

まえがき

本書は、経験のある(熟練でなくても学習意欲が旺盛な)プログラマで、Linux ソフトウェアの開発、または他のプラットフォームから Linux へのソフトウェアのポータリングを行う人々を対象にしている。これこそ、我々が Linux のプログラミングを学習したときにあればよかったのと思う本であり、現在机の上にリファレンスとして置いてある本である。最初の 3 章を書いた時点で、我々はすでに草稿をリファレンスマニュアルとして作業中に使用していた。

Linux は、Unix に似せて設計されている。本書は、Unix プログラミングの基本とスタイルにおける優れたバックグラウンドを提供する。Linux は、基本的には Unix と変わるところはないが、Linux を無視した Unix プログラミングリファレンスに頼っているプログラマを繰り返すまずかせるのに十分な程度には異なっている。したがって、本書はまさに Linux の視点から書かれたプログラミングガイドである。

Linux には、ダイレクト画面アクセス機能(20 章を参照)のような独自の拡張もあり、また S-Lang ライブラリ(22 章を参照)のように、他のシステムにおいてよりも頻繁に使用される機能がある。本書は、読者が Linux を真に活用するプログラムを作成できるよう、それらの拡張や機能の多くを説明する。

読者が C プログラマではあるが、Unix も Linux も知らない場合、本書を最初から終わりまで読んで、例に取り組みむことにより、有能な Linux プログラマになるための軌道にうまく乗るはずである。システム固有のドキュメントなどの助けを借りれば、あらゆるバージョンの Unix への移行が容易になる。

すでに熟練の Unix プログラマである場合、本書により Linux への移行が容易になることがわかるだろう。我々は、読者がまさに必要とする情報を簡単に見つけられるように、熱心に努力してきた。また、熟練の Unix プログラマでさえときに足をすくわれる、プロセスとセッショングループ、ジョブ制御、および tty 処理のようなトピックを慎重かつ明確に説明する。

すでに Linux プログラマである場合、本書は混乱を招くトピックを明確に説明し、読者のプログラミング作業の多くを容易にする。このタイプの読者は各章の基礎となる最小限の Linux の知識をすでに持っているので、ほとんどすべての章がそれだけで役に立つ。どのように経験を積んでいても、ここでの資料は、手近に置いておくことの価値を認めるだろう。

本書は、明らかに特定のオペレーティングシステムに固有なので、通常の Unix テキストとは異なる。本書では、「BSD はこれをこの方法で行い、SVR4 は他の方法で行い、HPUX は独自の方法でそれを処理し、SGI もまた独自の方法がある。これらのひとつひとつについて説明し、整

理は読者にゆだねることにする」といって、初心者を混乱させる必要がない。我々は経験を通じて、いずれかの Unix ライクなシステムを十分に習得すれば、他のシステムは学習しやすいことを知っている。

本書は、Linux プログラミングの詳細をすべてカバーするものではない。たとえば X Window System は、すべての Linux または Unix プラットフォームで同一なので説明しない。同様に、ANSI C で仕様が決定されている基本インターフェイスは他にかなりよい書籍があるので、本書では説明しない。DOS、Windows、または Macintosh など、他のシステムの C プログラマから Linux の C プログラマになるために必要な情報は、極度に冗長になることのない程度に説明する。Linux のために利用可能な他の多くのプログラミング言語は説明しないし、サポートされるシステムのどれを使用しても同一であるグラフィカルプログラミングライブラリは説明しない。その代わり、それらの領域に特化した書籍を紹介する。

『プログラミング Linux』は、以下の 4 つの部分からなる。

最初の部分は、Linux のオペレーティングシステム、ライセンス条件、ドキュメントおよび環境を紹介する。

2 番目の部分は、開発環境の最も重要な一面であるコンパイラ、リンカ、およびローダ、さらに他のプラットフォームではあまり使用されていないデバッグツールをいくつか説明する。

3 番目の部分は本書の中心部分で、カーネルや主にカーネルへのインターフェイスとして機能するシステムライブラリへのインターフェイスを説明する。この部分の最後の 3 つの章だけが特に Linux 固有で、それ以外の多くの部分は Linux の視点からの Unix プログラミングを説明する。

4 番目の部分は知識の総仕上げで、よりカーネルから独立したインターフェイスを提供する重要なライブラリをいくつか説明する。これらのライブラリは、正確に言うと Linux 固有ではないが、一部は他のシステムよりも Linux システムでより頻繁に使用される。

Linux または Unix プログラミングをすでに熟知している場合、本書をどのような順序で読んでもかまわない。興味のない章を読まなければならないと感じる必要はない。Linux も Unix もよく知らない場合は多くの章が役に立つが、1、2、4、5、8、9、10、11 の各章は、他の章を読むために必要な知識の多くを与えるので、これらの章を最初に読むといいだろう。特に 9、10、11 の各章は、Unix および Linux プログラミングモデルの中核を形成している。

以下の書籍は、あちらこちらで互いに少しずつ重複しているかもしれないが、より簡単に、またはより高度に、または関連するトピックについて説明することで、本書を補足する。

The C Programming Language, second edition [Kernighan, 1988] は、ANSI 標準の C プログラミングを簡潔に教えてくれるが、オペレーティングシステムにあまり言及していない。プログラミングの知識のあるユーザー、または「知識の豊富な同僚」の助けを得られ

るユーザー向けである。

Practical C Programming [Oualling, 1993]は、これまでにプログラミングの経験がない人々のために考案された段階的でありていきやすい方法で、C プログラミングとスタイルを教えてくれる。

Programming with GNU Software [Loukides, 1997]は GNU プログラミング環境の入門書で、C コンパイラ、デバッガ、make ユーティリティ、および RCS ソースコード制御システムの実行方法についての章を含む。

Advanced Programming in the UNIX Environment [Stevens, 1992]は、重要な Unix および Unix ライクなシステムを説明するが、これは Linux 以前に書かれたものである。この本には、本書『プログラミング Linux』の最後の 2 つの部分とよく似た資料、つまりシステムコールと共有ライブラリが収録されている。また、多くの例を提供し、多様な Unix バージョン間の差異を説明する。

Unix Network Programming [Stevens, 1990]は、少なくともこれを書いている時点で Linux では利用できない、古いタイプのネットワークを詳細に説明している。この本を読む一方で、バークレーソケットインターフェイス(16 章を参照)だけを使用して移植性を最大にすること。この本は、Linux のネットワークプログラムを Unix のいずれかのブランドにポーティングするために、わずかな変更を行う必要がある場合に便利かもしれない。

関連書籍の広範な一覧については、参考文献または <http://www.awl.com/csing/books/lad/biblio.html> を参照すること。

本書にあるソースコードは、すべて執筆のかたわらでテストした使用可能なサンプルである。本書のソースコードは、<http://www.awl.com/cseng/books/lad/src/> および <ftp://ftp.awl.com/cseng/books/lad/src/> で、電子的な形態で入手できる。明確にするために、短いソースコードセグメントは、考えるすべてのエラーをチェックするのではなく、システムがどのように機能するかを示す生じやすいエラーだけをチェックする。ただし、本書や Web サイトおよび FTP サイトの中の完全なプログラムは、合理的なエラーをすべてチェックするよう努めた(が、我々は完璧ではない)。

本書は、どの関数を使用し、それらが互いにどのように組み合わせられるかを説明する。あわせてリファレンス情報^{*1}の使用方法も習得することをお勧めするが、その大部分はシステムに付属している。

我々は、lad-comments@awl.com へのコメントを歓迎する。コメントは読むが、個別にお返事を差し上げることは約束できない。

Linux は、急速に発展しているオペレーティングシステムで、本書を読むときには、いくつか

*1 第 3 章では、Linux に関連したトピックに関する情報の入手方法を紹介している。

の事実は(ほとんどないことを祈るが)おそらく変更されているだろう。本書は、Linux 2.0.30 と Red Hat Software の Red Hat Linux 4.2 ディストリビューションに含まれる、C ライブラリバージョン 5.3.12 を参照して書かれている。また、ソースコードのテストは、Red Hat Linux 5.0 ディストリビューションに含まれる C ライブラリバージョン 6(glibc 2.0.5)でも行った。

読者の協力を得て、誤植と変更点の一覧を Web 上の <http://www.awl.com/csing/books/lad/errata.html> および FTP の <ftp://ftp.awl.com/csing/books/lad/errata/> 経由で管理することにする。

テクニカルレビューをしてくださった人々のひとりひとりに対し、その時間と慎重な意見に感謝したい。彼らの意見によって、本書は強力になった。特に Linus Torvalds、Alan Cox、および Ted Ts'o には、時間を取って質問に答えていただいたことに感謝する。

