Linux カーネル開発への参加方法

カーネル開発プロセスへの手引き

Jonathan Corbet、corbet@lwn.net

本書は、開発者（およびその上司）が開発コミュニティと一緒に作業する際のフラストレーションを最小限にすることを目的としています。ここでは、Linux カーネル（または一般のフリーソフトウェア）の開発に特に深い知識を持たない人でも理解できる形で、このコミュニティの仕組みを文書化しようとしています。一部には技術的な内容もありますが、本書の多くはプロセス志向の議論であり、その内容を理解する上でカーネルのプログラミングに関する深い知識は不要です。

1. カーネル開発プロセスへの手引き

本書は、開発者（およびその上司）が開発コミュニティと一緒に作業する際のフラストレーションを最小限にすることを目的としています。ここでは、Linux カーネル（または一般のフリーソフトウェア）の開発に特に深い知識を持たない人でも理解できる形で、このコミュニティの仕組みを文書化しようとしています。一部には技術的な内容もありますが、本書の多くはプロセス志向の議論であり、その内容を理解する上でカーネルのプログラミングに関する深い知識は不要です。

1.1: エグゼクティブサマリー

本セクションは、カーネル開発プロセス全般の話題と、開発者やその上司の方が遭遇しがちなフラストレーションについて述べています。カーネル・コードを公式の（メインライン）カーネルにマージすることが必要な理由は非常に多く、たとえばユーザーにとって自動的に利用可能になること、様々な形でのコミュニティのサポートが得られること、カーネル開発の方向付けに影響を及ぼすことが可能になることなどの理由があります。なお Linux カーネル用に提供されるコードは、GPL 互換の使用許諾で利用可能なものでなければなりません。

セクション 2 では、開発プロセス、カーネルのリリース・サイクル、およびマージ・ウィンドウの仕組みについて紹介し、パッチの開発、レビュー、マージ・サイクルの各フェーズについて説明します。また、一部にはツールやメーリングリストに関する説明も含まれます。カーネル開発に参入しようと志す開発者の最初の課題としては、バグの追跡と修正から始めるがお勧めです。

セクション 3 は、早い段階のプロジェクト計画について述べており、できるだけ早めに開発コミュニティを巻き込むことの重要性を強調しています。

セクション 4 はコーディング・プロセスについて、他の開発者が遭遇したことのあるいくつかの落とし穴について説明しています。パッチに対するいくつかの要件について述べ、またカーネル・パッチに間違いがないことの確認に役立つ一部のツールについて紹介しています。

セクション 5 では、パッチをレビューしてもらうために公開するプロセスについて説明します。開発コミュニティにまともに見てもらうためには、パッチは正しいフォーマットで、適切な説明を付けて、適切な場所に送付しなければなりません。このセクションに示されたアドバイスに従うことにより、その成果物が最適な形で受け入れられるようになります。

セクション 6 はパッチを公開した後に何が起きるかについて説明しています。この時点で、作業は完了までにまだまだ程遠い状態です。レビューアとの連携作業は開発プロセスの中で特に重要な部分ですが、ここでは、この重要な段階での問題発生を避けるための多くのヒントを提供しています。開発者は、パッチがメインラインにマージされたことで自分の仕事が終わったと考えてしまわないよう、注意が必要です。

セクション 7 では、git によるパッチ管理や、他人が公開したパッチのレビューなど、より上級の話題をいくつか紹介します。

セクション 8、および、9 では全体の結論をまとめ、またカーネル開発の詳しい情報が得られるソースへのリンクを示します。

1.2: 本書のねらい

Linux カーネルは、1000 人を大きく超えるコントリビュータ（貢献者）の協力による、現時点で世界最大クラスの最もアクティブなフリーソフトウェア・プロジェクトの一つであり、そのプログラムの規模は 600 万行を超えています。1991 年にわずかな規模で開始されたこのカーネルは、今では最も優れたオペレーティングシステムへと進化し、ポケットサイズのデジタル音楽プレーヤからデスクトップ PC、さらには現存する世界最大級のスーパーコンピュータに至るまでの、またこれらシステムの中間に位置するあらゆるシステムで使われています。Linux は、ほとんどあらゆる場面に対応可能な、堅牢で効率のよい拡張性に優れたソリューションです。

Linux の成長に伴い、その開発への参加を希望する開発者（および企業）の数も増加しています。ハードウェア・ベンダーは、Linux ユーザーにとって自社の製品が魅力的なものとなるよう、Linux が彼らの製品をサポートすることを望んでいます。Linux を

製品の一部として使う組み込みシステムのベンダーは、Linux が彼らの製品の機能を十分に引き出すことを望んでいます。Linux をベースとする製品の販売業者やその他のソフトウェア・ベンダーは、Linux カーネルの能力、性能、および信頼性に対して関心を持っています。また、エンドユーザーの場合も、そのニーズに沿わせるための変更を望むことも多いでしょう。

Linux が人々を魅了する素晴らしい特長のひとつは、これら開発者が Linux にアクセスできることです。すなわち、要求されるスキルを持つ人なら誰でもLinuxを改良することや、その開発の方向に影響を与えることができます。プロプラエタリーな（独自に開発された）製品の場合には、フリーソフトウェアのプロセスに特有なこの種のオープンさは得られません。とりわけ、このカーネルは他のフリーソフトウェア・プロジェクトよりも更にオープンなものです。平均すると 3 ヵ月のカーネル開発サイクルには 100 社以上、さらには企業以外からも含め、合計で 1000 人を超える開発者が関与しています。

カーネル開発コミュニティとの共同作業は特に難しいものではありません。しかし、そうは言うものの、多くの新規参加者がカーネルに関する作業を行おうとしたとき、その困難さを経験しています。カーネル・コミュニティは、毎日数千行ものコードが変更されていくという環境の中で、その運営が円滑に行えるよう（また高品質な製品が実現できるよう）、独特な運営方法を進化させてきました。このようなことから、Linux カーネルの開発プロセスがプロプラエタリな製品の開発方法とは大きく異なるということは特に驚くべきことではありません。

Linux カーネルの開発プロセスは新規参入の開発者にとっては奇妙で威嚇的なものに見えるかもしれませんが、その背景にはそれ相当の理由があり、また多くの経験に基づいています。カーネル・コミュニティのやり方を理解しようとしない開発者（更に悪いのはその方法を無視したり、迂回したりする開発者）には挫折感を伴うような困難が待ち構えています。開発コミュニティは、学習しようとする人を手助けしますが、他人に耳を傾けない人や開発プロセスに無関心な人のために費やす時間はありません。

この文書は、これを読んだ開発者がそのようなフラストレーションを避けられることを望んで作成されています。本書には実質的な内容が多く含まれており、これをよく読むことにより、短時間ですぐに元がとれます。開発コミュニティはカーネルの改良に協力できる開発者を常に必要としています。以下はそのような人や、またはあなたの部下に役立つはずです。是非、このコミュニティに参加してください。

1.3: コードを「メインライン」にマージすることの重要性

なぜカーネル・コミュニティとの共同作業の方法やコードのメインライン・カーネルへのマージの方法を学ぶ必要があるのかについて、疑問に思う企業や開発者が存在します（「メインライン」とは、Linus Torvalds が管理している、各 Linux ディストリビュータがベースとして使うカーネルのことです）。短期的に見れば、コードの提供（コントリビューション）に費用をかける必要はないと思えるかもしれません。すなわち、自分のコードは別途管理し、ユーザーを直接サポートする方が簡単に思えます。しかし実際は、コードを別途（「ツリー外で」）管理することは不経済なのです。

ツリーから外れたコードのコストを示す方法として、ここではカーネル開発プロセスに関係するいくつかの側面を示します。またこれらのほとんどは本書の中でも詳しく検討します。以下を十分に考えてみて下さい。

メインライン・カーネルにマージされたコードはすべての Linux ユーザーが利用できます。すなわち、それを含んだ意すべてのディストリビューションに自動的に含まれることになります。ドライバ・ディスクやダウンロード等も不要になり、また複数のディストリビューション、複数のバージョンに対応する煩雑さもなくなります。これは開発者にとっても、またエンドユーザーにとっても同様です。メインラインにマージすることで、配布とサポート上の多数の問題が解決します。

1. カーネルの開発者はユーザー空間における安定的なインタフェースを維持するように努めていますが、反面、カーネルの内部 API は常に流動的です。安定的なカーネル内部インタフェースを定めないことは、実は意図的な設計方針であり、これはいつでも基本的な改良を行って高品質のプログラムを実現できるようにするためなのです。しかし、この方針の一つの結果として、ツリーから外れたプロ

グラムは新しいカーネルを使うための維持管理が常に必要となります。ツリーから外れたコードを維持するには、そのコードを単に動作させるためだけでも相当な作業が必要になります。

これに対しメインラインに含まれたコードの場合、API 変更のために使えなくなったコードはその開発者が修正するという単純なルールがあるため、このような作業は不要となります。そのため、メインラインにマージされたコードの場合、その維持費用は非常に小さくなります。

- それだけでなく、カーネル内のコードは他の開発者によって頻繁に改良されます。ユーザー・コミュニティや顧客に自社の製品を改良する権限を与えることにより、すばらしい結果が得られます。
- カーネルのコードはメインラインへのマージの前後にレビューの対象となります。開発者のスキルがどんなに高くても、このレビュー・プロセスでは常にそのコードを改良する方法が見つかります。またレビューによって重大なバグやセキュリティ問題が発見される場合も多くあります。これは閉じられた環境で開発されたコード（たとえば内部の開発者だけでのみ開発、レビューを行った）の場合に特に顕著であり、そのようなコードは外部の開発者によるレビューで多くの利益が得られます。ツリーから外れたコードは低品質です。
- 開発プロセスに参加することで、カーネル開発の方向に影響を与えることができます。横から苦情を言うユーザーの意見も重要ですが、アクティブな開発者の意見はより強く、また必要性に合わせてカーネルを改良するための変更を行うことも可能です。
- コードを別に管理する場合、そのコードと同様な機能を実現する別のコードが他の開発者によりLinuxカーネルにマージされるという可能性が常にあります。そのような場合、自分のコードをマージしてもらうことは（不可能と言えるくらい）極めて困難になります。そうなると、(1) ツリーから外れた非標準の機能を永久に維持するか、(2) 自分のコードを捨て、ユーザーにツリー内のバージョンへと移行してもらうか、という不本意な選択肢に直面することになります。
- コードをコミュニティに提供することは、全体プロセスを機能させるための基本的な行為です。自分のコードを提供することにより、カーネルに新たな機能を追加することができ、同時に他のカーネル開発者が利用できる機能や用例も提供することができます。あなたが Linux 用のコードを開発したのであれば（または開発を検討中であれば）、このプラットフォームが継続して成功することに明らかな関心があるはずですが、コードの提供はその成功を確実にするための最も適切な方法の一つです。

上に示した理由は、全て、プロプラエタリなバイナリ形式で提供されるものも含め、ツリーから外れたカーネル・コードのすべてに当てはまります。しかし、バイナリのみのカーネル・コードの配布を検討する時に考慮すべき別の要素があります。これには以下の内容が含まれます。

- プロプラエタリなカーネル・モジュールの配布には法的な問題がつきまとい、少なくとも灰色の問題として存在します。すなわち多くのカーネル著作権保持者は、バイナリ・モジュールをカーネルから派生したプロダクトと考えており、従ってその配布は GNU の一般公有使用許諾に違反するものと考えます（これについては以下に補足します）。なお、本文書の筆者は法律家ではないため、ここに示す内容は決して法律上の助言として考えられては困りますが、ソースが公開されていないモジュールの法的地位は法廷でのみ判断されます。しかし、いずれにしても、これらモジュールには常に不確実性がつきまといます。
- バイナリ・モジュールは、カーネルの問題のデバッグを非常に難しくするため、ほとんどのカーネル開発者はデバッグそのものを行おうともしません。そのため、バイナリ・モジュールのみの配布をする場合、ユーザーはコミュニティからのサポートを得ることが困難になります。
- また、対象とするすべてのカーネル・バージョンやディストリビューションについて対応するモジュールを提供しなければならないため、バイナリ・モジュールの配布者にとってもサポートが困難になります。妥当と考えられる範囲に対応するだけでも、1 個のモジュールについて数 10 ものビルドが必要となる場合もあり、また、ユーザーはカーネルのアップグレードを行う都度、そのモジュールのアップグレードも別途必要になります。
- コード・レビューに関して既に述べた内容は、ソースが非公開のコードの場合は2倍に意味をもちます。このコードは全く公開されないため、コミュニティによるレビューも行われず、疑いの余地もなく、将来にわたり重大な問題を抱えこみます。

特に組み込みシステムのメーカーの場合、自己完結型の製品が出荷されること、固定されたカーネル・バージョンが使われ、リリース後の開発は不要なことから、これまでに述べた内容を

無視したいという誘惑に駆られる場合があります。この考え方は広範囲にわたるコード・レビューの価値を見逃しており、またユーザーに製品への機能追加を許可することの価値も見逃しています。しかし、これら製品もその商品寿命は限られており、その後は新バージョンのリリースが必要です。その場合、メインラインにあり、うまく維持されてきたコードを持つベンダーは、新製品を短時間で市場に投入することができるという点で特に有利な立場となります。

1.4: 使用許諾

Linux カーネルへのコード提供は複数の使用許諾契約に基づいて行われますが、コードはすべてカーネル全体の配布をカバーする GNU 一般公衆使用許諾契約バージョン 2 (GPLv2)と互換性を持つことが必要です。このことは、実際には提供されるすべてのコードが GPLv2(オプションとして、GPL の将来バージョンによる配布の許可を含めることもできる)、または、3 箇条の BSD 使用許諾に対応している必要があることを意味しています。カーネルへのコード提供は、適合する使用許諾契約が結ばれていない場合は受け付けられません。

カーネル用に提供されたコードに関する著作権の譲渡は不要です(要求もされません)。メインライン・カーネルにマージされたコードはすべて、その当初の所有権が保持されます。その結果、現在 Linux カーネルには数千人の所有者が存在します。

