions; as a joke, she told her friends, “Someone should name a kernel after me.” I said nothing, but decided to surprise her with a kernel named Alix.

It did not stay that way. Michael (now Thomas) Bushnell, the main developer of the kernel, preferred the name Hurd, and redefined Alix to refer to a certain part of the kernel—the part that would trap system calls and handle them by sending messages to Hurd servers.

Later, Alix and I broke up, and she changed her name; independently, the Hurd design was changed so that the C library would send messages directly to servers, and this made the Alix component disappear from the design.

But before these things happened, a friend of hers came across the name Alix in the Hurd source code, and mentioned it to her. So she did have the chance to find a kernel named after her.

## Linux and GNU/Linux

The GNU Hurd is not suitable for production use, and we don’t know if it ever will be. The capability-based design has problems that result directly from the flexibility of the design, and it is not clear solutions exist.

Fortunately, another kernel is available. In 1991, Linus Torvalds developed a Unix-compatible kernel and called it Linux. In 1992, he made Linux free software; combining Linux with the not-quite-complete GNU system resulted in a complete free operating system. (Combining them was a substantial job in itself, of course.) It is due to Linux that we can actually run a version of the GNU system today.

We call this system version GNU/Linux, to express its composition as a combination of the GNU system with Linux as the kernel.

## Challenges in Our Future

We have proved our ability to develop a broad spectrum of free software. This does not mean we are invincible and unstoppable. Several challenges make the future of free software uncertain; meeting them will require steadfast effort and endurance, sometimes lasting for years. It will require the kind of determination that people display when they value their freedom and will not let anyone take it away.

The following four sections discuss these challenges.

### Secret Hardware

Hardware manufacturers increasingly tend to keep hardware specifications secret. This makes it difficult to write free drivers so that Linux and XFree86 can support new hardware. We have complete free systems today, but we will not have them tomorrow if we cannot support tomorrow’s computers.

There are two ways to cope with this problem. Programmers can do reverse engineering to figure out how to support the hardware. The rest of us can choose the hardware that is supported by free software; as our numbers increase, secrecy of specifications will become a self-defeating policy.

Reverse engineering is a big job; will we have programmers with sufficient determination to undertake it? Yes—if we have built up a strong feeling that free software is a matter

of principle, and nonfree drivers are intolerable. And will large numbers of us spend extra money, or even a little extra time, so we can use free drivers? Yes, if the determination to have freedom is widespread.

[2008 note: this issue extends to the BIOS as well. There is a free BIOS, coreboot; the problem is getting specs for machines so that coreboot can support them.]

## Nonfree Libraries

A nonfree library that runs on free operating systems acts as a trap for free software developers. The library's attractive features are the bait; if you use the library, you fall into the trap, because your program cannot usefully be part of a free operating system. (Strictly speaking, we could include your program, but it won't *run* with the library missing.) Even worse, if a program that uses the proprietary library becomes popular, it can lure other unsuspecting programmers into the trap.

The first instance of this problem was the Motif toolkit, back in the 80s. Although there were as yet no free operating systems, it was clear what problem Motif would cause for them later on. The GNU Project responded in two ways: by asking individual free software projects to support the free X Toolkit widgets as well as Motif, and by asking for someone to write a free replacement for Motif. The job took many years; LessTif, developed by the Hungry Programmers, became powerful enough to support most Motif applications only in 1997.

Between 1996 and 1998, another nonfree GUI toolkit library, called Qt, was used in a substantial collection of free software, the desktop KDE.

Free GNU/Linux systems were unable to use KDE, because we could not use the library. However, some commercial distributors of GNU/Linux systems who were not strict about sticking with free software added KDE to their systems—producing a system with more capabilities, but less freedom. The KDE group was actively encouraging more programmers to use Qt, and millions of new “Linux users” had never been exposed to the idea that there was a problem in this. The situation appeared grim.

The free software community responded to the problem in two ways: GNOME and Harmony.

GNOME, the GNU Network Object Model Environment, is GNU's desktop project. Started in 1997 by Miguel de Icaza, and developed with the support of Red Hat Software, GNOME set out to provide similar desktop facilities, but using free software exclusively. It has technical advantages as well, such as supporting a variety of languages, not just C++. But its main purpose was freedom: not to require the use of any nonfree software.

Harmony is a compatible replacement library, designed to make it possible to run KDE software without using Qt.

In November 1998, the developers of Qt announced a change of license which, when carried out, should make Qt free software. There is no way to be sure, but I think that this was partly due to the community's firm response to the problem that Qt posed when it was nonfree. (The new license is inconvenient and inequitable, so it remains desirable to avoid using Qt.)

[Subsequent note: in September 2000, Qt was rereleased under the GNU GPL, which essentially solved this problem.]

How will we respond to the next tempting nonfree library? Will the whole community understand the need to stay out of the trap? Or will many of us give up freedom for convenience, and produce a major problem? Our future depends on our philosophy.

## Software Patents

The worst threat we face comes from software patents, which can put algorithms and features off limits to free software for up to 20 years. The LZW compression algorithm patents were applied for in 1983, and we still cannot release free software to produce proper compressed GIFs. [As of 2009 they have expired.] In 1998, a free program to produce MP3 compressed audio was removed from distribution under threat of a patent suit.