Kim Johnson と Brigid Nogueira には、特別に感謝したい。彼女らの絶え間ざる忍耐がなかったら、本書は書かれなかっただろう。

第1章

Linux開発の沿革

Linux はさまざまな意味に使用される。技術的に最も正確な定義は、以下のとおりである。

Linux は、自由に配布可能な、Unix ライクなオペレーティングシステムカーネルである。

ただし多くの人々は、Linux を Linux カーネルをベースにしたオペレーティングシステムという意味で使う。

Linux は、自由に配布可能な、Unix ライクなオペレーティングシステムで、カーネル、システムツール、アプリケーション、および完全な開発環境を含む。

読者はカーネルだけではなく、オペレーティングシステム全体をプログラムするので、本書では、2 番目の定義を使用する。

Linux(2 番目の定義による)が推奨するインターフェイス(本書で説明するインターフェイス)は、利用可能なほとんどすべてのバージョンの Unix、および多くの Unix クローンによってサポートされているので、プログラムを移植するための優れたプラットフォームを提供する。本書の内容を学習すれば、余分な作業をほとんどせずに、自分のプログラムをほとんどすべての Unix および Unix ライクなシステムに移植できる。

一方、Linux で作業をすると、Linux だけを使用して移植に悩むのをやめようと思うかもしれない。

Linux は、単なるもう1つの Unix ではない。これは、プログラムをポーティングするための優れたプラットフォームという以上に、アプリケーションを作成して実行するのに優れたプラットフォームでもある。Linux の長所の一部を以下に挙げる。

Linux カーネルの安定バージョンが、実験バージョンから明確に分離されており、両方が一般に公開されている。生産環境には既知の安定したシステムを使用できる。同時に、実験的な Linux バージョンが頻繁にリリースされ、望むならオペレーティングシステムの開発に追随することや、参加することすら可能である。

基本オペレーティングシステム全体のソースコードが検査、使用、および変更のためにさえ利用可能である。

Linux は、1つの企業ではなく、私的な個人の団体によって制御されている。技術的な決定は、マーケティング上の配慮ではなく技術的利点によって、技術指向の人々の手で行われ

る。これにより、Linux 開発の動きは非常に速く、他の補足的テクノロジーと歩調を合わせている。

Linux の開発者やユーザーのコミュニティは、大規模で一般に新参者を熱心に助ける。

1.1 フリーなUnixソフトウェアの簡単な沿革

Linux 用ソフトウェアを開発することがどのような意味を持つかを理解しようとするなら、簡単な歴史の講義が必要である。ここで述べる講義は、Linux システムの最も重要な要素だけを理解するように省略され、必要な部分を強調している。詳細で、より広範にわたる説明については、*A Quarter Century of Unix* [Salus, 1994] という本を通して読んでいただきたい。

コンピューティングの初期のころは、ソフトウェアはハードウェアの機能以上のものとはほとんど見られなかった。販売しようとしていたのはハードウェアだったので、企業はソフトウェアをシステムに付けて提供していた。拡張、新しいアルゴリズム、および新しい概念が学生、大学教授、および企業の研究者の間で自由に流通していた。

企業がソフトウェアを知的財産として認識するのに、そう時間はかからなかった。企業は自社のソフトウェアテクノロジーに著作権を主張し、ソースコードの配布を制限するようになった。公的財産として見られてきた革新的技術が、厳しく保護された企業資産となり、コンピュータソフトウェア開発の文化が変化した。

マサチューセッツ工科大学(MIT)の Richard Stallman は、世界中のどこでも、革新的ソフトウェア技術が企業の野心によって制御されるのを好まなかった。この進展に対する彼の答は、マサチューセッツ州ケンブリッジに Free Software Foundation(FSF) を設立することだった。FSF の目標は、フリーに再配布可能なソフトウェアの開発と使用を促進することである。

しかしながら、この文脈において「フリー」という語の用法が大きな混乱を招いた。Richard Stallman は、フリーを自由という意味で使用したのであって、無料という意味ではなかった。彼は、ソフトウェアとそれに関連するドキュメントはソースコードとともに入手可能で、再配布に制約を設けるべきではないと確信していた。この理想を追求するために、彼は他の人々の支援を得て、General Public License(GPL) を作成したが、この完全なコピーが本書の Appendix C にある。

GPL には、3 つの主要なポイントがある。

1. GPL を持つソフトウェアを受け取った人はすべて、(配達料金以上の) 追加料金なしにソフトウェアのソースコードを取得する権利を有する。
2. GPL を持つソフトウェアから派生したソフトウェアは、再配布のライセンス条件として、GPL を維持しなければならない。

3. GPL を持つソフトウェアを所持するものは、GPL に抵触しない条件の下で、そのソフトウェアを再配布する権利を有する。

これらのライセンス条件で注意すべき重要なポイントは、価格に言及していないことである（ソースが別料金の品目になれないことを除いて）。GPL を持つソフトウェアは、顧客にいかなる価格でも販売できる。ただし、それらの顧客は、ソースコードも含め、好きなようにそのソフトウェアを再配布する権利を有する。インターネットの出現とともに、この権利は GPL を持つソフトウェアの価格を下げるという効果を持ち、多くの場合ゼロにさえなったが、企業はそれでもなお、GPL を持つソフトウェアおよびサポートなどのソフトウェアの補足として考えられたサービスを販売できる。

GPL で最も議論を呼ぶ部分は、2 番目のポイントである。GPL を持つソフトウェアから派生したソフトウェアも、また GPL を持つ必要がある。批判者はこの条項のために GPL をウィルスと呼ぶが、支持者はこの条項は GPL の最大の強みの 1 つであると主張する。これは企業が GPL を持つソフトウェアを入手して機能を追加し、その結果を同社だけのパッケージとすることを防ぐ。

FSF がスポンサーとなっている主なプロジェクトが、GNU(GNU's Not Unix)プロジェクトだ。これは、自由に配布できる Unix ライクなオペレーティングシステムを作成することを目標としている。開始されたとき、GNU プロジェクトには自由に配布できる高品質なソフトウェアがほとんどなかったため、プロジェクトに寄与する人々は、オペレーティングシステムそのものよりもシステムのためのアプリケーションやツールを作成することから始めた。GPL もまた FSF によって作成されたので、GNU オペレーティングシステムの主要コンポーネントの多くは GPL を持つが、数年の間に GNU プロジェクトは多くのソフトウェアパッケージ、たとえば、X Window System や TeX 植字プログラム、および Perl 言語などを採用し、これらは他のライセンスの下で自由に再配布できる。

GNU プロジェクトの結果として、いくつかの主要なパッケージと多くの小さなパッケージが作成された。主要なものには、Emacs エディタ、gcc コンパイラ、bash および gawk(GNU の awk) などがある。小さなパッケージには、ユーザーが Unix システムに期待する高品質のシェルユーティリティやテキスト操作プログラムなどがある。

1.2 Linuxの開発

1991 年ヘルシンキ大学の学生だった Linus Torvalds は、低レベルの 80386 プログラミングを独学で勉強するためにプロジェクトを開始した。当時、彼は Andrew Tanenbaum 設計の Minix オペレーティングシステムを使っていたので、システムの Minix システムコールおよびファイルシステム構造との互換性を保つことで、作業がはるかに容易になった。彼は Linux カーネルの最初のバージョンをかなり制限したライセンスの下でインターネットにリリースしたが、すぐにライセンスを GPL に変更するように説得された。

GPL ライセンスと Linux カーネルの初期の機能性の組み合わせにより、他の開発者はカーネルの開発に協力することを納得した。当時休眠状態の GNU C ライブラリプロジェクトから派生した C ライブラリの実装がリリースされ、ネイティブなユーザーアプリケーションが構築できるようになった。gcc、Emacs、および bash のネイティブ版が、すぐこれに続いた。1992 年始めには、過度の作業を行うことなく、多くの Intel マシンに Linux 0.95 をインストールして起動することができた。

Linux プロジェクトは、最初から GNU プロジェクトと緊密に関連付けられている。GNU プロジェクトのソーススペースは、Linux コミュニティがシステム全体を構築するためのきわめて重要なリソースとなった。Linux ベースのシステムのかなりの部分が、カリフォルニア大学バークレー校や X コンソーシアムが開発した自由に利用可能な Unix コードなどのソースから生まれているが、実用的な Linux システムの重要な部分は GNU プロジェクトから直接来ている。

Linux が成熟するにつれ、Linux システムを新規ユーザーにとってインストールしやすく、使いやすいものにするのを重視して、一部の人は**ディストリビューション**と呼ばれるパッケージを作成した。これは、Linux カーネルとユーティリティのほぼ完全なセットを組み合わせ、全体として完全なオペレーティングシステムを構成するものである。マンチェスタコンピュータセンター(英国マンチェスタ大学内)は、MCC ディストリビューションによって初期の指導的存在の 1 つだった。SoftLanding Systems (SLS) ディストリビューションがこれに続き、ここから Slackware が生まれた。Slackware は、Unix に関して十分な知識のないユーザーが、容易にインストールして保守できる最初の Linux ディストリビューションである。これは非常に人気が高かったので、Linux のインストール、使用、および保守だけを取り上げたコンピュータの本がまたにあふれ始めた。Red Hat Linux は、Linux ディストリビューションの世界では後発であるが、Linux の安定と安全、およびインストールと使用を容易にすることに集中しているため、今では最も人気の高いディストリビューションの 1 つである。