この所有権構成は、カーネルの使用許諾契約を変更しようとする試みはほぼ確実に失敗するということを意味しています。すべての著作権者の合意を得る(またはそのコードをカーネルから削除する)ことができる現実的なシナリオはほとんどありません。そのため、予測可能な将来においてバージョン 3 の GPL(GPLv3)へと移行する見込みもありません。

カーネル用に提供されるコードは合法的なフリーソフトウェアであることが必須です。この理由から、匿名(または変名)のコントリビュータのコードは受け付けられません。すべてのコントリビュータに対し、そのコードが GPL により Linux カーネルと共に配布可能な旨を述べた「サインオフ(承諾)」を行うことが求められます。その所有者からフリーソフトウェアとしての使用許諾が行われていないコード、またはカーネルの著作権に関係する問題が発生するおそれのあるコード(適切な予防措置を講じることなくリバース・エンジニアリングにより得られたコードなど)については、貢献を行うことができません。

著作権に関係する問題についての質問は Linux 開発のメーリングリストでもよくある質問です。通常、そのような質問に対する回答は多く寄せられますが、これらの質問に答えている人々は法律家ではなく、したがって、その助言は法律上の助言ではないということに留意しておくことが必要です。Linux のソースコードに関する法律的な質問がある場合、この分野に造詣の深い法律家に相談する以外の選択はありません。技術的なメーリングリストで得られた回答を信頼することは危険です。

２．開発プロセスの仕組み

1990 年代の初めにおける Linux カーネルの開発は、ユーザーや開発者の数も比較的少なく、かなり自由な形で行われていました。しかし数 100 万人のユーザー基盤と 1 年間に 2,000 人もの開発者が関係するようになり、カーネルの開発が円滑に進められるよう、多くのプロセスを発達させることが必要でした。このプロセスの中で効率よく開発をするためには、このプロセスの仕組みについて確実に理解することが必要です。

2.1: 全体像

カーネルは、ほぼ 2〜3 ヵ月に一回の割合で主要なリリースが行われます。最近のリリース履歴は以下のようなものです。

2.6.26 : 2008 年 7 月 13 日

2.6.25 : 2008 年 4 月 16 日

2.6.24 : 2008 年 1 月 24 日

2.6.23 : 2007 年 10 月 9 日

2.6.22 : 2007 年 7 月 8 日

2.6.21 : 2007 年 4 月 25 日

2.6.20 : 2007 年 2 月 7 日

2.6.x のリリースは新機能の追加や内部 API の変更等が行われたカーネルの主要リリースです。標準的な 2.6 リリースには、10,000 件を超える変更セットによる、数 10 万行のコード変更が含まれます。このように、2.6 は Linux カーネル開発の最新版です。またカーネルは連続的に主要な変更をマージしていくローリング開発モデルを使用しています。

各リリースにおけるパッチのマージに関しては、比較的単純な規律に従うようになっています。各開発サイクルの開始時に、「マージ・ウィンドウ」が開きます。この時点で、十分に安定していると見なされ開発コミュニティの承認が得られたコードがメインライン・カーネルにマージされます。この時、1 日あたり 1000 件近い変更（「パッチ」または「変更セット」）の割合で、新しい開発サイクルの大部分の変更（および主要な変更のすべて）がマージされます。

（余談ですが、このマージ・ウィンドウ中にマージされる変更は、いずれも突然何も見えない状態から出てくるのではなく、すべて十分に前もってその収集、テスト、計画が行われていることが重要な点です。このプロセスがどのように行われるのかについては、この後で詳しく説明します。）

このマージ・ウィンドウは 2 週間続きます。この期間の満了時、Linus Torvalds がウィンドウの閉鎖（クローズ）を宣言し、最初の「rc」カーネルをリリースします。たとえば最終的に 2.6.26 となるカーネルの場合、マージ・ウィンドウの終了時に発生するリリースは 2.6.26-rc1 と呼ばれます。この -rc1 リリースは、新機能をマージする期間が過ぎ、次のカーネルの安定化の時間が開始されたことのシグナルとなります。

次の 6〜10 週間は、問題を修正するためのパッチのみをメインラインへと提出することが必要です。たまに、より重大な変更が許可される場合もありますが、そのようなケースは稀です。マージ・ウィンドウ以外のタイミングで新機能をマージしようとする開発者に対しては一般に友好的でない対応が行われます。一般的なルールとして、希望する機能がマージ・ウィンドウ中にマージできなかった場合、次の開発サイクルまで待つのが最善の方法です。（時折発生する例外として、従来は未対応だったハードウェアのドライバの追加があります。そのドライバがツリー内のコードにまったく関係しない場合、それによりリグレッション（退行）が発生するおそれはなく、いつでも安全に追加することができます。）

修正はメインラインで置き換えられるため、パッチの割合は時の経過とともに低下します。Linus は新しい -rc カーネルを約 1 週間に 1 回リリースします。カーネルが十分に安定したと考えられるのは通常 -rc6 から -rc9 の間で、その後最終的な 2.6.x のリリースが行われます。この時点で、全体プロセスが再び開始されます。

1 例として、バージョン 2.6.25 の場合の開発サイクルの進捗を以下に示します（日付はすべて 2008 年です）。

1 月 24 日 : 2.6.24 の安定リリース

2 月 10 日 : 2.6.25-rc1、マージ・ウィンドウの終了

2 月 15 日 : 2.6.25-rc2

2 月 24 日 : 2.6.25-rc3

3 月 4 日 : 2.6.25-rc4

3 月 9 日 : 2.6.25-rc5

3 月 16 日 : 2.6.25-rc6

3 月 25 日 : 2.6.25-rc7

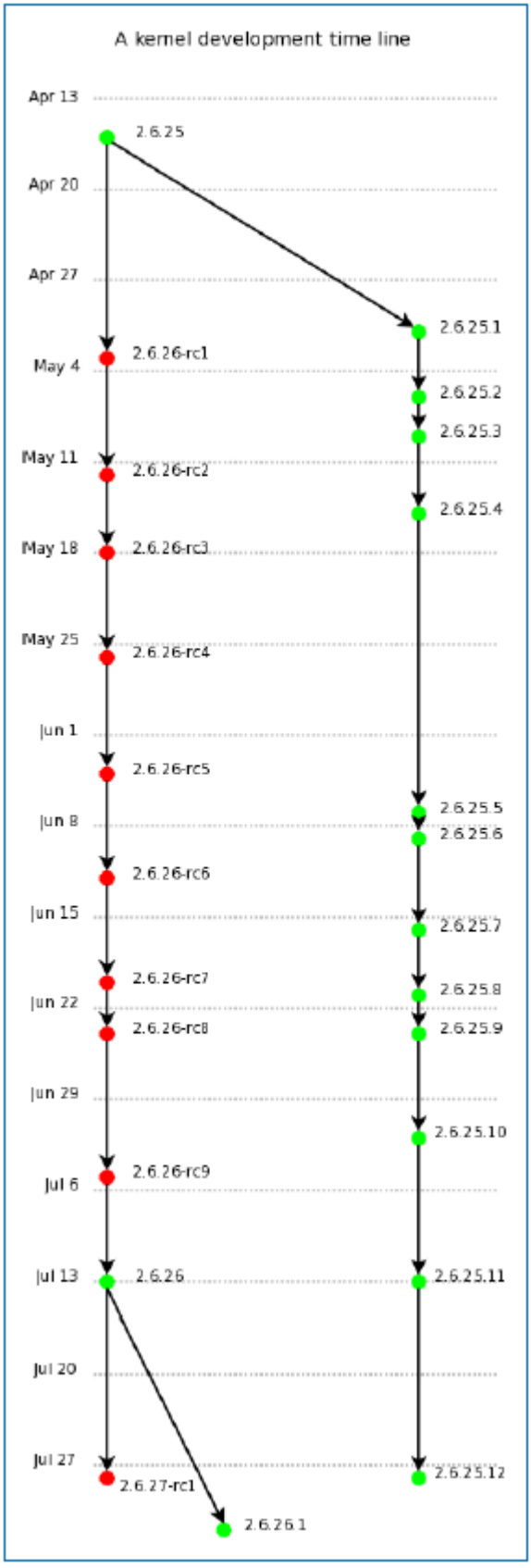
4 月 1 日 : 2.6.25-rc8

4 月 11 日 : 2.6.25-rc9

4 月 16 日 : 2.6.25 の安定リリース

ひとつの開発サイクルを終了して安定リリースを作成するタイミングをどのように決定しているのでしょうか？最も重要な測定基準（メトリック）は、過去のリリースにおけるリグレッション・リストです。バグがないことは良いことですが、過去に正常に機能したものでもシステムを壊すものは特に深刻です。この理由から、リグレッションを起こすパッチは好ましくないものと見なされ、安定化期間中に元の状態に戻される可能性が高くなります。

開発者の目標は、安定リリースの前に既知のリグレッションをすべて修正することです。しかし、このような規模のプロジェクトにはあまりにも多くの変数があるため、そのような完璧な目標は現実には達成困難です。これ以上最終リリースを遅らせることは単に問題を更に悪化させるだけ、という時が来ます。すなわち、次回のマージ・ウィンドウに予定される変更の件数が大きく積み上がり、そのため次のサイクルでリグレッションが更に増加することになるという状態です。そのため、ほとんどの 2.6.x カーネルは（深刻なものではないと望まれる）既知のリグレッションを少しだけ残した状態でリリースされます。



安定リリースが行われると、そのリリースの進行中の保守は「安定化チーム」（現在は Greg Kroah-Hartman と Chris

Wright が担当)に任されます。安定化チームは必要に応じ、安定リリースに対する更新リリースを「2.6.x.y」という番号方式でリリースします。更新リリースとして考慮されるためには、そのパッチが(1) 重大なバグを修正するもので、(2) 既に次回リリースの開発カーネルのメインラインにマージ済みのものであることが必要です。ここで先程の 2.6.25 の例を続けると、その履歴は（本文書の執筆時点で）以下のようになっています。

5 月 1 日 : 2.6.25.1

5 月 6 日 : 2.6.25.2

5 月 9 日 : 2.6.25.3

5 月 15 日 : 2.6.25.4

6 月 7 日 : 2.6.25.5

6 月 9 日 : 2.6.25.6

6 月 16 日 : 2.6.25.7

6 月 21 日 : 2.6.25.8

6 月 24 日 : 2.6.25.9

これらのカーネルの安定リリースの更新は約 6 ヶ月間行われ、その後の保守はすべてその特定のカーネルを出荷したディストリビュータの責任となります。

2.2: パッチのライフサイクル

各パッチは、開発者のキーボードから直接メインライン・カーネルにマージされるわけではありません。その間には（いくぶん非公式な形の）多少複雑なプロセスがあり、これは各パッチの品質レビューを確実に行うよう、またそのパッチによる変更を望ましい形でメインラインにマージするようにするためです。このプロセスは軽微な修正の場合は迅速な処理が可能ですが、賛否両論のある大きな変更については何年もかかる場合があります。多くの開発者が思い通りにならず挫折するのは、このプロセスに対する理解不足や、このプロセスを迂回しようとする行動が原因です。

そのような失敗を少なくするため、本書ではパッチをカーネルにマージするための方法について解説しています。以下に示すのは、このプロセスをやや理想的な形で解説した手引きです。より詳細な取り扱いについては後のセクションで解説します。

一般にパッチは以下のように作成されております。

- 設計 ― ここでは、パッチに対する要件（およびその要件を満足するための方法）が実際に計画される段階です。設計作業はコミュニティから独立して行われることも多いですが、この作業は可能な限りオープンな状態で行った方が後に再設計などで大きな時間をとられることがなくなり、良い結果が得られます。
- 早期のレビュー ― パッチを関連するメーリングリストに投稿（ポスト）すると、そのリストに参加している開発者が適宜コメントを寄せます。プロセスがすべて順調に進めばそのパッチに重大な問題がある場合ここで見つけだすことができます。
- 広範囲のレビュー ― パッチをメインラインにマージする状態が近くなった段階で、関連するサブシステムのメンテナーの承認を受けます。ただし、この承認は、そのパッチが最終的にメインラインにマージされることを保証するものではありません。パッチはメンテナーのサブシステム・ツリー上に現れ、後述のステージング・ツリー（準備段階のツリー）へと進みます。このプロセスがうまく機能した場合、この段階でそのパッチに対する広範囲なレビューが行われ、また他のメンバーが実施中の作業にそのパッチがマージされた結果に何らかの問題があればその内容が明らかになります。
- メインラインへのマージ ― すべてうまく進んだ場合、パッチは最終的に Linus Torvalds が管理するメインライン・レポジトリにマージされます。この時点で追加のコメントや問題点が浮上する場合もありますが、（パッチの）開発者はこれらに適切に対応して問題点を修正することが大事です。
- 安定リリース ― 最近ではパッチの影響を受ける可能性があるユーザーの数が多くなっているため、ここでも新しい問題が発生する可能性があります。
- 長期保守 ― 開発者はマージ済のコードについては忘れてしまうことも確かに可能ですが、そのような態度は開発コミュニティに対し悪い印象を与えます。コードをマージすることに

より、API 変更により発生する問題は誰かが修正してくれるという意味で、将来の保守作業の一部の負担が減らすことが可能です。しかし、そのコードが長期的に使われ続けるのであれば、開発元の開発者は引き続きそのコードに対する責任を負うべきです。

カーネル開発者（またはその事業主）が犯す最大の誤りは、このプロセスを「メインラインへのマージ」というひとつのステップで終わらせようとすることです。このようなアプローチは、まちがいなく関係者すべてにフラストレーションをもたらします。

2.3: パッチをカーネルにマージする方法

パッチをメインライン・カーネル・レポジトリにマージできるのは Linus Torvalds だけです。しかし、2.6.25 カーネルにマージされた 12,000 件を超えるパッチのうち、Linus 自身が直接選んだのは 250 件（約 2%）だけです。このカーネル・プロジェクトは長期にわたる成長を経て非常に大きな規模となったため、他から支援を受けずに 1 人の開発者、Linus Torvalds がすべてのパッチを検査し選択することは事実上不可能となっています。カーネル開発者達がこの成長に対処した方法は、「信頼のチェーン」を中心に構築された代理システムを利用することです。