There are ways to cope with patents: we can search for evidence that a patent is invalid, and we can look for alternative ways to do a job. But each of these methods works only sometimes; when both fail, a patent may force all free software to lack some feature that users want. What will we do when this happens?

Those of us who value free software for freedom's sake will stay with free software anyway. We will manage to get work done without the patented features. But those who value free software because they expect it to be technically superior are likely to call it a failure when a patent holds it back. Thus, while it is useful to talk about the practical effectiveness of the “bazaar” model of development, and the reliability and power of some free software, we must not stop there. We must talk about freedom and principle.

## Free Documentation

The biggest deficiency in our free operating systems is not in the software—it is the lack of good free manuals that we can include in our systems. Documentation is an essential part of any software package; when an important free software package does not come with a good free manual, that is a major gap. We have many such gaps today.

Free documentation, like free software, is a matter of freedom, not price. The criterion for a free manual is pretty much the same as for free software: it is a matter of giving all users certain freedoms. Redistribution (including commercial sale) must be permitted, online and on paper, so that the manual can accompany every copy of the program.

Permission for modification is crucial too. As a general rule, I don't believe that it is essential for people to have permission to modify all sorts of articles and books. For example, I don't think you or I are obliged to give permission to modify articles like this one, which describe our actions and our views.

But there is a particular reason why the freedom to modify is crucial for documentation for free software. When people exercise their right to modify the software, and add or change its features, if they are conscientious they will change the manual, too—so they can provide accurate and usable documentation with the modified program. A nonfree manual, which does not allow programmers to be conscientious and finish the job, does not fill our community's needs.

Some kinds of limits on how modifications are done pose no problem. For example, requirements to preserve the original author's copyright notice, the distribution terms, or the list of authors, are OK. It is also no problem to require modified versions to include notice that they were modified, even to have entire sections that may not be deleted or

changed, as long as these sections deal with nontechnical topics. These kinds of restrictions are not a problem because they don't stop the conscientious programmer from adapting the manual to fit the modified program. In other words, they don't block the free software community from making full use of the manual.

However, it must be possible to modify all the *technical* content of the manual, and then distribute the result in all the usual media, through all the usual channels; otherwise, the restrictions do obstruct the community, the manual is not free, and we need another manual.

Will free software developers have the awareness and determination to produce a full spectrum of free manuals? Once again, our future depends on philosophy.

## We Must Talk about Freedom

Estimates today are that there are ten million users of GNU/Linux systems such as Debian GNU/Linux and Red Hat "Linux." Free software has developed such practical advantages that users are flocking to it for purely practical reasons.

The good consequences of this are evident: more interest in developing free software, more customers for free software businesses, and more ability to encourage companies to develop commercial free software instead of proprietary software products.

But interest in the software is growing faster than awareness of the philosophy it is based on, and this leads to trouble. Our ability to meet the challenges and threats described above depends on the will to stand firm for freedom. To make sure our community has this will, we need to spread the idea to the new users as they come into the community.

But we are failing to do so: the efforts to attract new users into our community are far outstripping the efforts to teach them the civics of our community. We need to do both, and we need to keep the two efforts in balance.

## "Open Source"

Teaching new users about freedom became more difficult in 1998, when a part of the community decided to stop using the term "free software" and say "open source software" instead.

Some who favored this term aimed to avoid the confusion of "free" with "gratis"—a valid goal. Others, however, aimed to set aside the spirit of principle that had motivated the free software movement and the GNU Project, and to appeal instead to executives and business users, many of whom hold an ideology that places profit above freedom, above community, above principle. Thus, the rhetoric of "open source" focuses on the potential to make high-quality, powerful software, but shuns the ideas of freedom, community, and principle.

The "Linux" magazines are a clear example of this—they are filled with advertisements for proprietary software that works with GNU/Linux. When the next Motif or Qt appears, will these magazines warn programmers to stay away from it, or will they run ads for it?

The support of business can contribute to the community in many ways; all else being equal, it is useful. But winning their support by speaking even less about freedom and principle can be disastrous; it makes the previous imbalance between outreach and civics education even worse.

“Free software” and “open source” describe the same category of software, more or less, but say different things about the software, and about values. The GNU Project continues to use the term “free software,” to express the idea that freedom, not just technology, is important.

## **Try!**

Yoda’s aphorism (“There is no ‘try’”) sounds neat, but it doesn’t work for me. I have done most of my work while anxious about whether I could do the job, and unsure that it would be enough to achieve the goal if I did. But I tried anyway, because there was no one but me between the enemy and my city. Surprising myself, I have sometimes succeeded.

Sometimes I failed; some of my cities have fallen. Then I found another threatened city, and got ready for another battle. Over time, I’ve learned to look for threats and put myself between them and my city, calling on other hackers to come and join me.

Nowadays, often I’m not the only one. It is a relief and a joy when I see a regiment of hackers digging in to hold the line, and I realize, this city may survive—for now. But the dangers are greater each year, and now Microsoft has explicitly targeted our community. We can’t take the future of freedom for granted. Don’t take it for granted! If you want to keep your freedom, you must be prepared to defend it.