Linux ディストリビューションには、Linux カーネルに加えて、開発ライブラリ、コンパイラ、インタープリタ、シェル、アプリケーション、ユーティリティ、およびコンフィギュレーションツール、その他多くのコンポーネントが入っている。Linux システムがビルドされると、ディストリビューション開発者がさまざまな場所からツールを集めて、実用的な Linux システムに必要なすべてのソフトウェアコンポーネントの完全なコレクションを作成する。ディストリビューションの多くには、Linux システムのインストールと保守を容易にするカスタムコンポーネントも入っている。

多くの Linux ディストリビューションが入手可能で、それぞれに長所と短所がある。しかしながら、それらはすべて共通のカーネルと開発ライブラリを共有しており、それが Linux を他のオペレーティングシステムと異なるものになっている。本書は、開発者があらゆる Linux のためのプログラムを作成する手助けをすることを目的としている。すべての Linux ディストリビューションは同じコードを使用してシステムサービスを提供するので、プログラムバイナリやソースコードは、ディストリビューション間で非常に互換性が高い。

この互換性に貢献してきた 1 つのプロジェクトは、Linux Filesystem Standard(FSSTND)である。これは、多くのファイルがどこに保持されるべきかを指定し、また残りのファイルシステムを一般にどのように組織化すべきかを説明する。このドキュメントの新しいバージョンである Filesystem Hierarchy Standard(FHS)は、現在草稿を作成中である。両方の標準についての情報は、<http://www.pathname.com/fhs> で参照できる。

1.3 Unix システムの基本的な系譜

Linux の主要な部分は、従来の Unix ソースベースから独立して開発されたコードからなるが、Linux が提供するインターフェイスは、既存の Unix システムから大きな影響を受けた。

1980 年代初頭、Unix の開発は、カリフォルニア大学バークレー校と AT&T の Bell 研究所の 2 つの陣営に分かれた。それぞれの機関は、Bell 研究所で行われた元々の Unix の実装から派生した Unix オペレーティングシステムを開発し、保守した。

バークレーバージョンの Unix は Berkeley Software Distribution(BSD)として知られるようになり、学会では人気があった。BSD システムは TCP/IP ネットワークを含む最初のシステムだが、それがこのシステムの成功の一因となった。また Sun Microsystems が、Sun の最初のオペレーティングシステムである SunOS を BSD を基礎とする要因にもなった。

Bell 研究所もまた Unix の高度化に取り組んだが、残念なことにバークレーグループとは少し方法が違っていた。Bell 研究所からのさまざまなリリースは、System という語の後にローマ数字を付けて表された。Bell 研究所からの Unix の最後のメジャーリリースは System V(Sys V)であり、System V Release 4(SVR4)は、今日の多くの商用 Unix オペレーティングシステムのためのコードベースを提供している。

Unix がこのように分かれて開発されたことにより、システムコール、システムライブラリ、および基本的な Unix システムの基本コマンドにおいて大きな差異が生じた。この差異の好例の 1 つは、それぞれのオペレーティングシステムがアプリケーションに提供したネットワークインターフェイスにある。

BSD システムは、**ソケット**というインターフェイスを使用してプログラムが互いにネットワーク越しにやり取りできるようにした。これとは対照的に、System V は**トランスポートレイヤインターフェイス**(TLI)を提供した。これはソケットとはまったく互換性がないが、多くの領域でより大きな柔軟性がある。このように開発されたため、Unix のバージョン間におけるプログラムの移植性が大幅に減少し、コストが上昇し、すべてのバージョンの Unix について、サードパーティ製品を入手する可能性が減少した。

異なる Unix システムの間における非互換性の例は、ユーザーがオペレーティングシステムのプロセス情報を照会する `ps` コマンドである。BSD システムでは、`ps aux` はマシン上で実行されているすべてのプロセスの完全な一覧を表示するが、System V ではこのコマンドは無効で、代わりに `ps -ef` を使用できる。もちろん、出力形式はコマンドライン引数と同様に互換性がない。

この時期の分割された開発のために(これは愛情を込めて Unix 戦争と呼ばれる)分岐してしまった Unix のすべての側面を標準化しようと試みて、Unix 業界は Unix が提供するインターフェイスを定義する一連の標準を後援した。プログラミングとシステムツールインターフェイスに対処する標準の部分は POSIX として知られ、**米国電気電子学会(IEEE)**によって発行される。

1.4 Linuxの系譜

「標準の最良の点は、選択肢が多いことである」*1。Linux の開発者は、Linux の開発に際して詳細に検討すべき 20 年の歴史があり、さらに重要なことに、参照すべき高品質の標準があった。Linux は、主として POSIX にしたがって設計され、POSIX が規定していない箇所では一般に System V の例に従い、何らかの BSD インターフェイスのエミュレーションを提供する互換ライブラリを持つ。注意すべき例外は、ネットワークインターフェイスが完全に BSD で、System V の TLI インターフェイスはエミュレーションさえ提供していないことと、ネットワークユーティリティを含むユーティリティの一部が、BSD モデルに従っていることである。

System V は STREAMS インターフェイスを提供するが、このインターフェイスはパフォーマンスが悪く、その効果は他の方法で達成できるので、Linux は STREAMS を実装しない。この不適切な機能が欠けていることが残りのプログラミング環境に影響している。System V の ptys は STREAMS を必要とするクローンデバイス上に構築されているので、Linux は System V の ptys の代わりに、BSD の ptys を使用する(唯一の違いは、それがどのように割り当てられるかであって、この 2 つは同じように使用される)。

プログラミングの観点から見た SVR4 と Linux の間の最大の違いは、Linux はそれほど多くの重複したプログラミングインターフェイスを持たないことである。もっぱら SVR4 システム用にコーディングするプログラマでさえ、TLI よりバークレーソケットを好む。Linux は、TLI のオーバーヘッドを避け、ソケットの使用を強制する。汎用の STREAMS インターフェイスの代わりに、Linux はスタックブルライン方式とネットワークドライバ(カーネル内)を提供し、STREAMS のパフォーマンスの欠点を除去すると同時に、その最もよく使われ、最も有用な機能を維持している。

*1 Andrew Tanenbaum, *Computer Networks*, Prentice Hall, 1998, p.168

第2章

ライセンスと著作権

フリーソフトウェアの世界に新しく来た人は、フリーソフトウェアに規定されているライセンス条件の多様さに混乱することが多い。フリーソフトウェアの愛好家の中には、通常用語を専門用語に曲解し、専門用語のひとつひとつに与えられたすべてのニュアンスを理解しよう求める者もいる。

Linuxのようなフリープラットフォームで稼働するソフトウェアを作成して配布するには、ライセンスと著作権を理解する必要がある。これらの事項は、通常、知的で情報を多く持った人々、とりわけフリーソフトウェア愛好家によって混乱させられる。作成しようとしているのがフリーソフトウェアであっても商業ソフトウェアであっても、さまざまなライセンス条項が付いたツールを使って作業することになる。著作権とライセンスの領域の一般的な理解は、よくある誤りを避けるのに役立つ。

訴訟の多い昨今では、**我々は弁護士ではない**ことを読者に警告しておくことが非常に重要である。この章は、我々が議論することの我々なりの理解を反映しているが、法的助言を与えるものではない。自分自身の、または誰かほかの人の知的財産について決定を行う前に、問題をさらに検討し、不必要と考える場合以外は弁護士に相談する必要がある。

2.1 著作権

まずは簡単なことから始めよう。**著作権**は一定のタイプの知的財産の所有の簡単な主張である。最新の国際的な著作権協定では、作成したものについて著作権を主張する必要さえない。明示的に所有権を放棄しない限り、他者は公正な使用という、厳格に定義された方法でしか著作権者の知的財産を使用することが許されない。しかし著作権者が、それ以外の行為に対して明示的に許可、つまり**ライセンス**を与えた場合はこの限りではない。したがって、読者が本を書いたりソフトウェアを作成したりした場合、それを所有するために「Copyright © ××年」を付ける必要はない。しかしながら、この一節を付けなければ、誰かが(あなたが著作権を有していないかのように主張または行動することにより)著作権を侵害したり、ライセンス条件を侵害した場合に、法廷での所有権の主張がはるかに困難になるだろう。ベルン著作権協定^{*1}という、国際的な著作権協定と施行を扱う国際的な取り決めによると、協定国は著作権を、

*1 <http://www.wipo.org/>

著作権の抽象を合理的に通知するために、最初の出版時から、著者または他の著作権所有者の権限で出版された作品のすべてのコピーに、丸に入った小文字の"c"と、著作権者の名前と、このような方法で出版された最初の年および場所を伴ったシンボルを表示した場合……