カーネルのコードは、ネットワーク、特定アーキテクチャーへの対応、メモリー管理、ビデオ・デバイスなどいくつかのサブシステムへと論理的にブレークダウンされます。ほとんどのサブシステムに 1 人の保守メンテナーが指名され、開発者でもあるその保守メンテナーがそのサブシステム内のコードについて全体的な責任を持ちます。これらサブシステムのメンテナーは、その管理するカーネル部分の（あまり厳格でない）門番として活動し、（通常は）パッチをメインライン・カーネルに含める際の承認を行います。

サブシステムのメンテナーは、（必ずではありませんが）通常はソース管理ツールである git を使い、それぞれ自分自身用のカーネル・ソース・ツリーを管理します。その他の関連ツール（quilt や mercurial）も含め、git のようなツールを使うことでメンテナーは作者情報やその他のメタデータも含め、パッチのリストを追跡調査することができます。メンテナーは、自分のレポジトリにはあるがメインラインには含まれていないパッチをいつでも特定できます。

マージ・ウィンドウが開くと、最上位レベルのメンテナーは自分たちがレポジトリから選択したマージ用のパッチを「プル（引き出す）」するよう Linus に依頼します。Linus が同意すれば、パッチの流れは Linus のレポジトリへと流れ込み、メインライン・カーネルの一部となります。この「プル」操作で受け取った特定のパッチに対する Linus の関心の度合いは一様ではありません。明らかに、一部について、彼はかなり詳細なレベルまで確認を行っています。しかし、概して Linus は粗悪なパッチがアップロードされることはないという面でサブシステムのメンテナーを信頼しています。

一方、サブシステムのメンテナーも他のメンテナーからパッチをプルすることができます。たとえば、ネットワーキング・ツリーは、まずネットワーク・デバイス・ドライバや無線ネットワーク等の専用のツリーに集積されたパッチで構築されます。このレポジトリ・チェーンの長さは任意の長さに伸ばすことができますが、通常 2〜3 リンク以内におさまっています。そのチェーンのメンテナーはそれぞれその下位のツリーのメンテナーを信頼していることから、このプロセスは「信頼のチェーン」と呼ばれています。

明らかに、このようなシステムの場合、パッチのカーネルへのマージは適切なメンテナーを見つけることに依存しています。パッチを直接 Linus に送付するのは、一般に適切な方法ではありません。

2.4: ステージング・ツリー

サブシステム・ツリーのチェーンは、カーネルへとマージされるパッチの流れを導くものですが、ここでもうひとつ興味深い問題があります。それは、次のマージ・ウィンドウに向けて準備中のパッチのすべてを誰かが見たい場合はどうするか、という問題です。開発者は、問題となるコン

フリクト（競合）がないかどうかを確認するため、他の部分で準備中の変更にはどのようなものがあるかに関心があると考えられます。たとえば、コア・カーネル関数のプロトタイプを変更するパッチが、その関数の旧バージョンを使用する他のパッチとの間で競合を生じる場合などが考えられます。レビューやテストを行う人は、変更がすべてメインライン・カーネルにマージされる前に、それらの変更がマージされた状態にアクセスしたいと考えます。関心のあるすべてのサブシステム・ツリーから変更をプルすることも可能ですが、これは大きな作業であり間違いの発生しやすい方法です。

このような問題に対する解答がステージング・ツリーで、各サブシステム・ツリーをこのステージング・ツリーに集めてテストやレビューを行います。これらのツリーは従来より Andrew Morton が管理するツリーがありました。これは最初に開始されたときの「メモリー管理」という意味で「-mm」と呼ばれています。この -mm ツリーはサブシステム・ツリーの長いリストからパッチを集めてマージするもので、このツリーにはデバッグ作業に役立つパッチも含まれています。

それ以上に、この -mm には Andrew 自身が直接選定したパッチの大きなコレクションが含まれています。これらのパッチはメーリングリストにポストされたものや、指定のサブシステム・ツリーがないカーネル部分に適用されるパッチなども含まれています。その結果、 -mm は最後の頼みとなるサブシステム・ツリーとして機能します。すなわち、パッチをメインラインにマージするための明確な経路が存在しない場合、最終的には -mm を経由する可能性が高くなります。この -mm 内に蓄積される種々雑多なパッチは最終的には適切なサブシステム・ツリーに送られるか、または直接 Linus へと送られます。標準的な開発サイクルでは、メインラインにマージされるパッチのうち約 10%が -mm を経由します。

最新の -mm パッチは、常に以下のページのフロントページに置かれています。 http://kernel.org/

また -mm の最新の状態を見たい場合は、以下のサイトから「-mm of the moment」ツリーを入手できます。

http://userweb.kernel.org/~akpm/mmotm/

ただし、コンパイルさえ不可能という場合もあり、MMOTM ツリーの利用はかなり困難なものとなる可能性があります。

最近開始されたその他のステージング・ツリーとして、Stephen Rothwell が管理する linux-next があります。この linux-next ツリーは、その設計上、次回のマージ・ウィンドウが閉じた時にどのような形のメインラインが期待されるかを示すスナップショットとして提供されます。この linux-next ツリーは、その集約後、linux-kernel および linux-next の各メーリングリストで通知され、以下のサイトからダウンロードが可能です。

http://www.kernel.org/pub/linux/kernel/people/sfr/linux-next/

この linux-next に関する一部の情報は以下のサイトに集められています。

http://linux.f-seidel.de/linux-next/pmwiki/

この linux-next ツリーを開発プロセスにマージする方法は、現在も変化を続けています。この文書を執筆した時点は、（2008 年 7 月初旬）linux-next が関係した最初の完全な開発サイクル（2.6.26）が終りに近づいている段階です。これまでのところ、マージ・ウィンドウの開始前にインテグレーションに関係する問題を発見し修正する上で、このリソースの高い価値が実証されています。この linux-next が 2.6.27 マージ・ウィンドウのセットアップにどう使われたか、その詳しい内容については

http://lwn.net/Articles/287155/

をご覧ください。

一部の開発者は、将来の開発にも linux-next をターゲット

として使うべきだという提言を始めています。この linux-next ツリーは、メインラインよりも大きく先行する傾向があり、新しい成果がマージされるツリーとしての性格がより強くなっています。ただしこの考え方の弱点として、 linux-next の不安定さに起因する、開発ターゲットとして使う場合の困難さがあります。この話題については http://lwn.net/Articles/289013/ を継続的にご覧ください。linux-next に関してはすべてが依然として流動的です。

2.5: ツール

以上述べたように、カーネル開発プロセスはパッチの集合をあらゆる方面で把握し管理することのできる能力に大きく依存しています。それに適した強力なツールがなければ、カーネル全体が現在の状態やそれに近い状態で開発を続けることはできません。これらツールの使用方法を解説することは本文書の範囲を大きく超えていますが、ここでは多少のスペースを使い、リンク先をいくつか示します。

カーネル・コミュニティが圧倒的に主流のツールとして使うソースコード管理システムは git です。git は、フリーソフトウェアのコミュニティで開発され配布されている多数のバージョン管理システムのうちのひとつです。これは特に大きなレポジトリや多数のパッチを扱う場合の性能が優れているという意味で、カーネル開発用に良くチューニングされています。また、長い期間を経て改善されてはいますが、その習得や使用が難しいという評判もあります。カーネル開発を行う上で、仮に自分の仕事には使わない場合でも、他の開発者やメインラインで行われていることに遅れをとらないためには、git をある程度熟知することはほとんど必須要件となっています。

現在、ほとんどすべての Linux ディストリビューションには git が同梱されています。git のホームページを以下に示します。

http://git.or.cz/

このページには資料や手引きへのリンクが含まれています。特に、以下に示す「Kernel Hacker's Guide to git（カーネル・ハッカーの git ガイド)」には、カーネル開発に特有の情報が含まれています。

http://linux.yyz.us/git-howto.html

git を使わないカーネル開発者の中での最もポピュラーな選択は、まちがいなく以下の Mercurial です。

http://www.selenic.com/mercurial/

Mercurial の多くの機能は git と共通ですが、多くの人から使いやすいと評判のインタフェースが提供されています。

それ以外で知っておく価値のあるツールは以下の Quilt です。

http://savannah.nongnu.org/projects/quilt/

Quilt は、ソースコード管理システムというよりもむしろ、パッチ管理システムです。このツールは時系列による履歴の追跡は行わず、その代わりに改良されるコードのベースに対する特定の変更セットを追跡します。一部の有力なサブシステムメンテナーは、quilt を使って上流へと流す予定のパッチを管理しています。ある特定の種類のツリー（たとえば-mm）の管理を行う場合、そのような作業には quilt が最適なツールとなります。

2.6: メーリングリスト

Linux のカーネル開発作業の多くはメーリングリストを通じて行われます。このコミュニティのメンバーとしての完全な役割を果たすためには、少なくともひとつのメーリングリストに参加することが不可欠です。しかし、Linux のメーリングリストは開発者にとって潜在的な危険性をはらんでおり、電子メールの山に埋もれてしまったり、Linux メーリングリストで用いられる慣習と衝突してしまったりする危険性があります。

大部分のカーネル・メーリングリストは vger.kernel.org を中心としており、そのマスター・リストは以下の場所にあ

ります。

http://vger.kernel.org/vger-lists.html

それ以外にホストされているメーリングリストがたくさんあり、その多くは redhat.com にあります。

カーネル開発の中核となるメーリングリストは、もちろん linux-kernel です。このメーリングリストは非常にプレッシャーの高い場所です。すなわち、1 日に 500 件ものメッセージが発信され、ノイズも多く、会話の内容は高度な技術的内容で、また参加者はそれなりの礼儀正しさを心得ているとはかぎりません。しかし、カーネル開発コミュニティ全体が一堂に会する場所は他にありません。開発者がこのメーリングリストを避けてしまえば重要な情報が得られないというリスクを犯してしまう可能性があります。

ここで、linux-kernel で生き残るために役立つヒントがいくつかあります。

- 自分が主に使うメールボックスではなく、別のフォルダにリストが自動配信されるようにすること。長時間継続してこの流れを無視できることが必要です。

- すべての話についていく努力をしないこと。誰もそのようなことは考えません。関心のあるテーマや参加者によるフィルタリングを行うことが重要です（ただし、長く続いたやりとりの場合、メールの件名を変更せずに当初のテーマから外れてしまっている場合があります）。

- 「荒らし」を相手にしないこと。誰かがその場をかき回して怒りの反応を得ようとしている場合、これを無視します。

- linux-kernel のメール（または他のリストのメール）に回答する場合、関係者全員に伝わるよう Cc: ヘッダーはそのまま保存します。（明示的な依頼などの）特別な理由がない限り、決して宛先を変更、削除すべきではありません。自分が回答を送ろうとする相手が Cc: リストに含まれていることを常に確認します。この約束事があるため、自分のポストに対する応答に自分を CC : の宛先に含めるよう明示的に依頼することも不要になります。

- 質問する前にリストのアーカイブ（およびネットの全体）を検索し調べること。一部の開発者は、明らかに自分がやるべきこと（宿題）をやらずに質問する人に対して我慢ができません。 - トップ・ポスティング（回答の対象となる引用したテキストの上に自分の返事を置くこと）を避けること。これを行うと、その返事が読みにくくなり、貧弱な印象を与えます。

- 正しいメーリングリスト上で質問すること。Linux-kernel は全体会議の場ですが、すべてのサブシステムからの開発者を見つけるための最適な場所ではありません。

最後のポイントとして、「正しいメーリングリストを見つけること」は、初心者の開発者が間違えやすいことです。ネットワーキングに関係する質問を linux-kernel 上で行った場合、ネットワーキングに関係する開発者の多くは netdev リストに参加していることから、netdev リストで質問するよう求める親切なメールがだされるでしょう。それ以外にも、SCSI、video4linux、IDE、filesystem 等の各サブシステム用のリストがあります。メーリングリストを探す時の最適な情報は、カーネルのソースにパッケージされている MAINTAINERS ファイルにあります。

2.7: カーネル開発の開始

カーネル開発プロセスに着手する方法についての疑問は、個人からの場合でも企業からの場合でも、いずれも同じようなものです。さらに両者が同じように持っている問題は困難なものにしてしまう、様々な失策についてです。

企業は、開発グループを立ち上げるため、名の通った開発者に頼る場合が多くなります。たしかにこれはひとつの有効な手法となり得ます。しかし、これは費用がかさむ場合が多く、また経験の深いカーネル開発者の数を増やすことにはほとんどつながりません。多少の時間を投資して社内の開発技術者を育成し、Linux カーネルの開発を加速することも可能です。その事業主はこのような時間をかけることにより、カーネルと企業の両方をよく理解し、同時に他

のメンバーも教育できる開発者のグループを確保することができます。中期的に見れば、このようなアプローチの方がより有益な場合が多いでしょう。

当然のことですが、個々の開発技術者はどこから最初に手を付けたら良いか、途方に暮れてしまいます。人は最初から大きなプロジェクトを手がけることにはおじけづくもので、最初は何か小さなもので試してみたいと考えます。一部の開発者は、このような段階でスペルミスや軽微なコーディング・スタイルの問題を修正するためのパッチの作成に飛び込みます。残念ながらそのようなパッチは開発コミュニティ全体を混乱させるノイズとなり、その開発者はますます侮りを受けるようになります。このコミュニティへの参加を望む新人の開発者は、このような手段では自分の望んだ歓迎を受けることができません。

Andrew Morton は向上心のあるカーネル開発者に対し以下のように助言しています。

> すべてのカーネル開発の初心者が第一に考えるべきことは、「そのカーネルを、自分の扱えるすべてのマシン上で常に完全な形で実行可能にすること」です。通常、このためには、その物事をまとめるための他人との共同作業が必要になります（これには忍耐が必要です！）。しかし、これは問題ありません。これもカーネル開発の一部です。
>
> （http://lwn.net/Articles/283982/ ）

明らかに修正を必要とする問題がない場合の開発者への助言は、まず最新のリグレッションや未解決バグのリストを確認することです。修正を必要とする問題が不足することは決してありません。そのような問題に取り組むことにより、開発者はそのプロセスについての経験を積み、同時に開発コミュニティの他のメンバーから尊敬される関係を築くことができます。

3: 早い段階の計画

Linux カーネル開発プロジェクトを計画する場合、ただちにスタートしてコーディングを開始したい誘惑に駆られることがあります。しかし、他の大きなプロジェクトと同様、コードの最初の 1 行を書き始める前に、成功のための十分な基礎を固めておくことが必要です。最初の計画やコミュニケーションに多少の時間を費やすことが、後で非常に大きな時間の節約につながります。

3.1: 問題の特定

一般のエンジニアリング・プロジェクトの場合と同様、カーネルの機能拡張を成功させるには、まず解決すべき問題を明確に定義することが必要です。この段階は、たとえば特定のハードウェアに対するドライバが必要な場合など、一部のケースでは非常に簡単です。しかし、そのようなケースを除けば、一般に真の問題と提案されたソリューションとを混同してしまいがちであり、そのような場合には困難に陥ります。

たとえば、以下のような例を考えてみます。数年前になりますが、Linux のオーディオ系を担当していたある開発者が、システムの過大な遅延時間が原因で発生するドロップアウトやその他の副作用なしにアプリケーションを実行する方法を追求していました。彼らが到達した解は Linux Security Module (LSM)にフックを入れるカーネル・モジュールで、このモジュールは特定のアプリケーションに対しリアルタイム・スケジューラへのアクセスを許可するように設定するものでした。このモジュールが作成され、linux-kernel メーリングリストに送られましたが、ここで直ちに多くの問題が発生しました。

そのオーディオ系の開発者にとっては、このセキュリティ・モジュールは直面する問題解決に十分なものでした。しかし、カーネル・コミュニティ全体から見れば、これは LSM フレームワークの誤用であり、システムの安定度を損なう危険性があるものでした（すなわち、LSM は特定のプロセスに、他の方法では得られないような優先権を与えることを意図してはいません）。カーネル・コミュニティの希望する解は、短期的には rlimit のメカニズムを介したリアルタイム・スケジューリングを用いるものであり、また長期的には遅延時間を短縮するための現在進行中の取り組みによるものでした。

しかし、オーディオ・コミュニティは彼らが作成した解を捨て去ることができず、代替案を受け入れることに難色を示しました。結果としての意見の不一致により、彼ら開発者はカーネル開発の全体プロセスに幻滅を感じることとなり、そのうちの 1 人があるオーディオ系のメーリングリストに以下の内容をポストしました。

> 非常に優秀な Linux カーネル開発者は多いが、そのような開発者の声も、傲慢で愚かな群衆の大きな声にかき消されがちである。ユーザー要件についてこのような人々と話し合おうと努力することは時間の無駄である。彼らはあまりに「賢い」ため、弱者の声に対し耳を傾けない。
>
> (http://lwn.net/Articles/131776/).

この状況の真実は、これとは異なります。すなわちカーネル開発者達にとっては、特定のモジュールに対する関心より、システムの安定度や長期的な保守、およびその問題に対する正しい解決策を見つけることに対する関心の方がはるかに強いのです。この話の教訓としては、特定の解に注目するのではなく、その問題点に注目すべきだということです。 また、コード作成に投資する前に、開発コミュニティとの間で検討を行うべきだということです。

このように、カーネル開発プロジェクトを計画する場合には、以下のような簡単な質問に対する回答を得ておくことが必要です。

- 真に解決が必要な問題は何か？
- この問題の影響を受けるユーザーは誰か？その解決策はどのユースケースに対応するのか？
- 現在、その問題に対処する上で、カーネルにどのような不足があるのか？

可能な解決策の検討を開始することに意味があるのは、これらの答が得られた後に限られます。

3.2: 早い時期からの議論

カーネル開発プロジェクトを計画する場合、その具体化に着手する前にコミュニティとの間で議論を行うことは非常に意味のあることです。早めのコミュニケーションは、様々な面で時間の節約とトラブルの回避に役立ちます。

- その問題はカーネルが既に対応しているが、自分がまだ理解していない可能性があります。Linux カーネルは多数の機能を持つ大規模なものであり、その機能はすぐに分かるようなものではありません。カーネルの全機能が好ましい形で文書化されているとは限らず、また、よく見落とされることもあります。本文書の筆者自身も、既存のドライバと重複するドライバのポストを見たことがありますが、これはその作者が既存のドライバに気付かなかったことが原因です。「車輪」のような既存物を再発明するかのようなコードは単に無駄なだけでなく、メインライン・カーネルには決して受け入れられません。
- 提案された解にメインラインへのマージが認められないような要素が含まれている場合も考えられます。このような問題は、実際にコードを書く前に発見しておくことが望ましいものです。
- 他の開発者がその問題について既に検討を行っており、より優れた解のアイデアを持っているという場合も考えられます。その場合、その解の作成を好意的に支援してくれるかもしれません。

カーネル開発コミュニティの長年の経験から得られた明らかな教訓があります。それは、閉じられた環境で設計・開発されたカーネル・コードには必ずいくつかの問題が含まれており、その問題はコミュニティに対し公開されるまでは発見されない、ということです。これは極めて深刻な問題の場合もあり、カーネル・コミュニティの標準を満たすものになるまでに数ヶ月や数年を要する場合もあります。その例を以下にいくつか示します。

- Devicescape ネットワーク・スタックはシングルプロセッサ・システム用に設計され、実装されました。このコードはマルチプロセッサ・システム対応が行われるまで、メインラインへのマージは行えませんでした。改修のためのロック処理等をコード化するのは困難な仕事であり、その結果、現在 mac80211 と呼ばれているコードのマージは 1 年以上も遅れてしまいました。
- Reiser4 ファイルシステムには、中心的なカーネル開発者達の意見では、仮想ファイルシステム・レイヤで実現すべきだった多くの機能が含まれていました。また、ユーザーの操作に起因するデッドロック発生の可能性を認めない限り簡単には実装できないような機能も含まれていました。これらの問題の発見が遅れたため、またその一部の問題への対処が拒否されたこともあり、Reiser4 はメインライン・カーネルにマージできないままになっています。
- AppArmor セキュリティ・モジュールは、仮想ファイルシステムの内部データ構造を利用していましたが、その方法は安全性や信頼性に欠けると考えられるものでした。その後、このコードには大幅な修正が行われてきましたが、依然としてメインラインにはマージされていません。

このような事例の場合、もしカーネル開発者達との間で早い時期からの議論が行われていれば、このような苦労や追加作業の多くは回避できた可能性があります。

3.3: 誰と話すか？

開発者がその計画の公開を決心した時、次の問題は「どこからスタートするか？」ということになります。その答は、適切なメーリングリストと適切なメンテナを見つけることです。メーリングリストについて最も適切なアプローチは、MAINTAINERS ファイルで適切なポスト先を見つけることです。適切なサブシステム・メーリングリストがある場合、その関連するサブシステムの専門知識が豊富な開発者に出会える可能性が最も高く、またより充実した支援が得られる可能性があることから、linux-kernel にポストするよりもそのリストにポストした方が望ましいでしょう。

メンテナを見つけることは、これよりも少し難しい作業です。この場合も MAINTAINERS ファイルからスタートします。ただし、このファイルは最新のものではない場合も

あり、またすべてのサブシステムが掲載されているとは限りません。MAINTAINERS に記載されている人は、実際には現在その役割を果たしている人とは異なる場合があります。そのため、誰に連絡すべきかについて疑わしい場合に役立つ方法は、git（特に「git log」）を使うことで、これにより現在、誰がその対象サブシステムで活動しているかを調べることができます。誰がパッチの作成者か、またそのパッチに対し誰が「Signed-off-by（承認者）」の行を追加しているかを調べます。そのような人々が新規開発プロジェクトへの協力には最適です。 これらがうまく行かなかった場合、特定のコードに対応するメンテナを探すには Andrew Morton に相談するのも有効な方法です。

3.4: ポスティングの時期

可能な限り早い段階に計画をポストすることが有益です。解決する問題について記述し、またその具体化の方法についての計画があれば記述します。提供可能な情報はすべて提供することにより、そのプロジェクトに関する有用なインプットを開発コミュニティが提供しやすくなります。

この段階で発生する可能性のある最も思わしくない反応は、敵対的な反応ではなく、ほとんど、または全く反応がないことです。残念ながら (1)カーネル開発者は多忙な場合が多い、(2) 全体計画だけでその裏付けとなるコードをほとんど出さない（またはコードの見通しさえない）人が多い、(3) 他人がポストしたアイデアについてレビューを行ったりコメントしたりする義務は誰にもない、というのが実際のところです。コメント要求のポストに対しほとんどコメントが出なかった場合でも、そのプロジェクトに対する関心がないものとは考えないでください。また、残念ながら、出したアイデアに問題がないと仮定することもできません。このような状況での最善の行動は、コミュニティに対し自分のプロジェクトの状況を常に知らせておくことです。

3.5: 正式な承認を得ること

仕事が企業内で行われている場合（Linux カーネル開発の多くは企業内で行われています）、自社の計画やコードを公開のメーリングリストにポストするには、当然のことですが、適切な権限を持つメンテナーの許可を得ることが必要です。GPL 互換の使用許諾に基づくリリースが可能になっていないコードのポストは特に問題となる場合があります。カーネル開発プロジェクトのポストについて会社の経営者や法務担当の合意の時期が早ければ早いほど、すべての関係者が楽になります。

読者によっては、まだ正式にその存在を認めていない製品のためのカーネル開発について考えているかもしれません。事業主の計画を公開のメーリングリストで明らかにすることは不可能な場合があります。このような場合、その秘密主義が真に必要なものかどうかを検討する価値もあります。実際には開発計画を秘密にする必要性がない場合も多いからです。

当然ながら、会社として、開発プロセスの初期にはその計画を合法的に開示できない場合もあります。経験の深いカーネル開発者を持つ会社は、後で深刻なインテグレーション上の問題が発生することを避けられるために、オープンに議論を進めることを選択する場合もあります。そのような社内の専門家を持たない会社の場合、社外の開発者を雇い、機密保持契約を結んで計画のレビューを行わせることが最適な選択肢となる場合も多いでしょう。The Linux Foundation では、このような時に役立つ NDA プログラムを運用しており、その詳細は以下のサイトに紹介されています。

http://www.linuxfoundation.org/en/NDA_program

この種のレビューは、プロジェクトの開示なしに、後で重大な問題の発生を避けるのに有効です。

4: 適切なコードの作成

コミュニティ指向の設計プロセスについて語るべきことは多いですが、カーネル開発プロジェクトの証明はその成果としてのコードにあります。他の開発者による検証を受けてメインライン・ツリーにマージされるのはコードです。すなわち、プロジェクトの最終的な成功を決定するのはコードの品質です。本セクションではこのコーディング・プロセスについて考察します。まず、カーネル開発者が陥りやすい多くの過ちについて見ていきます。次に、**物事を正しく行うこと、またそのために役立つツール**に焦点を移して説明します。

4.1: 落とし穴

- コーディング・スタイル

カーネルには長い間、Documentation/CodingStyle に記述されている標準のコーディング・スタイルがありました。その間、そのファイルに記述された方法は、たかだか「助言」として位置付けられてきました。その結果、カーネル内にはこのコーディング・スタイルのガイドラインに合わない相当量のコードが含まれるようになりました。このようなコードの存在は、カーネル開発者にとって二つの危険性につながります。

そのひとつの危険性は、カーネルのコーディング標準は強制されるものではなく、問題にはならないという考え方です。実際、コードが標準に従って書かれていない場合、新しいコードのカーネルへの追加は非常に困難なものとなります。またレビューを行う前にコードの再フォーマットを要求する開発者も多く存在します。Linux カーネルのように巨大化したコードベースでは、各開発者がすべてのコードの内容を素早く理解できるよう、コードの均質性が要求されます。そのため、特異なフォーマットのコードが許される余地はありません。

場合により、Linux カーネルのコーディング・スタイルが、開発者の属する企業で義務付けられているスタイルと矛盾することがあります。そのような場合、コードをマージ可能にするためには、Linux カーネルのスタイルを用いなければなりません。コードをカーネルにマージするということは、様々な面でその管理をある程度放棄することを意味しており、これにはコードのフォーマット方法も含まれています。

もうひとつの誤りは、カーネル内にある既存コードはコーディング・スタイルの修正が必要だと仮定することにあります。開発者の中にはこの Linux 開発プロセスに慣れるため、またはカーネルの changelog に自分の名前を入れるため（またはその両方）の方法として再フォーマット用パッチの作成を行うことができます。しかし、単なるコーディング・スタイルの修正は開発コミュニティからはノイズとして見られ、冷たい対応を受けがちです。そのため、この種のパッチは極力避けるべきです。別の理由で作業を行いながらコードのスタイルを修正することは当然なことですが、単にコーディング・スタイルの修正のためだけにこれを行うべきではありません。

また、コーディング・スタイル文書は、決して違反してはならない絶対的な法律として読むべきものでもありません。スタイルに違反すべき妥当な理由がある場合（たとえば 80 文字の制限内に収めるために分割してしまうと極めて読みにくくなる行の場合など）は、そうするべきです。

- **アブストラクション・レイヤ**

コンピュータ・サイエンスの教授は学生に対し、柔軟性と情報隠蔽のためアブストラクション・レイヤを最大限に活用するよう教えます。たしかに Linux カーネルではアブストラクション（抽象化）を多用しています。数百万行のコードが関係するプロジェクトではこれなしに生き残る方法はありません。しかし、経験によれば、過度な、または中途半端なアブストラクションは中途半端な最適化と同様に有害なことが分かっています。アブストラクションは必要とされるレベルまで使用し、それ以上には多用しないことが必要です。