にのみ強制することが求められている。

(c)というシーケンスは、丸の中に小文字のcを書いた記号の代替として一般に使用されているが、法廷はこれを支持しているわけではない。著作権を主張するときには、常にCopyrightという語を使用し、それに(c)というシーケンスを追加する必要がある。

著作権は永久の権利ではない。すべての知的財産は最後にはパブリックドメインになる。つまり、公衆がその財産に対する著作権を所有し、誰もがその財産でどのようなことでもできるようになる。財産がパブリックドメインになってしまうと、ライセンス条件は拘束力を失う。ここに1つのゆがみがある。パブリックドメインの作品を元に派生的作品を作成した場合、その改変に対して著作権が発生する。したがって、多くの古い本は現在著作権が切れてパブリックドメインになっているが、編集者はたびたびあちらこちらに小さな変更を行い、元の原稿にある誤りを正している。そして、その変更を含む派生的作品の著作権所有をたびたび主張するのである。この著作権があるために、編集されたバージョンは合法的にコピーできないが、著作権が切れたパブリックドメインのオリジナルは自由にコピーできる。

著作権を主張できるものには制限があることに注意すること。*the*とだけ書いた本を出版して、*the*という語を使うたびに、皆からライセンス料を要求しようとすることはできない。しかしながら、*the*という語の十分に様式化された絵を描いた場合、同じ表現の芸術がこれまでに存在しないことを示せる程度に識別可能である限り、その特定の表現に対し著作権を有するだろう。この文の中で*the*という語を自由に使用することはできるが、ライセンスを著作権者から取得することなく、この絵の複製を販売することは許されない。

このような制限は、ソフトウェアにも適用される。第三者のソフトウェアを改変するライセンスを持っていて、どうでもいい1語だけを変更した場合、その変更に対して著作権を主張するのは合理的ではない。その変更に対する著作権の主張を法廷で防御することは不可能だろう。行った変更が、*the*という語がそうであるのと同じくらいパブリックドメインだからである。しかしながら、ソフトウェアにかなりの追加を行った場合、その変更に対して著作権を所有するだろう。ただし、たとえばオリジナルの著作権の所有者が、すべての変更についてその著作権が彼らに帰属するという制限付きでライセンスを供与しているような場合はこの限りではない。

2.2 ライセンス供与

著作権所有者は、ライセンス条件の決定において広範な自由を与えられている。一般的に制限する領域には、使用、コピー、配布、および変更がある。具体的な例として、GNU General Public License(GPL、一般的にはコピーレフトと呼ばれている。Appendix C 参照)は、明示的に使用を制限しない。制限しているのは「コピー、配布、および変更」だけである。

知っておいたほうがよいと思われるフリーソフトウェアの専門用語の例を挙げよう。フリーソフトウェアの世界では、パブリックドメインという語は、ほとんど例外なく所有権に関して使用される。ソフトウェア以外の分野では、これはよく使用にも適用される。GPLを「パブリックドメインの著作権」として語る雑誌記事は明らかに誤りである。なぜなら、GPLは、著作権の所有をパブリックドメインにするものではない。GPLは使用についてライセンス上の制約を明示的にまったく設けていないので、それを「パブリックドメインのライセンス」と呼ぶ記事は、ある意味では正しい。しかしながら、フリーソフトウェアの愛好家は、このパブリックドメインの使用にしりがみすることが多く、多くの者がそれを完全に不正確だと信じている。

特定のライセンスの制限は、特定の地方で合法でないことがある。多くの統治機構は、それが公正な使用と考えるものが、ライセンス契約の中で制限されることのないようにしている。たとえば欧州の多くの国々は、一定の目的のためのソフトウェアおよびハードウェアのリバースエンジニアリングを明示的に許容しているが、それにもかかわらずライセンス条項はこのような活動を制約している。このため、多くのライセンス契約にはGPLの以下のような可分性条項が入っている。

本条項の或る部分が何らかの特別な状況下で無効または適用不可能になった場合、本条項のその他の残りの部分が適用されるように意図されており、また、本条項は全体としてその他の状況に当てはまるように意図されています。

多くのライセンス契約書は、これよりも包括的でない言葉を使って同じことを言っている。

法律家の助けを借りずに自分でライセンス条項を書こうと試みる場合、多くは法律上の効力がない条項が入ったライセンスを書き、可分性条項を含むライセンスはほとんどない。自分のソフトウェアのためのライセンスを自分で書こうとしていて、人々がライセンス条項を遵守するかどうか重要な場合は、知的財産権専門の弁護士にライセンス条項を綿密に調べさせる必要がある。

2.3 フリーソフトウェアのライセンス

2.3.1 フリーソフトウェアと商用ソフトウェアの組み合わせ

いろいろなフリーソフトウェアのライセンスがあり、それぞれ商用での使用、変更および配布が可能かどうか違っている。多くの場合、既存のコードを自分のプロジェクトで再利用することが好ましい。それはある程度は避けられないことでもある。たとえば、読者が作成するほとんどすべてのプログラムはCライブラリとリンクするので、Cライブラリのライセンス条項や、さらには自分のプログラムにリンクする他のライブラリのライセンスに気を付ける必要がある。また、他のプログラムのソースコードの一部を自分のプログラムに取り込むこともたびたびある。

2.3.2 GNU General Public License

GPL は、制限の大きいフリーソフトウェアライセンスの1つである。GPL の条項の下でライセンス供与されたソースコードを別のプログラムに取り込む場合、そのプログラムもまた GPL の条項の下でライセンス供与する必要がある^{*2}。Free Software Foundation(FSF。GPL の作成者) は、ライブラリへのリンクを「派生的作品の作成」と考えている(その他の者の中には、これを「単に集合させる仕事」と考える者もいる)。したがって、FSF はリンクしたプログラムも GPL の条項によってカバーされている場合でなければ、GPL の条項でカバーされたライブラリとリンクできないという姿勢を崩さない^{*3}。しかしながら、一部の人は、リンクは「単なる集合」だとしており、GPL は以下のように述べている。

さらに、「プログラム」(又は「プログラム生成物」) と「プログラム生成物」とはならない他のプログラムとを、単に保管や頒布のために同一の媒体上にまとめて記録したとしても、本使用許諾は他のプログラムには適用されません。

実行ファイルを「保管の媒体」と考えると、リンクを単なる集合と考えることができるだろう。

我々が知る限りでは、この区別はまだ法廷で吟味されていない。非常にまれなケースであるが、GPL の条項でライセンス供与されていないプログラムを GPL の条項でライセンス供与されているライブラリにリンクしたい場合は、その解釈について、当該ライブラリの作者に問い合わせること^{*4}。

2.3.3 GNU Library General Public License

GNU Library General Public License(LGPL)は、ライブラリを全般的により有用にすることを目的に設計された。LGPL のポイントは、ユーザーがそれらのライブラリに対してリンクされる新しいバージョンのプログラムを入手しなくても、自分のライブラリをアップグレードまたは拡張できる点である。そのため LGPL は、プログラムが LGPL の下でライセンス供与されている共有バージョンのライブラリに対してリンクされているか、またはアプリケーション用のオブジェクトファイルとともに提供される限り、いかなるライセンス制限も課そうとはせず、ユーザーが新しい、または変更されたバージョンのライブラリに再リンクできるようになっている。

実際には、この制限は大きなものではない。共有ライブラリが利用可能な場合に、それに対してリンクしないのは不合理だからである。

GPL の条項の下でライセンス供与されているライブラリはほとんどなく、多くは LGPL の条項の下でライセンス供与されている。GPL の条項の下でライセンス供与されているライブラリは、

*2 このため、一部の人は GPL をウィルスと呼んでいる。

*3 慌てないで! 次の 2.3.3 を参照。

*4 GPL の条項の下でライセンス供与されている唯一の共通ライブラリは、GNU Database Management ライブラリ(gdbm)である。db は gdbm より制限的でないライセンス条項を持ち、機能性が高いので、本書では、代わりに Berkeley Database library(db)について説明する。

通常は単に作成者が LGPL について知らないか、それを検討しなかつただけである。ていねいな要求に応じて、多くは LGPL の条項の下でライセンスし直している。

2.3.4 MIT/Xスタイルライセンス

MIT/X スタイルライセンスは、GPL や LGPL よりはるかに簡単である。制約は、既存の著作権の表示とライセンス条項をすべて変更することなく配布するソースとバイナリに保持しておくこと、および事前に書面によって許可を得ることなく、派生的作品を保証または販売促進するために、どの作成者の名前も使用しないことだけである。

2.3.5 BSDスタイルライセンス

BSD スタイルライセンスは、本質的には MIT/X スタイルライセンスの条件に加えて、機能またはソフトウェアの使用に言及する広告宣伝資料に謝辞を入れることが規定されている。