簡単な例として、すべての呼び出し元から常にゼロとして

渡される引数を持つ関数を考えます。その引数は、その関数が提供する柔軟性の拡張として、誰かが必要とすることになる場合を想定して保持しておくこともできます。しかしその頃には、この余分な引数を使用するコードは、それが 1 回も使用されなかったため気付かれないような微妙な形で壊れている可能性が高いものです。あるいは、その追加の柔軟性がまさに必要になった場合でも、プログラマーの当初の期待に一致した形でその引数が使われるとは限りません。カーネル開発者は未使用の引数を削除するパッチを定期的に提出します。一般的に言えば、この引数は、最初から無かった方が良かったということです。 ハードウェアへのアクセスを隠すアブストラクション・レイヤは多くのドライバを複数の OS 上で使えるようにするためによく使われますが、これらは特に問題視されます。そのようなレイヤはコードを見えにくくし、また性能上のペナルティが課される場合もあるため、Linux カーネルには含めるべきではありません。

一方、別のカーネル・サブシステムから大量のコードをコピーしている場合、その中の一部のコードを取り出して別のライブラリにするか、またはその機能を上位レベルで具体化することについて考慮すべきではないかと質問すべき時です。ひとつのカーネル内で同じコードを複製することは意味がありません。

- **全般的な #ifdef およびプリプロセッサの使用**

一部の C プログラマーにとって、C プリプロセッサはソースファイルに大きな柔軟性を持たせる効率的な方法と考えられ、強力な誘惑になると思われます。しかし、プリプロセッサは C ではなく、これを多用したコードは読みにくくなり、またコンパイラによる正当性チェックもより困難になります。プリプロセッサの多用は、ほとんどすべての場合において、クリーンアップの必要なコードの徴候です。

#ifdef を用いた条件付きコンパイルは実に強力な機能であり、カーネル内で使われています。しかし、 #ifdef ブロックが大量にばらまかれたコードを見たいとは思いません。原則として、#ifdef の使用は可能な限りヘッダー・ファイルに限定すべきです。条件付きコンパイルが行われるコードを、コードが存在しない場合は単純に空になる関数に限定することもできます。これによりコンパイラは最適化によりその空の関数へのコールを簡単に除外することができます。その結果、大幅にクリーン化されたコードが作成され、容易に理解できるようになります。

C プリプロセッサのマクロは､数式が副作用を伴って複数回評価されることや、型の安全性がないこと等、多くの害をもたらします。マクロを定義したいという誘惑に駆られた場合には、代替案としてインライン関数の作成を検討してみてください。結果としてのコードは同じですが、インライン関数は読みやすく、その引数を複数回評価することもなく、またコンパイラが引数および戻り値の型検査を行えるようになります。

- **インライン関数**

しかし、インライン関数にもそれ自身の危険性があります。プログラマーは、関数呼び出しを避けつつソースファイルをインライン関数で満たすことに、見かけ上の効率の良さに満足してしまう可能性があります。しかし、これらの関数は実際には性能を低下させることがあります。これらのコードは各呼び出しで複製されるため、コンパイル後のカーネルのサイズを膨張させてしまいます。これがプロセッサの命令キャッシュに対する圧力となり、実行速度が劇的に低下します。原則として、インライン関数は非常に小さいこと、また少ない使い方が必要です。結局のところ、関数コールの負担はそれほど大きくありません。多数のインライン関数を作成することは、中途半端な最適化の古典的な例です。

一般に、カーネルのプログラマーは危険を承知でキャッシュの影響を無視します。初心者向けのデータ構造のクラスで教えられる古典的な処理時間とメモリスペースのトレードオフは、最新のハードウェアの場合には当てはまらない場合が多いでしょう。大きなプログラムはコンパクトなプログラムよりも実行速度が低下する、という意味で「スペースは時間」と言われます。

ロック処理

2006 年 5 月、Devicescape ネットワーク・スタックが派手なファンファーレと共に GPL でリリースされ、メインライン・カーネルへのマージすべく提供されました。この貢献のニュースは歓迎されました。すなわち Linux における無線ネットワーキング対応は未だ標準以下と考えられており、Devicescape スタックはこの状況の改善を約束するものでした。しかし、このコードは 2007 年 6 月になるまで、実際にメインラインにはマージされませんでした（2.6.22）。一体、何が起きたのでしょうか？

このコードには、企業内の閉じた環境で開発されたことを示す多くの徴候がありました。しかし、特に大きなひとつの問題は、このコードがマルチプロセッサ・システムで動作するようには設計されていなかったことでした。このネットワーキング・スタック（現在の mac80211）をマージできるようにするため、ロック処理のための改修が必要でした。

昔はマルチプロセッサ・システムで必要な並行処理の問題は考えなくても Linux カーネルのコード開発が可能でした。しかし今では、たとえばこの文書でさえもデュアル・コアのラップトップで執筆しています。またシングルプロセッサ・システムの場合でも、応答性の改善を図るためにはカーネル内の並行処理レベルを高めることが必要です。ロック処理を考慮せずにカーネル・コードを書くことのできた時代は遠く過ぎ去ってしまいました。

複数のスレッドから同時にアクセスされる可能性のあるリソース（データ構造、ハードウェア・レジスタ等）はすべて、ロックによる保護が必要です。新しいコードはこの要件を念頭に置いて書かれますが、ロック処理を後から追加するという改修は非常に困難な仕事です。カーネル開発者は時間をかけて、利用可能なロック・プリミティブを十分に理解してから、その作業に適したツールを選定することが必要です。並行処理に対する注意が足りないと見られるコードは、メインラインへのマージまで困難な道を歩むことになります。

- **リグレッション**

最後に言及する価値のある危険性がこれです。大きな改善をもたらす可能性のある変更は魅力的ですが、それにより既存のユーザーが使用中の何かが壊れることがあります。この種の変更は「リグレッション（退行）」と呼ばれていますが、リグレッションはメインライン・カーネルで最も歓迎されないものとなっています。いくつかの例外を除いて、リグレッションを起こした変更は、修正がタイムリーに行われない場合、却下されます。最初からリグレッションを避けることが大事です。

その変更より問題が発生する人の数よりも、多くの人に良い効果をもたらすものであれば、リグレッションも正当化されるという議論がよくあります。つまり、1 個のシステムを壊すかわりに 10 個のシステムに新機能を提供するものであれば、その変更を行ってはどうかというものです。この質問に対する最善の回答は、2007 年 7 月に Linus が表明しています。

> そうすると、新しい問題を発生させても、そのバグ修正を行わないということになる。このようなやり方は狂気の沙汰であり、それでは真の進歩があったのか、まったく誰にも分からなくなる。それは 2 歩前進して 1 歩後退なのか、それとも 1 歩進んで 2 歩後退なのか？
>
> (http://lwn.net/Articles/243460/)

特に歓迎されないタイプのリグレッションは、ユーザー空間 ABI に対する何らかの変更です。一度ユーザー空間に提供したインタフェースは、永久にサポートしなければなりません。この事実が、ユーザー空間インタフェースの作成を特に困難なものとしています。すなわち、互換性のない変更方法は許されないため、最初から正しく作成することが必要です。このことから、ユーザー空間のインタフェースについては相当量の考察、明確な文書化、および広範囲なレビューが常に要求されます。

4.2: コードチェック用ツール

少なくとも今のところ、エラーのないコードを書くことはほとんど誰もが達成できない理想です。しかし、我々が望むことができるのは、コードがメインライン・カーネルにマージされる前に、このようなエラーを可能な限り多くとらえて修正することです。カーネル開発者達はこの目的のため、多岐にわたる問題を自動的に捕捉できる、すばらしいツールのセットを編成しました。コンピュータが捕捉した問題が、後でユーザーを悩ます問題とならないよう、当然ながら可能な限り自動化ツールを使うべきです。

最初のステップは、単にコンパイラが発生する警告に注意することです。最新バージョンの gcc は、多数の潜在的エラーの検出（と警告）が可能です。非常に多くの場合、これらの警告は実際の問題を示唆しています。規則として、レビュー用に提出されるコードはコンパイラの警告が発生しないものでなければなりません。警告が出ないようにするに際して、その真の原因の把握に努めること、原因に対処せず警告だけが無くなるような修正は避けるよう、十分に注意します。

また、コンパイラのすべての警告がデフォルトで有効になっているとは限らないことに注意が必要です。すべての警告を有効にするには、「make EXTRA_CFLAGS=-W」でカーネルのビルドを行います。

カーネルには、デバッグ機能を有効にするいくつかの設定オプションがあり、そのほとんどは「kernel hacking」サブメニューにあります。開発やテストに使うカーネルの場合、これらオプションのいくつかを ON にしておくことが必要です。

- 特に、以下のものは ON にしておくことが必要です。ENABLE_WARN_DEPRECATED、ENABLE_MUST_CHECK、および FRAME_WARN は、非推奨インタフェースの使用や、関数からの重要な戻り値を無視するといった問題に対する一連の警告を有効にするものです。これら警告の出力は冗長なこともありますし、カーネルの他の部分からの警告については気にする必要はありません。
- DEBUG_OBJECTS は、カーネルに生成される各種オブジェクトの生成から消滅までを追跡し、異常が発生したときに警告するコードを追加します。複雑なオブジェクトを作成してエクスポートするようなサブシステムを追加する場合、このオブジェクト・デバッグ・インフラの追加を考慮してください。
- DEBUG_SLAB は、メモリーの割当／使用に関係する各種のエラーを発見します。このオプションは開発用カーネルで使うべきです。
- DEBUG_SPINLOCK、DEBUG_SPINLOCK_SLEEP、および DEBUG_MUTEXES は、多くの一般的なロック・エラーを発見します。

その他、多くのデバッグ・オプションがありますが、その一部について以下で検討します。一部のオプションは性能に重大な影響があるため、常時使用すべきではありません。しかし、多少の時間をかけて利用可能なオプションについて学習すれば、すぐにその何倍も元が取れるようになります。

重いデバッグ・ツールのひとつに、ロック処理チェッカー「lockdep」があります。このツールは、システム内のすべてのロック（spinlock または mutex）について、その獲得と開放、ロック獲得の順序、現在の割込環境、その他の追跡を行います。ロックが常に同じ順序で獲得されることや、すべての状態に同じ割込条件が適用されていること等の確認を行います。言い換えると、lockdep はシステムがデッドロックする可能性のある多くのシナリオを発見することができます。ユーザーに展開済のシステムの場合、この種の問題の発生は開発者にとっても、ユーザーにとっても非常な苦痛を伴うものです。lockdep を使うことで、これらを前もって自動的に発見できます。何らかの重要なロック処理を含むコードについては、メインラインへの提出前に lockdep を有効にして実行することが必要です。

まじめなカーネル・プログラマーであれば、疑いの余地なく、エラーに終わる可能性のあるすべての処理（メモリー割り当てなど）の戻りステータスをチェックするはずです。

しかし実際には、そのエラーの結果としての実行される障害回復パスについてはテストが疎かになりがちです。未テストのコードは不正な場合が多く、これらエラー処理パスのすべてを数回実行することにより自分のコードに対する自信が深まります。

Linux カーネルには、特にメモリー割り当てに関係する部分について、まさにこれを行う障害挿入（フォールトインジェクション）の仕組みが用意されています。障害挿入を有効とし、メモリー割り当て失敗のパーセンテージを指定して、メモリー割り当てを失敗させることができます。また挿入する障害をコード内の特定の範囲に限定することができます。障害挿入を有効に設定してシステムを動作させることにより、プログラマーは不具合が発生したときのコードの反応の仕方を確認することができます。この機能の使用法の詳細については Documentation/fault-injection/fault-injection.text を参照してください。

その他の種類のエラーは、「sparse」という静的分析ツールで発見することができます。このsparseは、ユーザー空間とカーネル空間のアドレスの混同、数値のビッグエンディアンとリトルエンディアンの混用、ビット・フラグのセットが予期されている時に整数値を渡した場合等についてプログラマーに警告を与えます。このsparseは別にインストールする必要があり（ディストリビュータのパッケージに含まれていない場合、http://www.kernel.org/pub/software/devel/sparse/で入手可能です）、インストール後に、自分のコードのmakeコマンドに「C=1」を追加して実行します。

その他の移植時の障害については、そのコードを他のアーキテクチャー用にコンパイルして発見するのが最善の方法です。例えば、S/390 システムまたは Blackfin 開発ボードを持っていない場合でも、コンパイルのステップを実行することは可能です。様々な x86 システム用のクロス・コンパイラは以下のサイトにあります。

http://www.kernel.org/pub/tools/crosstool/

多少の時間をかけて各コンパイラをインストールし使ってみることは、将来の困惑を避ける上で役立ちます。

4.3: ドキュメンテーション

カーネル開発におけるドキュメントの作成はルールというよりは、むしろ例外といえる程でした。たとえそうであっても、適切な文書を作成することで新しいコードのカーネルへのマージが容易になり、他の開発者も楽になり、ユーザーにも役立ちます。多くのケースにおいて、文書の添付は基本的な義務となってきています。

あらゆるパッチについて、その最初の資料は changelog（更新履歴）です。このログには、解決される問題、解決方式、パッチ作成に関与した人々、性能への何らかの影響、およびそのパッチの理解に必要なその他の情報について記述することが必要です。

新たなユーザー空間インタフェースを追加するコード（sysfs ファイルや/proc ファイルを含む）の場合、ユーザー空間の開発者がその内容を把握できるインタフェース資料を含めることが必要です。この資料の形式や提供が必要な情報の内容については Documentation/ABI/README を参照してください。

Documentation/kernel-parameters.txt ファイルには、Linux カーネルのすべてのブートタイム・パラメータが記述されています。新規パラメータを追加するパッチの場合、このファイルに該当する項目を追加することが必要です。

新しいコンフィギュレーション・オプションがある場合、必ずそのオプションの内容と、ユーザーがそのオプションを選択する場合について明確に説明するヘルプ・テキストを添付しなければなりません。