2.3.6 アーティスティックライセンス

Perl 言語のソースコードは、GPL の条項または代替のライセンスの条項に従うことを可能にするライセンスとともに配布されているが、このライセンスは気まぐれに**アーティスティックライセンス**と呼ばれる。アーティスティックライセンスの主な目標は、再配布の権利を保護することと、ユーザーがオフィシャルバージョンだと見せかけて変更された独自の改変版を販売するのを防止することである。他のソフトウェア作成者は、ユーザーが GPL またはアーティスティックライセンスのいずれかの条項に従えるという Perl の例を採用しているが、アーティスティックライセンスの条項だけの下でライセンス供与する作成者は少数である。

2.3.7 ライセンスの非互換性

ライセンスの異なるソフトウェアからコードを混合することは、ときに問題となることがある。共有ライブラリとリンクする場合は問題にならないが、派生的作品を作成する際には必ずあてはまる。他人のソフトウェアを変更している場合、そのライセンス条項を理解する必要がある。異なるライセンスを持つ2つのソフトウェアを1つの派生的作品に組み合わせようとするときは、そのライセンス上の矛盾を判定する必要がある。これもまた、最初から自分のコードを作成するときには当てはまらない。

GPL または LGPL の条項の下でライセンス供与されたコードで作業を行っている場合には、BSD スタイルの下でライセンス供与されているコードに含めることができない。GPL と LGPL が「追加的制約」を禁止しているのに、BSD ライセンスには広告と裏書きについての追加的制約（つまり、GPL または LGPL 以上のもの）が入っているためだ。この矛盾のため、多くのソフトウェアは2種類の条項の下でライセンス供与されている。つまり、GPL と BSD スタイルライセンスの条項が両方提供されているので、どちらのライセンス条項に従うかを選択できる。

GPL または LGPL でライセンス供与されたコードが MIT/X スタイルライセンスから派生した

作品に取り込まれる場合、(あらゆる実務的な目的で)派生する作品全体がそれぞれ GPL または LGPL の条項の下でライセンス供与される必要がある。

潜在的な非互換性は、これ以外にも多くある。特定のフリーソフトウェアで何を許されているのかが疑わしいときは、恥ずかしがらずに著作権所有者に尋ねることである。彼らはどのような方法でもソフトウェアの使用ライセンスを与えられるということに注意すること。

第3章

Linuxに関する情報源

読者もやがて、本書が説明していないことを知りたいと思うようになるだろう。Linux についての情報検索は、どこをどのように見るかがわからないということがない限り、困難ではない。

Linux 情報を探す場合、Linux はインターネット上で設計され作成されたため、Linux のドキュメントの多くはインターネット用に設計されていることを理解する必要がある。ローカルに情報源がある場合でも、その多くは元々オンラインの、インターネットで利用可能な情報源として設計されている。これに留意すれば、たとえ印刷されたドキュメントに限ったとしても、よりスムーズに検索できるようになる。

Addison-Wesley の本書の Web サイト、<http://www.awl.com/cseng/books/lad/>には、本書の本文の更新版、本書の射程を越えた詳細情報、およびインターネット上の詳細情報へのポインタがある。この章に示す URL は時代遅れになっていることがあるので、古くなった URL を見つけた場合は、前述の Web サイトに掲載することにする。

3.1 Linuxのドキュメントの概要

Linux の世界で整理された最大の情報源は、Linux Documentation Project(LDP)である。LDP は、主に標準のオンラインマニュアルページ(man ページ)、多様なトピックについての LDP の書籍、および数百のトピックをカバーするハウズドキュメントに責任があり、それらの程度や詳細度はさまざまである。特にハウズドキュメントは、印刷、ハイパーテキスト、およびプレーンテキスト形式で利用可能だ。

Linux Usenet **ニュースグループ**階層(`comp.os.linux.*`)もまた、一般に Usenet からの情報を収集することに慣れた人々には便利な情報源となる。Linux の進歩についていこうとするなら、少なくとも週に 1 回は `comp.os.linux.announce` をのぞく必要がある。

Linux システムで実行する可能性の高いソフトウェアの大部分は、GNU プロジェクト、BSD プロジェクト、または X プロジェクトから来ているので、ドキュメントも同じように分けられる。GNU プロジェクトのメインのドキュメント形式は Texinfo で、同じソースから活字とハイパーテキスト、また多くの man ページを提供するが、Texinfo ドキュメントのほうが新しい。BSD プロジェクトのドキュメントは、多くが man ページである。公式の X ディストリビューションには、man ページと印刷できる書籍が入っているが、オンラインで表示するのは簡単ではない。多くの X プログラマやユーザーは 2 次ソースのドキュメントを購入しており、その中には O'Reilly から出版されている有名な一連のプログラミングリファレンスが含まれる。

Linux Documentation Project が管理している Web サイトは、Linux の世界のすべてがそうである程度に標準に近い。プロジェクトのホームページは、<http://sunsite.unc.edu/LDP/>で、世界中の数百という他のサイトにミラー処理され、ほとんどすべての人々に素早く、比較的ローカルなアクセスを提供している。

Linux Documentation Project によって発行される印刷可能なドキュメントの多くは、多様なベンダーからさまざまなタイトルで不定期に作成されている。1997 年中盤、これらの本は、約 2,000 ページの長さになろうとしている。Linux Software Labs^{*1}、Red Hat Software^{*2}、および Yggdrasil^{*3}は、現在、このような本の 3 つの最大のベンダーである。

また、ベンダーの 1 つである Red Hat Software は Linux Library という CD-ROM 製品を販売しており、これには LDP Web サイト全体のスナップショットを含む、インデックスの付いたブラウザ可能なアーカイブが入っている。

3.1.1 ハウツーとミニハウツー

Usenet から発信されるドキュメントは、よく FAQ(よく聞かれる疑問)の形態を取る。このようなドキュメントは、そのテーマの多くに適切な Q&A の形式を取る。しかし、FAQ は多くの種類のドキュメントに適しているわけではないので、LDP は解説的な、または物語風のドキュメントのための標準形式を確立した。このようなドキュメントの多くは、何かを行う方法、つまりハウツーを説明することから、ハウツードキュメントと呼ばれる。それらは活字、ハイパーテキスト、およびプレーンテキスト形式で利用可能だ。他の形式に変換するように設計された SGML で作成されている。

ミニハウツーと呼ばれる非常に簡単なドキュメントはわずか数ページの長さで、一般に 1 つのトピックについて説明する。

3.1.2 LDPの書籍

LDP の当初のプロジェクトの 1 つは、Linux についての書籍をすべて製作することだった。このような本の多くは、ユーザーとシステム管理者向けに書かれている。あるものはアプリケーションプログラミングのガイドであり、またあるものはカーネルプログラミングのガイドである。これらはすべて、LDP ホームページからなんらかの形式でアクセス可能である。

3.1.3 Linux Software Map

必要なプログラムが見つからなくて困っている場合は、<http://www.execpc.com/~lsm/>にある Linux Software Map(LSM)が、Linux で利用可能なソフトウェアパッケージの数千の記録を持っている。これらの記録には、名前、説明、バージョン番号、およびパッケージの位置が入

*1 <http://www.lsl.com/>

*2 <http://www.redhat.com/>

*3 <http://www.yggdrasil.com/>

ている。

3.1.4 man ページ

man コマンドを使って、システムリファレンスマニュアルページ(man ページ)にアクセスしよう。man コマンド自体のリファレンスページを表示するには、man man というコマンドラインを実行する。man ページに入っているのは、一般にリファレンスマニュアルであってチュートリアル情報ではなく、あまりに簡潔なのでほとんど理解できない場合があることでよく知られている。それでもなお、リファレンス資料が必要なときは、これがまさに求めるものであることもある。

man ページには、3つのプログラムでアクセスできる。man プログラムは個々の man ページを表示し、apropos と whatis コマンドは man ページのセットからキーワードを検索する。apropos と whatis コマンドは同じデータベースを検索する。違いは、whatis が検索している語と完全に一致する行だけを表示するのに対し、apropos は検索している語を含むすべての行を表示する点である。つまり、man を検索した場合、apropos は manager や manipulation にもマッチするが、whatis は man.config のように、空白または句読点によって他の文字と区切られている man だけにマッチする。whatis man と apropos man のコマンドを試して、その違いを確認すること。

Linux システムの man ページは、LDP によって組み立てられた大きなパッケージの一部である。特にセクション 2(システムコール)、セクション 3(ライブラリ)、セクション 4(特殊事項またはデバイス、ファイル)、およびセクション 5(ファイル形式)のページは、主に LDP の man ページコレクションから来ており、一般にプログラミングに最も便利なページである。どのページを見るべきかを指定したい場合、表示する man ページの名前の前にそのセクションの番号を指定する。たとえば、man man は man コマンドについてのセクション 1 の man ページを表示する。man ページの作成方法についての仕様を表示する場合、man 7 man としてセクション 7 を指定する必要がある。

man ページを見るときには、多くのシステムコールやライブラリ関数が同じ名前を使用していることに留意する必要がある。多くの場合、知りたいのはリンクしようとしているライブラリ関数についてであって、それらのライブラリ関数が最後に呼び出すシステムコールについてではない。ライブラリ関数の一部は、セクション 2 のシステムコールと同じ名前を持つので、忘れずに man 3 *function* を使って、ライブラリ関数の説明を表示する必要がある。