多くのサブシステムの内部 API 情報は特別なフォーマットのコメントで文書化されており、これらのコメントは「kernel-doc」スクリプトを通じ、様々な方法で抽出してフォーマット化することができます。kerneldoc コメントを持

つサブシステムを対象とした開発を行う場合、既存のコメントを維持しつつ、必要に応じ外部から利用可能な関数についてコメントの追加を行うことが必要です。そのように文書化されていない部分であっても、将来のためにkerneldocコメントを追加しても何ら害はありません。実際、これはカーネル開発の初心者にとっては役に立つ行為です。これらコメントのフォーマットやkerneldocテンプレートの作成方法については、Documentation/kernel-doc-nano-HOWTO.txtファイルに示されています。

誰でも既存のカーネル・コードを読み進めてみると、多くの場合はコメントがないことの方に気が付きます。繰り返しになりますが、従来に比べて新しいコードに対する期待度は高まっています。コメントのないコードのマージはより一層困難になります。とは言うものの、過剰なほど詳細にコメントされたコードも望まれません。コードはそれ自体による読み取りが可能なものとすべきであり、コメントはより微妙な内容について説明するものです。

特定の内容については常にコメントが必要です。メモリーバリアを使う場合、そのバリアが必要な理由を説明する1行の追加が必要です。データ構造のロッキング・ルールについては、一般にどこかで説明することが必要です。一般に、主要なデータ構造については分かりやすい資料が必要です。コードの個々のビット間の、明白でない依存関係についても指摘しておくべきです。コードの整理係が誤って「クリーンアップ」したくなるような部分があれば、その部分がそのように設計されている理由を述べるコメントが必要です。その他、必要な部分にコメントします。

4.4: 内部 API の変更

カーネルがユーザー空間に対し提供するバイナリ・インタフェースは、極めて厳しい状況の場合以外、維持されます。しかし、カーネルの内部プログラミング・インタフェースは非常に流動的であり、その必要性が発生したときには変更することが可能です。カーネル API 周辺に手を入れる必要性を感じたとき、または自分のニーズに合わないため特定のカーネル機能を使わないと決めたような場合、それはその API の変更が必要な徴候かもしれません。カーネル開発者は、そのような変更を行う権限を持っています。

もちろん、これには多少の問題点があります。API の変更は可能ですが、そのためには十分な正当化が必要です。そのため、内部 API を変更するパッチを作成する場合、その変更がどのような内容なのか、なぜ必要なのかを説明する文書を付けなければなりません。この種の変更は大きなパッチの中に埋めてしまわず、別々のパッチに分けることが必要です。

もうひとつの問題点として、一般に内部 API を変更する開発者には、その変更により影響を受けるカーネル・ツリー内のコードの修正が求められるということがあります。広く使用されている関数の場合、この仕事は文字通り数百件から数千件もの変更につながり、またその多くは他の開発者が行っている作業との間でコンフリクトを発生させる可能性が高くなります。言うまでもなくこれは大きな作業であり、その正当化理由が堅固なものであるという確信が必要です。

非互換の発生する API 変更を行う場合、可能な限り更新されていないコードもコンパイラにかけることが必要です。これにより、ツリー内でそのインタフェースが使用される部分をすべて特定することができます。また、これによりツリー外のコードを作成した開発者に対しても、対応すべき変更があるということが警告されます。ツリー外のコードへの対応はカーネル開発者が心配すべきことではありませんが、必要以上にツリー外の開発者を困らせる必要はありません。

5: パッチのポスティング

遅かれ早かれ、自分の成果物をコミュニティに提示してレビューを受け、最終的にはメインライン・カーネルにマージするという時がきます。当然ながら、カーネル開発コミュニティではパッチのポストに関する一連の約束事や手順を発展させたものであり、これらに従うことは関係者すべてにとって有益なことです。本書では、これらの内容について簡単に述べてあります。またより詳しい情報はカーネル・ドキュメンテーションのディレクトリにある SubmittingPatches、SubmittingDrivers、SubmitChecklist の各ファイルにも記述されています。

5.1: ポスティングの時期

パッチが完全に「準備完了」の状態になるまでポストは避けたいという誘惑を持つこともあるでしょう。単純なパッチの場合、これで問題はありません。しかし、その開発が複雑なものだった場合、その完成までの過程でコミュニティからのフィードバックを得られることに大きな意味があります。そのため、進行中の成果をポストすること、あるいは関心のある開発者がその進捗をいつでも把握できるよう、git ツリーを提供することも考慮すべきです。

まだマージ可能なレベルには達していないと考えられるコードをポストする場合、そのポストの中でその旨を述べることもいいでしょう。また、残りの主な作業内容や、既知の問題点についても言及します。未完成と分かっているパッチに注目する人は少ないでしょうが、その開発を正しい方向に向けるために役立ちたいと考える人は参加してきます。

5.2: パッチ作成の前に

開発コミュニティにパッチを送る前にやるべきことは多数あります。これには以下の内容が含まれます。

- 自分にできる範囲でコードのテストを行う。これには、カーネルのデバッグ・ツールを使うこと、オプションのあらゆる合理的な組み合わせに対しカーネルのビルドが可能なことを確認すること、クロス・コンパイラを使い、異なる CPU アーキテクチャー用にビルドを行うこと等が含まれます。
- そのコードがカーネルのコーディング・スタイル・ガイドラインに適合していることを確認してください。
- その変更は性能に影響しませんか？影響がある場合、その変更の影響度（または効果）を示すベンチマークを行い、その結果をまとめてパッチに添付します。
- コードをポストする権限が自分にあることを確認します。その仕事が企業のために行われたものであれば、その企業がその成果物の権利を持っている可能性が高く、GPL に基づくリリースに合意していなければなりません。

一般に、コードをポストする前に少し余分に考えるという努力は必要で、また十分に価値があることです。

5.3: パッチの作成

ポスト用のパッチ作成には驚くほど多くの作業が必要になる場合があります。しかし、繰り返しになりますが、仮に短期的なものであっても、ここで時間を節約しようという考えはお勧めできません。

パッチはカーネルの特定のバージョンに対して作成されなければなりません。原則として、パッチは Linus の git ツリーにある最新のメインラインをベースにすることが必要です。ただし、より広範囲なテストやレビューを行うため、 -mm や linux-next、あるいはサブシステムのツリーに対応するバージョンの作成が必要となる場合もあります。そのパッチの分野やその他の進行中の状況により、これら別のツリーをベースにした場合、コンフリクトの解決や API 変更への対応などを含む多大な作業が必要になる場合があります。

最も単純な変更のみを単一のパッチにすべきです。それ以外は一連の論理的なパッチ群として作成することが必要です。パッチの分割には多少の技術が必要です。開発者によっては、コミュニティの期待する方法に従うにはどうしたら良いか、長い期間をかけて理解しなければならない場合があります。一方、すぐに役立ついくつかの経験則もあります。

- ポストする一連のパッチ群は、まず間違いなく自分の作業用のリビジョン管理システムにある変更とは別のものになります。まず、自分の行った変更がその最終的な状態にあることを確認し、次に他

人が理解できるような方法でこれらを分割します。開発者達は、その変更に至るまでに通過した経路にではなく、自己完結の状態にある独立した変更に関心があります。

- 個々の論理的に独立した変更を別々のパッチとしてフォーマットすることが必要です。これらの変更には小さなもの（例：「この構造にフィールドをひとつ追加」）も、大きなもの（例：新しい大きなドライバの追加）もありますが、いずれも概念的には 1 行で説明が行える程度に小さいことが必要です。各パッチはそれ自体でレビューが可能で、その記述された通りの実行内容を検証可能な、具体的な変更になっていることが必要です。
- このガイドラインを別な言葉で言い換えると、「同じパッチに別の種類の変更を混在させないこと」ということになります。ひとつのパッチで重要なセキュリティ上のバグを修正し、いくつかの構造を再編成し、更にコードの再フォーマットを行うといった場合、そのパッチは無視されてしまい、重要な変更が失われる可能性が高くなります。
- 個々のパッチはそれぞれカーネルにマージされ、そのビルドと実行が正しく行えなければなりません。すなわち、その一連のパッチ群が途中で中断されたとしても、カーネルとしては依然として動作可能でなければなりません。リグレッションを発見するために「git bisect」ツールが使われる場合、パッチの部分適用が一般的によく行われます。これによりカーネルが壊れてしまうような場合、問題点を突き止めるための作業に従事している開発者やユーザーに多大の迷惑をかけることになります。
- しかし、これを過度に行うことも避けなければなりません。最近、ある開発者はひとつのファイルに対する一式の修正を 500 個の独立したパッチとしてポストしましたが、この行為はそのカーネルのメーリングリストで彼を一番の人気者にはしませんでした。1 個のパッチは、それが論理的な変更内容を 1 個だけ含むものである限り、合理的な範囲で大きなものとすることができます。
- ひとつの一連のパッチ群の中の最後のパッチによりそのパッチ全体が有効になるまではそのインフラが使用されないようにするという方法で、まったく新しいインフラを追加することは魅力的かもしれません。しかし、可能な限り、このような方法は避けるべきです。もしその一連のパッチによるリグレッションが発生した場合、原因の切り分け作業により、実際のバグはどこか他にあった場合でも、その最後のパッチが問題の原因として特定されてしまうことになります。新しいコードを追加するパッチは、可能な限りそのコードを直ちに有効にすることが必要です。

完全なパッチ群を作成する作業は、「実際の作業」が終了した後に相当な時間と検討を要する、非常に困難なプロセスとなる場合があります。しかし、正しく行われれば、その費やした時間には高い価値があります。

5.4: パッチのフォーマッティング

この段階でポスト用の完全なパッチ群が得られたことになりますが、まだ多くの作業が残されています。各パッチは、その目的を素早くかつ明確に他人に伝えるためのメッセージへと書式を整えなければいけません。そのため、各パッチは以下に示す内容で構成されます。

- そのパッチの作者名を示す、「From:」行（オプション）。この行が必要なのは他人のパッチをＥメールで送付するときに限られますが、疑問に思ったときは追加しておいても害はありません。
- そのパッチが実行する内容の 1 行の記述。このメッセージは、そのパッチのスコープが何かを理解するのに十分なものである必要があります。これは「簡易版」の changelog に表示される内容となります。このメッセージは通常、関係するサブシステム名を最初に置き、次にそのパッチの目的を記述します。たとえば、以下のようになります。

gpio: fix build on CONFIG_GPIO_SYSFS=n

- 空白行に続く、パッチ内容の詳細な説明。この説明は必要に応じて長文にすることができ、そのパッチの機能と、カーネルへの適用が必要な理由について記述します。
- 1 行以上のタグ行と、少なくとも、パッチの作者による 1 行の「Signed-off-by:」行。タグについて以下に詳しく説明します。

通常、以上の 3 項目がその変更をリビジョン管理システムに入力する際のテキストとなります。これらの項目の後には以下の内容が続きます。

- パッチ本体、「-u」オプションのユニファイドパッチ形式にて。「-p」オプションを diff に使用すると、その変更に関数名が関連付けされ、他の人にとって読みやすいパッチになります。

パッチ内の無関係なファイル（ビルド・プロセスで生成されたファイルやエディターのバックアップ・ファイルなど）に変更を含めることは避けることが必要です。この点についてはドキュメンテーション・ディレクトリ内の「dontdiff」ファイルが役立ちます。「-X」オプションを付けて diff に渡してください。

上述の各タグは、そのパッチの開発にどれだけ多くの開発者が関係しているかの説明に使われます。これらは SubmittingPatches ドキュメントに詳しく解説されていますが、以下にその簡単なまとめを示します。これらの行はそれぞれ以下の構成を持っています。

tag: 氏名<e-mail address> その他のオプション

一般によく使わるタグには以下のものがあります。

- Signed-off-by: これは、そのパッチをカーネルに含めるために提出する権限がその開発者にあるということを示す、開発者の証明書です。これは「Developer's Certificate of Origin（作成元に関する開発者の証明書）」への同意を示すものであり、その完全なテキストは Documentation/SubmittingPatches にあります。適切なサインオフが行われていないコードはメインラインにマージできません。
- Acked-by: これはそのパッチをカーネルに含めることが適切であるという同意を示す、別の開発者（通常は該当するコードのメンテナー）の証明書です。
- Tested-by:そのパッチのテストを行い、正常に動作することを確認した人の名前を示します。
- Reviewed-by: ここに名前を記された開発者がそのパッチが適正かどうかのレビューを行ったことを示します。詳細については Documentation/SubmittingPatches にあるレビューアのメッセージを参照して下さい。
- Reported-by: そのパッチ修正の対象となった問題を報告してくれたユーザーの名前を示します。このタグはコードをテストして不具合があれば報告してくれる（正当に評価されないことの多い）人々の功績を評価するために使われます。
- Cc: パッチのコピーを受け取り、それについてコメントした人の名前を示します。

パッチにタグを追加する際には注意が必要です。その人の明示的な許可なしに名前を記載してもよいのは Cc: の場合のみです。

5.5: パッチの送付

パッチをメールで送信する前に考慮すべきことがいくつかあります。

- 自分が使用しているメーラーがそのパッチを壊すことがないと確信できますか？メール・クライアントにより行われる不必要な余白の変更や行折返しが含まれたパッチは受け取り側の環境に適合せず、その細部を検討してもらえない場合が多くなります。もし何らかの疑念があれば、そのパッチを自分宛に送り、そのまま正しく表示されることを確認してください。

Documentation/email-clients.txt には、パッチを送信する際に特定のメール・クライアントを正しく動作させるためのヒントがいくつか記載されています。

- 自分のパッチにばかげた間違いが含まれていない自信がありますか？パッチは常に scripts/checkpatch.pl を実行し、そこで出力される問題指摘に対処することが必要です。ただし、checkpatch.pl はカーネル・パッチのあるべき姿について十分に検討した結果として具体化されたものですが、自分よりも有能だということはないということを忘れないでください。checkpatch.pl の指摘による修正を行うとコードが更に悪化するような場合もあります。そのような場合、修正は行わないでください。