また、C 言語の man ページが C ライブラリ自体とは別に管理されていることに特に注意する必要がある。C ライブラリは、通常、動作を変更しないので、これは普通は問題にならない。Linux のプラットフォームはすべて、(前述したように)統合した C ライブラリである GNU C ライブラリに移行する過程にある。GNU C ライブラリには完全なドキュメントが用意され、ライブラリによって維持される。この情報は、Texinfo 形式で利用することができる。Texinfo ドキュメントの検索と表示については「3.6 その他のドキュメント」を参照してほしい。GNU C ライブラリがあらゆる Linux プラットフォームで使用できる場合でも、man ページはやはり別に管理され、さらに Texinfo 形式のドキュメントが増えるだけだ。

3.2 その他の書籍

LDP だけが Linux についての書籍の出版元ではない。*Practical Guide to Linux* [Sobell, 1997] は、Linux の使用方法、シェルプログラミング、およびシステム管理の紹介が入った 1,000 ページの大きな本である。これには、Linux システムに含まれている多くのユーティリティの簡単なリファレンスも用意されている。*Linux in a Nutshell* [Heckman, 1997] は、それよりも小さくて短く、O'Reilly がそれ以前に発表した要約的リファレンスから派生したユーティリティに集中している。

Beginning Linux Programming [Matthew, 1996] は Linux プログラミングの親切で詳細な入門書で、シェルプログラミングのような重要なトピックが入っている。*Linux Multimedia Guide* [Tranter, 1996] には、Linux のために利用できるマルチメディアソリューションについての情報および Linux サウンドインターフェイスのプログラミングに関する情報が入っている。*Linux Device Drivers* [Rubini, 1998] は、オペレーティングシステムのコードに触ったことのない人と扱ったことのある人の双方を対象に、Linux デバイスドライバの作成方法を説明する。

3.3 ソースコード

フリーソフトウェアの利点を活用しよう。ソースコードを読むのである。Linux のディストリビューションを購入した場合、ベンダーは GPL の条項によってソースコードを提供する法的義務がある。ソースコードを見て、自分がしようとしていることに近いことをしているプログラムを探そう。ディストリビューションには、ソースコードの検索方法の説明があるはずだ。

3.4 Linux(およびその他の)ニュースグループ

Linux ニュースグループは、他の Usenet ニュースグループと同様、圧倒的な量の偽情報や誤情報の中に隠れて多くの情報がある。以下では、プログラマに有用だと思うニュースグループについて説明する。

さらに、我々は Linux が Unix ライクなオペレーティングシステムであると認識しているので、Linux に固有ではないが、読む価値のある Unix のニュースグループをいくつか勧めることにする。

議長が取り仕切るニュースグループである comp.os.linux.announce は、*cola* という愛称が付いており、少なくとも他の Linux のニュースグループと比べてトラフィックが少ない。おそらく 1 週間に平均 50 のポストがある。ポストの多くは、その内容を区別するタグが付いている。内容が地域的な利益だけに限られるものは LOCAL と指定され、広告や商業上の利益のために作成されたものは COMMERCIAL と指定されるなどということになる。

このニュースグループのトラフィックはすべて、以下のように複数の場所でアーカイブされている。

<http://www.cs.helsinki.fi/~wirzeniu/linux/cola.html>

<http://sunsite.unc.edu/pub/Linux/docs/linux-announce.archive/>

Red Hat Linux Library

議長が取り仕切るニュースグループである `comp.os.linux.answers` には、Linux についての FAQ、HOWTO、README などが入っている。

議長のいないニュースグループである `comp.os.linux.development.app` は、Linux アプリケーションの作成または古いアプリケーションを Linux にポータリングする人々を対象にしている。

議長のいないニュースグループである `comp.os.linux.development.system` は、Linux カーネル、デバイスドライバ、モジュール、および特に Linux 固有のプログラムに取り組んでいる人々を対象としている。

議長のいないニュースグループである `comp.os.linux.hardware` は、Linux におけるハードウェア関連の問題を議論する人々を対象としている。

議長のいないニュースグループである `comp.unix.programmer` は、一般的な Unix プログラミングを議論する人々を対象としている。その多くは Linux システムに関連している。

議長のいないニュースグループである `comp.lang.c` は、使用するオペレーティングシステムに関係なく C プログラミング言語を議論する人々を対象としている。

3.5 メーリングリスト

メーリングリストは、それを介して電子メールが複数の他のアドレスに再配信される電子メールアドレスである。一般に、メーリングリストに参加している人は誰でもメッセージを送信ことができ、それが他の全員に配信される。議長が取り仕切るメーリングリストでは、議長のいるニュースグループと同様、管理者がメッセージを再配信する前に個々のメッセージを承認しなければならない。

メーリングリストには、一般にリスト管理ソフトウェアが必要である。最も一般的なリスト管理ソフトウェアパッケージは、`smartlist` と `Majordomo` の 2 つである。どちらの場合も、自動リスト管理ソフトウェアに購読開始または購読停止を要求するメールを送信する。`smartlist` と `Majordomo` では、アドレスと要求形式が異なっている。`Majordomo` リストを購読するには、`majordomo@host.domain` あてに、`subscribe listname you@yourdomain.com` という本文 件名

ではなく)の入ったメッセージを送信する。smartlist リストを購読するには、`listname-request@host.domain` へてに、件名(または本文の最初の行)に `subscribe` と書いたメッセージを送信する。メーリングリストについて話すときは、一般に、どのソフトウェアがそれを管理するか、またはどのように購読するかを伝える。

通常、リストを購読すると、ポストの方法と購読停止の方法を伝えるメッセージが自動的に送信される。それをわかるところに保存しておくこと!

リストの購読開始または購読停止について助けを必要とする場合、リストの所有者に連絡できる。多くのリストには、リストの管理者が読む `listname-owner` アドレスがある。アドレスが `owner-listname` の場合もある。

<http://www.linuxhq.com/lxlists/> というサイトは、Linux 固有の内容を持つ多くのメーリングリストを保管している。このサイトにはまた、<http://www.linuxhq.com/lxlists/subscribe.html> に購読方法のページがあり、サイトが保管しているリストを購読する方法を説明している。

これらのリストに参加する方法についての情報は、本書を書いている時点でのことである。リスト管理者がリストを管理するソフトウェアをときどき変更したり、リストがときどき移動されることに留意する必要がある。これらのリストについての最新の情報を、<http://www.awl.com/cseng/books/lad/lists.html> に置いておく。

3.5.1 vger

Linux に関連するものすべてが「公式」と呼べる範囲において、`vger.rutgers.edu` がホストするメーリングリストは、公式の Linux メーリングリストである。特に主要なカーネル開発者の多くは、`linux-kernel@vger.rutgers.edu` のリストを読んでいる。

vger リストは Majordomo によって実行されている。Majordomo コマンドについてのヘルプは、本文に `help` とただ 1 語だけ書いたメッセージを `majordomo@vger.rutgers.edu` に送信することによって入手できる。`lists` という語を使うと、vger から現在利用可能なメーリングリストのリストを要求できる。

多くのリストは、Linux の特定の部分の開発(カーネルやネットワーク、または DOS エミュレータ)または管理のなんらかの側面に関連している。

3.5.2 その他のリスト

世界中の人々は、メーリングリストをさまざまな目的で維持している。多くのリストがどこかの Web ページで言及され、多くが Web 上でアーカイブされているので、知っている Linux の Web サイトからのリンクをたどったり、Alta Vista のような Web 検索サイトで検索したりすることは、興味のあるトピックについてのメーリングリストを見つけるのに便利な方法である。一部のリストは、`comp.os.linux.announce` ニューズグループでアナウンスされている。cola アーカイブを検索することが実を結ぶこともある。`linuxhq` にアーカイブされているリストを見るのが有益なこともある。

Linux ベンダーは、メーリングリストをホストすることにより Linux 開発に貢献している。彼らがこのサービスを恥ずかしがって隠すことはほとんどなく、通常そのベンダーの Web サイトを見れば、彼らがどのリストをホストしているかがわかる。

3.6 その他のドキュメント

確かに Linux 固有や Linux 指向のドキュメントはたくさんあるが、特に Linux だけについて書かれたのではないドキュメントにも有用なものが多く存在する。

3.6.1 GNU

Free Software Foundation の GNU プロジェクトは、常にドキュメントに力を入れており、LDP 同様そのドキュメントの大部分に使用されている公式のフォーマットもある。Texinfo は、元々活字出力と info と呼ばれる単純なハイパーテキスト形式の 2 つの形式の出力を作成するように設計されていた。今では、HTML の作成にも使用できる。

info ファイルは、info リーダーで読むことができる。GNU プロジェクトには、Emacs エディタに内蔵された info リーダーに似たインターフェイスを持つ info というスタンドアロンの info リーダーが用意されている。jed などの他のいくつかのエディタにも info リーダーが用意されている。info の使用方法を学ぶために、`info info` というコマンドを実行できる。その代わりに、<http://www.awl.com/cseng/books/lad/info/>にある本書の Web サイトで、GNU のドキュメントのセットを HTML 形式で表示することもできる。