パッチは必ず平文テキストで送付することが必要です。添付ファイルにはしないでください。レビューアが返信する際にパッチの該当部分の引用が難しくなってしまいます。パッチは添付ファイルではなく、自分のメッセージ中に直接入力して下さい。

パッチをメールで送る場合、そのパッチに関心を持つ可能性がある人にもコピーを送ることが大事です。他の一部のプロジェクトとは異なり、カーネル・プロジェクトではコピー先が十分以上に多過ぎることを奨励しています。メーリングリスト上へのポストを関連する人々が必ず見るとは思わないでください。特に、コピー先には以下を含めることが必要です。

- 影響を受けるサブシステムのメンテナー　既に述べた通り、まずMAINTAINERS ファイルでこれらの人々を探します。
  - 同じ分野に関係する他の開発者、特に現在そこで活動中と思われる開発者。自分の対象とするファイルで過去に修正を行ったことのある人を git で確認することも有効です。
  - それがバグ報告や機能要求に対応したものであれば、その当初のポスト元にもコピーを送ります。
  - 関連するメーリングリストにコピーを送るか、または該当するリストがない場合は linux-kernel に送ります。
  - そのパッチがバグ修正用の場合、その修正を次回の安定バージョンにも適用すべきか検討し、適用が必要な場合、stable@kernel.org もパッチのコピーを受け取る必要があります。また、パッチ自体のタグにも「Cc: stable@kernel.org」を追加します。これにより、その修正がメインラインにマージされた時、安定化チームがその通知を受け取れるようになります。

パッチの受取人を選択する場合、最終的にそのパッチを承認しマージさせてくれる人は誰かを考えておくとよいでしょう。パッチをLinus Torvaldsに直接送り、彼にマージしてもらうことも可能ですが、通常はそのような方法はとられません。Linus は多忙であり、カーネルの特定部分の世話をするサブシステムのメンテナーが存在します。通常はメンテナーにパッチをマージしてもらうことになります。明確なメンテナーが存在しない場合、多くの場合は最後の手段として Andrew Morton がそのパッチを扱います。

パッチ本体には適切な表題行が必要です。パッチ表題行の正規のフォーマットは以下に示すような形になります。

[PATCH nn/mm] subsys: パッチの説明（1 行）

ここで、「nn」はそのパッチの順番、「mm」はその系列に含まれるパッチの合計数、「subsys」は影響を受けるサブシステムです。当然ながら単一の独立したパッチの場合「nn/mm」は省略可能です。

大きなパッチ群の場合、パート・ゼロとして前書きを送ることが慣例になっています。ただし、この約束事は例外なく守られているわけではありません。これを使う場合、その前書きの内容はカーネルの changelog には含まれないことに注意してください。そのため、パッチ自体に完全な changelog 情報を含めるようにして下さい。

原則として、複数の部分で構成されるパッチの場合、2 番目以降の部分は最初の部分に対する返信として送ることが必要です。これによりすべての送信が受信側ではひとつのスレッドでつながります。git や quilt などのツールには、ひとつのパッチの組を適切にスレッド化して送信するコマンドが含まれています。ただし、長い組のパッチで git を使う場合、「--no-chain-reply-to」オプションを設定し、過度に深いネスティングは避けるようにして下さい。

6: フォロースルー

この段階になると、これまでに説明したガイドラインに従い、また自分自身の設計スキルを活用し、完全なパッチ群をポストしたことになります。経験の深いカーネル開発者でさえ犯してしまうことがある最大の間違いのひとつは、この段階で自分の作業は完了したと結論付けてしまうことです。実際には、パッチのポスト全体プロセス中の次の段階への移行のみを示すものであり、その後に行うべき作業もおそらく相当に多いものと考えられます。

最初のポスト時から非常に優れており、その後の改良の余地がないようなパッチは稀です。カーネル開発プロセスではこの事実がよく認識されているため、ポスト後のコードの改良に向けた動きに重点が置かれています。そのコードの作者には、カーネルの品質基準が満足されるまで、カーネル・コミュニティとの共同作業に参加することが期待されます。このプロセスに参加しなかった場合、そのパッチがメインラインに含められなくなる可能性が非常に高くなります。

6.1: レビューアとの共同作業

何らかの重要性があるパッチの場合、コードをレビューした他の開発者からのコメントが多く寄せられます。多くの開発者にとって、これらレビューアとの作業がカーネル開発プロセスの中でも最もフラストレーションが大きい部分です。ただしいくつかの点に留意することで、このフラストレーションは相当に軽減されます。

- そのパッチについて十分な説明があれば、レビューアはそのパッチの価値や、その困難なパッチ作成に至った理由を理解してくれます。しかし、その価値が理解されただけでは、「このコードを含めたカーネルを 5～10 年後に保守する場合はどうなるか？」という彼らの基本的な質問を抑えることはできません。コーディング・スタイルの調整から大幅な書き直しまで、多くの変更が要請される可能性がありますが、これらはいずれも Linux は 10 年後にも使われており、開発が継続されているだろうという認識に基づくものです。
- コード・レビューは骨の折れる、比較的感謝されることの少ない仕事です。カーネル・コードを作成した人は記憶されますが、そのレビューを行った人の名声が長く続くことはほとんどありません。そのためレビューアは、特に同じ間違いを何度も繰り返された場合など、機嫌を悪くしてしまう場合があります。怒り、侮辱、あるいは明らかな攻撃に満ちたレビューを受けた場合は、同様の方法で応酬したいという衝動にかられるかもしれません。しかしじっと耐えてください。コード・レビューはあくまでコードに対するレビューであり、人に対するものではありません。またレビューアは個人的な攻撃を行っているわけでもありません。
- 同様に、コード・レビューアは、レビュー対象者を食い物にして彼らの企業方針を推進しようとしているわけでもありません。カーネル開発者は今後何年間もカーネル開発に取り組むことを期待している場合が多いですが、同様にその雇用主が変わってしまう可能性についても承知しています。彼らは本当のところ、ほとんど例外なく、可能な限り最善のカーネルを構築するために働いており、その企業の競争相手を苦しめようとしているわけではありません。

すなわち、これらのことを総合して言えば、レビューアからのコメントを受けた場合、その技術的な意見に注意を払う必要があるということです。彼らの表現スタイルや自分自身のプライドのために、このことがおろそかにならないようにして下さい。パッチに対するレビュー・コメントを受けたら、時間をかけてそのレビューアが言いたいことを理解してください。可能であれば、そのレビューアが望む修正を行ってください。それからレビューアに回答し、感謝を述べ、彼らの質問にどう対応するかについて説明してください。

なお、レビューアから提案された変更のすべてに同意する必要はありません。そのレビューアが自分のコードを誤解していると考えられる場合、実際に行っていることを説明します。提案された変更に対し技術的に賛成できない場合はその内容を説明し、問題に対する自分の解決策を正当化します。その説明が道理にかなったものであれば、レビューアはその内容を受け入れます。自分の説明に説得力がなく、また特に他の参加者もレビューアに賛成し始めた場合には、少し時間をおいてもう一度考え直してみてください。何かが基本的に間違っていることや、自分の解決策が真の問題を解決してないことに気が付かなくなることもあります。このように問題に対する自分の解決策に固執するあまり盲目になってしまうことはよくあることです。

決定的な間違いは、そのコメントがなくなることを期待してレビュー・コメントを無視することにあります。コメントが消えることはありません。以前に受けたコメントに対する回答を行わずに再びコードをポストした場合、おそらく何も先に進まなくなることがわかります。

コードの再ポストについて言えば、レビューアは前回ポストされたコードの内容をすべて記憶していることはないと考えておいてください。そのため、以前に出た問題をレビューアに思い出させ、それに対しどう対処したかを説明する方がいいでしょう。パッチの変更履歴はこの種の情報を示す良い場所になります。前回の議論の内容を思い出すためレビューアがメーリングリストのアーカイブを検索しなければならないような状況は好ましくありません。彼らが直ちにスタートできるよう準備しておけば、彼らは気分良くコードの再検討ができるようになります。

やるべきことはすべて実行したのに、依然として先に進まない場合はどうでしょうか？ほとんどの技術的な不一致は討論を重ねることで解決しますが、誰かが決定しなければならない時があります。その決定が自分にとって不当なものだと正直に信じられる場合、より上位に位置する権威者に訴えることができます。この文書の作成時点における上位の権威者とは、Andrew Morton のことです。Andrew はカーネル開発コミュニティの尊敬を集めており、ほとんど絶望的に行き詰まった状況を打開できる場合も多くあります。ただし Andrew への支援要請は軽々しく行われるべきではなく、あらゆる代替案の追求が終わるまで待つことが必要です。また、彼も自分の意見に同意してくれるとは限らないということを念頭に置いてください。

6.2: 次に何が起きるか？

そのパッチのカーネルへの追加が望ましいものと考えられ、またすべてのレビュー問題が解決した場合、次のステップは通常、メンテナーのツリーへの追加となります。その方法はサブシステム毎に異なり、メンテナーがそれぞれ自分自身の方法を用いています。特に、ツリーが二つ以上用意されている場合もあります。この場合、おそらくひとつは次回のマージ・ウィンドウに予定されるパッチ専用のツリーで、もうひとつは長期的な作業に用いるものとなるでしょう。

明確なサブシステム・ツリーの存在しない分野のパッチ（たとえばメモリー管理のパッチ）の場合、そのデフォルト・ツリーは最終的に -mm ツリーとなる場合が多くなります。複数のサブシステムに影響するパッチも最後は -mm ツリーに落ち着く場合があります。

サブシステム・ツリーに含めることにより、そのパッチの可視性が向上します。この段階では、そのツリーに関係する他の開発者はデフォルトでそのパッチを入手することになります。通常、各サブシステム・ツリーは -mm や linux-next にも接続されており、その内容が開発コミュニティ全体から見えるようになります。この時点で、再び別のレビューア集団から追加コメントを受ける可能性がかなりありますが、これらのコメントについても前回と同様に回答することが必要です。

この段階で更に発生する可能性があることは、そのパッチの性質により、他の開発者が行っている開発との競合が見えてくることです。最悪のケースとしてパッチの競合の度合いが非常に大きい場合、一部のパッチを棚上げとし、残りのパッチを統合して機能させるといった結果になることもあります。また別のケースとして、競合の解決のため別の開発者と共同で作業を行い、すべてが明確に適用できるよう、場合によっては一部のパッチをツリー間で移動する場合もあります。このような作業は苦痛ですが、それでも以前に比べれば恵まれています。すなわち、linux-next ツリーが実現する前はこのような競合が判明するのはマージ・ウィンドウ期間中であり、短い時間での素早い対応が必要でした。今ではマージ・ウィンドウが開くまでの間、時間的な余裕を持って解決にあたることができます。

すべてがうまく行けば、いずれは自分がログオンした時、自分のパッチがメインライン・カーネルにマージされていることを確認できます。おめでとうございます（MAINTAINERS ファイルに自分の名前が追加されます）。ところで、お祝いが終わったところで、小さなことですが、ひとつ重要なことを覚えておくことが必要です。つまり、仕事はまだ終わっていないということです。すなわち、メインラインにマージしたことによる課題が発生します。

まず、自分のパッチの可視性がさらに向上しました。このパッチにこれまで気付かなかった開発者達からのコメントが新たに寄

せられる可能性があります。自分のコードのマージについて心配する必要はもうないため、そのようなコメントは無視したいという誘惑に駆られるかもしれません。しかし、そのような誘惑は捨ててください。自分には質問や提案を寄せる開発者達に対し敏感に対応する責任があります。

しかし更に重要なことは、自分のコードがメインラインに含まれたことにより、そのコードがより大きなテスターの集団の手に渡ったということです。まだ利用可能になっていないハードウェア用のドライバを提供した場合でさえ、驚くほど多くの人がそのコードを自分のカーネルへとマージします。そして、当然のことですが、バグ報告も送られてきます。

バグ報告のうち、最悪のものはリグレッションの報告です。そのパッチでリグレッションが発生した場合、不快なほど多くの視線が自分に注がれることがわかります。リグレッションについてはできるだけ早急な修正が必要です。自分がその修正に不賛成、またはそれが不可能な場合（または誰も修正してくれない場合）、そのパッチはほぼ間違いなく安定化期間中に削除されます。パッチをメインラインにマージするために行ったこれまでの自分の努力がすべて否定されるだけでなく、リグレッションを解決するための修正ができずにパッチが削除されたという事実は、将来的に自分の成果物をマージしようとした場合それが困難なものとなってしまう可能性があります。

リグレッションへの対応が完了しても、それとは別に通常のバグ対応も必要となる場合があります。安定化期間は、これらのバグ修正を行い、メインライン・カーネル・リリースに向けての自分のコードのデビューを可能な限り確実なものにするための最適な機会です。そのため、バグ報告に回答するとともに、可能な限り問題の修正を行ってください。安定化期間はそのために設けられています。そのパッチへの対処が完了すれば、新たに素晴らしいパッチの作成を開始することができます。

また、次回のメインラインの安定リリースや有力なディストリビュータが自分のパッチを含むカーネルのバージョンを採用したときなど、新たなバグが報告される可能性もあります。このような場合のマイルストーンについても忘れないようにしてください。そのような報告への対応を継続することは、自分の仕事に対する基本的なプライドの問題です。もしそれだけでは動機付けが不足する場合、マージ後の自分のコードに対し関心を失ってしまうような開発者はそのように開発コミュニティに記憶されてしまうということを理解しておいてください。次にパッチをポストしたとき、彼らは作者によるマージ後の保守対応が行われないという前提でその評価を行うでしょう。

6.3: 起きる可能性のある出来事

ある日メール・クライアントを開くと、誰かが自分のコードに対するパッチを送付してきたことに気付くかもしれません。これは、結局のところ、自分のコードを公開したことによる利点のひとつです。そのパッチに同意した場合、自分でそのパッチをサブシステムのメンテナに転送するか（この場合、その帰属を示すための適切な「From:」行を追加し、自分のサインオフを追加します）、または「Acked-by:」応答を返し、そのポスト元にその先の提出を行わせることができます。