3.6.2 BSD

Linux システムにある多くの BSD 派生ドキュメントは、元々 BSD 用に作成されたプログラムがインストールした man ページの形式である。BSD 4.4lite のソースツリーには man ページ形式でない他の補助ドキュメントがあり、man ページと補助ドキュメントは、O'Reilly から印刷されている [CSRG, 1994a] [CSRG, 1994 b] [CSRG, 1994 c] [CSRG, 1994 d] [CSRG, 1994 e]。

3.7 ディストリビューションベンダー

多くの Linux ディストリビューションベンダーは、メーリングリスト、Web ページ、および ftp サイトのように、ユーザーのための情報リソースを持っている。時間をかけてベンダーが提供するものを知ってほしい。少なくとも、バグフィックスがどのように通知されるかは調べておこう。

第4章

開発ツール

Linux については驚くほど多くの開発ツールが存在する。そのうち重要ないくつかについては精通しておくことが必要だろう。

Linux プログラムには、多くの定評ある開発ツールが同梱されている。そのうち多くは、Unix 開発ツールとして長年提供されてきた。これらのツールの見た目は地味で、GUI がないコマンドツールである。しかし長年の使用による実績があり、これらについて学ぶことは大きな価値がある。

すでに Emacs、vi、make、gdb に精通している読者にとっては、以下の説明に何も目新しいことはないだろう。本書の後の章は、読者がテキストエディタになじんでいることを前提にしている。また、フリー Unix や Linux のソースコードのほぼすべてが make で作成されており、gdb は Linux および Unix で最も広く使用されているデバッガである。

4.1 エディタ

Linux でも(いくつかの異なったプログラミング言語の)統合開発環境(IDE)がいくつか提供されているが、他のプラットフォームのIDEほどには普及していない。Unix 開発者は、特にエディタについて各人がさまざまな好みを持ち、かつその好みにこだわるというのが従来からの特徴である。

プログラマが使用しているエディタの多くは、誰でも使ってみることができる。そのうち最も広く使用されている2つのエディタが vi と Emacs だ。どちらも一見した印象より強力なエディタで、どちらも比較的短時間で操作を習得できるが、しかしこの2つのエディタは大きく異なっている。Emacs は大規模なエディタで^{*1}、それ自身が操作環境とでもいうべきものだ。vi は小規模で、Unix 環境の一部として設計されている。この2つのエディタには多数のクローンや別バージョンがあり、それぞれにファンがいる。

vi と Emacs のチュートリアルは、本書に含めるには分量が多すぎる。これらのエディタについては *A Practical Guide to Linux* [Sobell, 199] という本の中の1章にすぐれた紹介がある。また O'Reilly がそれぞれのエディタについて1冊ずつ本を書いている。 *Learning GNU Emacs* [Cameron, 1996] と *Learning the vi Editor* [Lamb, 1990] である。以下に Emacs と vi の比較と、それぞれのオンラインヘルプの参照の仕方を紹介する。

*1 Emacs とは "Editor MACroS" の略ではなく "Eighteen(あるいは Eight または Eighty) Megs And Constantly Swapping" の略であるという人もいる。

Emacs には詳細なマニュアルが付属しており、Emacs をエディタとして使用方法だけでなく、Emacs による電子メールやニュースグループの読み書きの方法、ゲームの遊び方(五目並べはなかなかおもしろい)、シェルコマンドの実行方法なども説明されている。Emacs では常にコマンドを実行できる。キーに割り当てられていないコマンドはタイピングで入力する。

これに対して、vi のドキュメントはやや不親切で普及度も低い。vi の機能はエディタのみで、コマンドの多くが1個のキーに割り当てられている。入力した文字がそのまま文書中に書き込まれるモードと、文字がコマンドになるモードとを切り替える必要がある。たとえば、コマンドモードでは h、j、k、l の各キーは文書をスクロールする矢印キーとして動作する。

どちらのエディタでも作業の能率を高めるためにマクロを作成できるが、それぞれのマクロ言語は大きく異なっている。Emacs のプログラミング言語である elisp(Emacs Lisp)は、Common Lisp プログラミング言語と近い関係にある。これに対し vi のプログラミング言語は、より硬派なスタックベースの言語である。ユーザーの多くは単にキーをシンプルな1行 vi コマンドに割り当てているだけだが、コマンドの中には vi の外でプログラムを実行して vi 内のデータを操作するものも多い。Emacs Lisp にはチュートリアルを含む膨大なマニュアルがあるが、vi の言語についてのマニュアルは少ない。

エディタの中には、Emacs と vi の機能を混合して使用できるものがある。Emacs を vi モードで(viper と呼ばれる)使用すると標準 vi コマンドを実行することができる。vi クローンの1つは vile(Vi like Emacs)と呼ばれている。

4.1.1 Emacs

Emacs にはいくつかのバージョンがある。オリジナルの Emacs エディタは、有名な Free Software Foundation の Richard Stallman が作成した。この GNU Emacs は長年にわたり最も人気のあるバージョンだった。最近では、GNU Emacs にグラフィカルな要素を加えた XEmacs も広く使用されている。XEmacs の前身は Lucid Emacs だが、これは現在では消滅した Lucid Technologies が GNU Emacs の機能強化を行ったもので、将来は GNU Emacs に正式に組み込もうとしていた。しかし技術的な差異により、このコードを組み込むことはできなかった。しかし、これらの2つのエディタには高い互換性があり、プログラマは互いにコードを流用することが多い。この2つのバージョンはきわめて類似しているため、ここでは両方とも Emacs と呼ぶことにする。

Emacs エディタに慣れる最善の方法は、チュートリアルを実行することである。emacs を実行して ^h t(コントロール - ヘルプ、チュートリアルの意味)と入力する。Emacs を終了するには ^x^c と入力する。チュートリアルが Emacs についての学習方法を教えてくれる。チュートリアルは、Emacs に添付されている Emacs マニュアルの見方を教えるものではない。マニュアルの見方を知りたい場合は ^h i(コントロール - ヘルプ、インフォメーション)とする。

Emacs のユーザーインターフェイスは、他のシステムの IDE ほどの派手さはないが、プログラマが望む強力な機能はほとんど備えている。たとえば Emacs で C コードを編集する場合、Emacs がファイルタイプを認識して「C モード」に入り、C の構文を認識して入力ミスを教えてくれる。

Emacs 内でコンパイラを実行すると、エラーと警告メッセージを認識し、1 コマンドでエラーがある行まで移動する(必要なら新しいファイルを読み込む)。またデバッグモードでは、デバッガを1つのウィンドウに表示しておき、デバッグするコードを別のウィンドウに表示できる。

4.1.2 vi

タッチタイプができて常に指をホームポジションに置いておきたい場合には^{*2}、vi が向いている。vi ではコマンドを最小の指の動きで実行できるように設計されているからだ。また Unix ユーザー向けに設計されているため、標準**正規表現**を使用する sed や awk などの Unix プログラムに慣れているユーザーは、`^`で行頭に移動し`$`で行末に移動するなどの操作をごく自然に感じるだろう。

残念なことに、vi は Emacs に比較して学習が難しい。vi には、標準 Emacs チュートリアルに似たチュートリアルはあるが、どのバージョンの vi からでもチュートリアルを起動できる標準的な方法はない。ただし、ほとんどのバージョンは、一般的な Linux に同梱されているバージョンを含め、`:help` コマンドが使用できる。

4.2 make

Unix 開発ツールの主力は make である。make はプログラムのコンパイル方法を簡単に記述できる。小規模なプログラムでは、1つのコマンドだけで1つのソースコードファイルを1つの実行ファイルにコンパイルできる場合もある。しかしそれでも、`gcc -O2 -ggdb -DSOME_DEFINE -o foo foo.c` と入力するよりは make と入力するほうが簡単だ。また、コンパイルするファイルが多数あるのに変更したソースファイルが少ししかない場合、make は関連するソースファイルが変更された場合のみ新しいオブジェクトファイルを作成する。

make にこの魔法を実行させるには、Makefile にすべての関連ファイルを記述しておく必要がある。以下に例を示す。

```
1: # Makefile
2:
3: OBJS = foo.o bar.o baz.o
4: LDLIBS = -L/usr/local/lib/ -lbar
5:
6: foo: $(OBJS)
7:     gcc -o foo $(OBJS) $(LDLIBS)
8:
9: install: foo
10:     install -m 644 foo /usr/bin
11: .PHONY: install
```

*2 Qwerty 配列のタッチタイプができる場合。Dvorak 配列のユーザーが vi を使用する場合は、マクロで vi を使いやすく設定するのが一般的だ。

1行目はコメント行である。make は通常の Unix の伝統に従い、#でコメントを示す。

3行目は、OBJS という変数を foo.o bar.o baz.o に設定している。

4行目は、別の変数 LDLIBS を定義している。

6行目は**ルール**の定義の始まりで、この場合ファイル foo は変数 OBJS に名前が含まれているファイルに依存する(この場合はこれらのファイルから作成される)ことを表している。foo は**ターゲット**と呼ばれ、\$(OBJS) は**依存リスト**と呼ばれる。変数展開の構文に注意。変数名を\$(...) で囲むのである。

7行目はコマンドラインで、依存リストからターゲットを構築する方法を示す。コマンドラインは複数にわたることもある。コマンドラインの最初の文字は必ずタブでなければならない。

9行目はちょっと変わったターゲットである。これは実際には install というファイルを作成するものではなく、(第10行でわかるように)標準の install プログラムを使用して/usr/bin に foo をインストールする。しかしこの行は make の解釈上あいまいである。もし install という名のファイルが存在し、それが foo より新しいファイルであったときにはどうなるのだろうか? その場合は make install コマンドを実行すると make が"install' is up to date"('install' は更新済み)と表示して実行を終了する。

11行目は、install がファイルではなく、install の依存ファイルを計算する場合に"install" という名のファイルは無視されることを示している。したがって、install の依存が呼び出された場合には(その方法は後述)第10行のコマンドが常に呼び出される。.PHONY は make の動作を変更する命令である。この例では、.PHONY は install ターゲットがファイル名でないことを make に通知する。.PHONY ターゲットは、インストールや作成すべき他の複数のターゲットに依存する1つのターゲット名の作成に使用されることが多い。たとえば以下のとおりだ。

```
all: foo bar baz
.PHONY: all
```

ただし、make のバージョンの中には .PHONY をサポートしていないものもある。以下は、これより手間はかかるが同じ効果があり、バージョン間の共通性も高い方法である。

```
all: foo bar baz FORCE
FORCE:
```

この方法がうまくいくのは"FORCE"という名前のファイルが存在しない場合のみだ。

依存リスト内のアイテムはファイル名であることが多いが、makeにとって、これらのアイテムは別のターゲットである。たとえば、install 依存リスト内の foo アイテムはターゲットである。install の依存を解決しようとするれば、先に foo の依存を解決する必要がある。foo の依存を解決するには、foo.o、bar.o、baz.o の依存を解決しなければならない。

foo.o、bar.o、または baz.o を作成する方法を make に明確に指示した行がないことに注意してほしい。これらのファイルをエディタで作成することはない。make が暗黙の依存を提供するため、ユーザーが作成する必要はないのである。 .o で終わるファイルに依存があり、かつ .c で終わることを除いて同じ名前のファイルがある場合には、make はオブジェクトファイルがソースファイルに依存しているものと判断する。make の組み込みサフィックスルールにより Makefile は大幅にシンプルにすることができ、また自分に適した組み込みルールがない場合は自分でサフィックスルールを作成できる(「4.2.3 サフィックスルール」参照)。

デフォルトでは、make はコマンドの実行に失敗した場合には直ちに実行を終了する(エラーを戻す)。これを回避したい場合、方法は2つある。

引数 -k をつけると、make はコマンド呼び出しによりエラーが返されても実行を終了せずに、できるだけ多くのファイルを作成する。これが有益なのは、たとえば移植を行う場合である。できるだけ多くのオブジェクトファイルを作成し、そして中間ファイルの作成を待たずに、作成に失敗したファイルの移植を始められる。

あるコマンドが常にエラーを返すことがわかっており、かつエラー状態を無視したい場合には、シェルの魔法を使用できる。/bin/false コマンドは常にエラーを返す。したがって、以下のコマンド、

```
/bin/false
```

を実行すると、-k オプションが指定されていない限り実行を終了する。しかし、以下のコマンド、

```
any_command || /bin/true
```

では、make はコマンドの実行を終了しない。any_command が偽を返した場合、シェルは/bin/true を実行してその終了コードを返すが、これは必ず成功する。

make は、認識されないコマンドライン引数でダッシュ(-)^{*3}で始まらないものを、作成するターゲットとして解釈する。したがって make install を実行すると、make は install ターゲットを作成しようとする。foo ターゲットが最新でない場合、make はまず foo を作成することで依存を解決し、次に foo をインストールする。ダッシュで始まるターゲットを作成する場合は、ターゲットの前に2つのダッシュ(--)引数を付ける必要がある。

*3 マイナスとハイフンを合わせてダッシュと呼ぶことが多い。

4.2.1 複雑なコマンドライン

各コマンドラインはそれぞれ独自のサブシェルで実行されるため、あるコマンドラインの `cd` コマンドはそのコマンドが記載されている行にのみ影響する。Makefile 内の複数行はバックスラッシュを付けることで結合できる。つまり、行の終わりにバックスラッシュを付けると、その次の行もスペースにより結合されているものと解釈される。コマンドラインは以下になることが多い。

```
1:      cd some_directory ; \
2:      do this to file $(FOO); \
3:      do that
4:      cd another_directory ; \
5:      if [ -f some_file ] ; then \
6:      do something else ; \
7:      done ; \
8:      for i in * ; do \
9:      echo $$i >> some_file ; \
10:     done
```

上の例は、`make` にとっては2行しかない。最初のコマンドラインは1行目から3行目までである。第2のコマンドラインは4行目から10行目までである。ここでは注意すべき点がある。

`another_directory` は `some_directory` に対する相対パスではなく、`make` が作動しているディレクトリに対する相対パスである。これらの2つは別個のサブシェルで実行されるからだ。

各コマンドラインを構成している行は1つの行としてシェルに渡される。したがって、シェルが必要とするすべての、文字をリストに示したとおりに書かなければならない(通常は、改行があれば、文字があることを意味しているので、シェルスクリプトでは省略される)。シェルプログラミングの詳細は *Learning the bash Shell* [Newham, 1995] を参照。

`make` 変数を間接参照する場合、通常はそのまま間接参照すればよいが(つまり `$(VAR)`)、シェル変数を間接参照する場合は、`$` を2回繰り返すことで `$` 文字をエスケープする必要がある(`$$i`)。

4.2.2 変数

変数の内容を、一度に1項目ずつ定義したい場合がよくある。この場合は以下のように書きたいだろう。

```
OBJS = foo.o
OBJS = $(OBJS) bar.o
OBJS = $(OBJS) baz.o
```

こう書くことで OBJS が `foo.o bar.o baz.o` と定義されることを期待したわけだが、実際には `$(OBJS) baz.o` と定義される。make は実施に使用されるまで変数を展開しないからである^{*4}。もしこれを実行すると、ルール内で OBJS を参照した場合に make は無限ループに入ってしまう^{*5}。そのため Makefile の多くには以下のような部分がある。

```
OBJS1 = foo.o
OBJS2 = bar.o
OBJS3 = baz.o
OBJS = $(OBJS1) $(OBJS2) $(OBJS3)
```

プログラマが変数宣言が長くなりすぎると感じたときには、上の例のように記述されることが多い。

変数展開は Linux プログラマが考慮すべき代表的な問題である。Linux に同梱されている GNU ツールは、他のシステムに同梱されているバージョンよりも高い機能を持っているのが通常であり、GNU make も例外ではない。GNU make の作者たちはこの問題を回避するために他の変数代入方法も用意している。しかし make のすべてのバージョンが GNU make の代替代入方法を認識できるわけではない。幸いにも、GNU make は Linux で書かれたソースコードを簡単に移植できるすべてのシステムで構築できる。あなたのコードを他のシステムに移植しようとする人に GNU make を使わせたい場合は、以下のように**単純変数代入**を行う。

```
OBJS := foo.o
OBJS := $(OBJS) bar.o
OBJS := $(OBJS) baz.o
```

`:=`演算子を使った場合、GNU make はルール内で式が使用されるまで待たずに、代入された時点で変数式の評価を行う。このコードによって OBJS には `foo.o bar.o baz.o` が含まれることになる。

単純変数代入は有益であることが多いが、GNU make には特に前述の問題に対処するための別の代入構文が用意されている。これは C 言語からそのまま取り入れたものだ。

```
OBJS := foo.o
OBJS += bar.o
OBJS += baz.o
```

4.2.3 サフィックスルール

サフィックスルールも、標準の Makefile を作成するか、あるいは GNU の拡張を利用するかを判断しなければならない事項である。標準のサフィックスルールは GNU のパターンルールより

*4 これは不便に感じられるかもしれないが、重要な機能であってバグではない。変数を展開しないことは暗黙の依存を定義する汎用サフィックスルールを作成する上できわめて重要である。

*5 GNU バージョンを含め、Linux に同梱されている make のほとんどのバージョンは無限ループに入ったことを検出して実行を終了し、エラーメッセージを返す。