そのパッチに同意できない場合は、その理由を説明した礼儀正しい回答を送ります。可能であれば、そのパッチを自分が承認するためにはどのような変更が必要かをその作者に説明します。コードの作者およびメンテナーが反対するパッチをマージすることには一定の抵抗がありますが、それ以上ではありません。優れた成果を原作者が不必要に妨害していると見られた場合、そのようなパッチは最終的には先の過程へと流され、いずれはメインラインにマージされます。Linux カーネルにおいては、いかなるコードについても絶対的な拒否権を持つ人はいません。ただし、おそらく Linus は別でしょう。

非常に希なケースとして、上記にはあげなかったまったく異なった状況に遭遇する場合も考えられます。すなわち、自分が解決した問題に対し、別の開発者が別の解決策をポストした場合です。この時点で、おそらく二つのパッチのうちひとつはマージされなくなると考えられます。また「自分の方が先だった」という主張は、技術的には説得力がなく考慮されることはありません。自分のパッチが他人のパッチで置き換えられてメインラインにマージされた場合、それに対する反応はまさにひとつだけです。すなわち、その問題が解決されたことに満足し、自分の仕事を続けていくことです。このような形で自分の成果が脇に押しのけ

られてしまうのはフラストレーションであり、落胆もありますが、コミュニティは、誰のパッチが実際にマージされたかを忘れてしまった後でも、その作者の対応はずっと覚えていてくれます。

7: 上級の話題

この段階で、読者は開発プロセスの仕組みについて理解したものと期待されます。しかし、学習すべきことがまだあります。本セクションでは、Linux カーネル開発プロセスの常連になることを目指す開発者に役立つ、様々な話題について述べます。

7.1: git によるパッチ管理

Linux カーネルの分散型バージョン管理は 2002 年に開始され、この時に Linus は初めて独自の BitKeeper アプリケーションによるバージョン管理に取り組みました。BitKeeper については賛否両論がありますが、その中で具体化されたソフトウェア・バージョン管理のアプローチについては疑う余地のない確立されたシステムです。この分散型バージョン管理により、カーネル開発プロジェクトはさらに加速されました。現在は BitKeeper の代りになるフリーソフトウェアがいくつかあります。良くも悪くも、カーネル・プロジェクトはツールとして git を選択しました。

特にパッチの数が増すにつれ、git によるパッチ管理は開発者の負担を大幅に軽減します。git には荒削りな部分もあり、多少の危険性もありますが、若く力強いツールであり、現在もその開発者による改良が行われています。この文書は、読者に対し git の使用法を説明するようなことはしません。そのことを説明している資料はほかにたくさんあります。その代わり、ここでは git がカーネル開発プロセスにどうマージされているかということに特に焦点を当てます。git について十分に学習し使えるレベルに到達したい開発者向けには、以下のサイトに詳しい内容があります。

http://git.or.cz/

http://www.kernel.org/pub/software/scm/git/docs/user-manual.html

また、上記以外にも、Web 上には様々な教材があります。

git を使ってパッチを他の人に利用させようとする前に、最初に取り組むべきことは、これらのサイトの内容を良く読み、git の動作を確実に理解することです。git を使う開発者は、メインライン・レポジトリのコピーの取得、リビジョン履歴の調査、ツリーへの変更の追加、ブランチの利用等ができるはずです。また、履歴の書き直しのための git ツールを理解することも有用です（rebase など）。git にはその特有の用語や概念があります。git を初めて使うユーザーは ref、remote branch、index、fast-forward merge、push や pull、detached head といった内容について理解することが必要です。これらはいずれも最初は少し困難さを感じますが、少し学習すれば、その概念を把握することはそれほど難しくありません。

git を使ってパッチを作成しメールで提出することは、git を使いこなせるようになるための良い練習になります。

他から見ることのできる git ツリーを立ち上げる準備ができたら、当然ながらその内容をプルするためのサーバが必要となります。インターネットにアクセス可能なシステムがあれば、そのようなサーバに git-daemon をセットアップするのは比較的簡単な作業です。さもなければ、フリーのパブリック・ホスティング・サイト（たとえば Github）がネット上でサービスを提供しております。また既に経験のある開発者は kernel.org にアカウントを得ることができますが、そのような開発者になるのは簡単なことではありません。詳細については http://kernel.org/faq/を参照してください。

通常の git ワークフローには多くのブランチが使用されます。各開発ラインは独立したトピックブランチ「topic branch」に分けることができ、独立した管理が可能です。git におけるブランチは簡単に作れます。これらを自由に使わない理由はありません。また、いかなる場合であっても、外からプルさせようとするブランチで自分の開発を行うべきではありません。一般に公開するブランチは慎重に作成することが必要です。開発ブランチからのパッチのマージは、そのパッチが十分に完成して準備が整うまでは行わないでください。

git には、自分の開発履歴を書き直すことができる、強力なツールがあります。不都合なパッチ（bisection を破壊するものや、その他の明白なバグのあるものなど）は所定の場所に固定したり、あるいは履歴から完全に消去したりすることができます。何ヶ月も作業を続けてきたパッチ群であっても、あたかも現在のメインライン上に書かれてきたかのようにパッチ群を書き直すことができます。変更をひとつのブランチから別のブランチへと透過的に移すことができます。その他、多くの機能があります。履歴を改訂できる git の能力を賢く利用することで、問題の少ないクリーンなパッチ群が作成できます。

ただし、この機能を過度に利用すると、完全なプロジェクト履歴を作成したいという単純な要求以外の別の問題が発生する可能性があります。履歴の書き直しは、その履歴に含まれる変更の書き直しとなり、テスト済み（と期待される）カーネル・ツリーが未テストのツリーに変わってしまいます。しかし、それ以上に、プロジェクト履歴の表示が共有できない場合、開発者間の連携が困難になります。たとえば、他の開発者がそのレポジトリにプルした履歴を自分が勝手に書き直してしまった場合、その開発者にとって面倒なことになります。そのため、ここでは、「外部にエクスポートされた履歴は原則としてそれ以降、不変とする」という簡単な経験則が適用されます。

したがって、その変更セットを自分の公開サーバにプッシュした後は、その変更について書き直しを行ってはなりません。もし「fast-forward merge」が行えないような変更（同じ履歴を共有していない変更）をプッシュしようとしても、git はこのルールを適用されます。このチェックを無効にすることは可能で、またエクスポート済みツリーの書き直しが必要となる場合もあると考えられます。その 1 例として、linux-next での競合を避けるためツリー間で変更セットを移動することもあります。しかし、そのようなことは稀です。これは、開発は（必要に応じ書き直しが可能な）プライベート・ブランチで行い、十分に熟成した状態になってからパブリック・ブランチに移動させるべきだということのひとつの理由です。

メインライン（または変更のベースとなるその他のツリー）が進化するにつれ、その最先端のツリーを追いかけるためにこのツリーにマージしたいと思うでしょう。プライベート・ブランチの場合には別のツリーに遅れないようレベルを合わせていくためのベースの変更は容易ですが、外部の世界に対しツリーをエクスポートしてしまった後は、ベースを変更するという選択肢はありません。一度これを行なってしまった後は、全体の完全なマージを行わなければなりません。時折マージを行うことはたいへん意味のあることですが、過度な頻度でマージを行うと、履歴が不必要に乱雑になってしまいます。この場合に提案される手法はマージの頻度を減らすことで、一般には特定のリリース・ポイント（メインラインの -rc リリースなど）に限定することです。特定の変更が気になる場合、プライベート・ブランチでのテスト・マージはいつでも可能です。git の「rerere」ツールはそのような場合に有効です。このツールはマージの競合がどのように解決されたかを記憶しているため、同じ作業を 2 回繰り返す必要はありません。

何回もむしかえされる git のようなツールに関する不満は、パッチがひとつのレポジトリから別のレポジトリへと大量に移動することにより、不具合のある変更も紛れ込み、レビューのレーダー監視網を逃れてメインラインに入り込むというものです。カーネル開発者は、このような事が発生すると悲しい思いをします。レビューされていないパッチや話題からずれたパッチが含まれたツリーを立ち上げてしまうと、将来的には自分のツリーをプルしてもらえなくなってきます。Linus の言葉を引用します。

> **皆さんは私にパッチを送ることができますが、私が git パッチをプルするためには、私はそのツリーで何が行われているかを知る必要があります。その時、個々の変更を自らの手でチェックすることなく、そのことを信用できることが必要です。**
>
> （http://lwn.net/Articles/224135/ ）

このような状況を避けるため、そのブランチに含まれるすべてのパッチがその関係するトピックに密接に関係してい

るようにすることが必要です。たとえば「driver fixes」というブランチがコアメモリー管理用のコードの変更を行うようなことがあってはなりません。また、最もやってはいけないことはなことは、レビュー・プロセスをバイパスするために git ツリーを使わないということです。時折、関連するメーリングリストにツリーのサマリーをポストし、適切なタイミングでそのツリーを linux-next に含めるよう依頼しましょう。

他から自分のツリーに含めるようパッチが送られてくるようになった場合、それらパッチのレビューを忘れないようにします。また、正しい作者情報を維持するようにします。git の「am」ツールはこの点で最適ですが、サードパーティ経由で送られてきた場合、自分でそのパッチに「From:」行を追加することが必要となる場合もあります。

プルを依頼する場合、そのツリーのある場所、プルすべきブランチ、またそのプルによりどのような変更が行われるかなど、関係する情報をもれなく提供するようにしてください。git の「request-pull」コマンドはこの点で役に立ちます。このツールは他の開発者の期待するフォーマット化を行うと共に、パブリック・サーバにそれらの変更をプッシュしたことが記憶されていることも確認します。

7.2: パッチのレビュー

一部の読者は、初心者のカーネル開発者でもパッチのレビューを行うはずだということから、この項目を「高度なトピック題」に含めることには反対でしょう。たしかに、他人がポストしたコードをレビューする以外にカーネル環境におけるプログラミングの方法を学習する良い方法はない、ということは言えます。それに加え、レビューアは常に不足しています。コードを良く見ることで、全体としてのプロセスに大きく貢献することができます。

コードのレビューにはプレッシャーがかかりがちです。特に、新たにカーネルの開発を行う人は、より経験の深い開発者がポストしたコードについてオープンな場で質問することには緊張感を感じるでしょう。しかし、最も経験の深い開発者が書いたコードであっても、その改良が可能な場合はあります。おそらく、レビューア（すべてのレビューア）に対する最も適切なアドバイスは、レビューのコメントを「批判」ではなく「質問」の形で表現することです。「このロックは、このパスでどのように開放されるのでしょうか？」という質問は、「ここのロックは間違っている」という言い方よりも常に良い結果が出ます。

様々な開発者が、それぞれ別の観点からレビューを行います。コーディング・スタイルや、コード行の最後にスペースが入っているかどうか、といったことを主として気にする人もいます。また、そのパッチの変更がカーネルにとって全体的に良いことか悪いことかに重点を置く人もいます。またその他にも、問題の多いロックや過度のスタックの利用、セキュリティ問題の可能性、他の部分とのコードの重複、ドキュメンテーションの適切さ、性能に対する悪影響、ユーザー空間 ABI の変更といった内容をチェックする人もいます。そのレビューが優れたコードをカーネルにマージすることにつながるなら、すべて歓迎される価値のあるレビューです。

8: 詳しい情報

Linux カーネルの開発やその関連トピックについての情報源は非常に多くあります。これらのうち、まず一番に読むべきはカーネル・ソースのディストリビューションに含まれる documentation ディレクトリです。最上位レベルの「HOWTO」ファイルは重要なスタート・ポイントで、また SubmittingPatches と SubmittingDrivers もすべてのカーネル開発者が読むべき内容です。多くの内部 API は kerneldoc のメカニズムで文書化されています。またこれらのドキュメントを HTML や PDF のフォーマットで作成する場合、それぞれ「make htmldocs」と「make pdfdocs」が使えます（ただし、一部のディストリビューションで配布されている TeX のバージョンは内部の制限にかかり、ドキュメントの処理が適切に行われません）。

カーネル開発については、様々な Web サイトで、あらゆるレベルの詳しさで議論されています。筆者としては、http://lwn.net/ をひとつのソースとして謙虚に推薦したいと思います。また多くの具体的なカーネルの話題に関する情報は、以下のサイトにある LWN kernel index で見つけることができます。

http://lwn.net/Kernel/Index/

さらに、カーネル開発者にとって価値のあるリソースが以下の場所にあります。

http://kernelnewbies.org/

linux-next ツリーに関する情報は以下のサイトに集められています。

http://linux.f-seidel.de/linux-next/pmwiki/

また、当然ながら、カーネルのリリース情報について最も信頼できる http://kernel.org/ を忘れるわけにはいきません。

カーネル開発に関する本は以下のものを含め多数あります。

Linux Device Drivers, 3rd Edition (Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman)、オンライン公開中（http://lwn.net/Kernel/LDD3/ ）

Linux Kernel Development (Robert Love)

Understanding the Linux Kernel (Danial Bovet and Marco Cesati)

ただし、これらの本には共通の問題があります。それは、書店の店頭に出る頃には既に内容が多少古くなってしまう傾向があるということで、また既に発行からある程度の期間が経過しています。とは言うものの、これらの本には価値のある情報が多く含まれています。

git のドキュメンテーションは以下のサイトにあります。

http://www.kernel.org/pub/software/scm/git/docs/

http://www.kernel.org/pub/software/scm/git/docs/user-manual.html

9: 結論

この、長いドキュメントを最後まで読んでいただいた読者の皆様を祝福させていただきます。おそらく、Linux カーネルの開発方法や、そのプロセスに参加する方法についての理解に役立てていただけることと思います。

結局のところ、参加することが大事です。オープンソース・ソフトウェアのプロジェクトはいずれも、そのコントリビュータが提供したものの集合でしかありません。Linux カーネルはこれまでと同様に迅速かつ順調な進歩を遂げてきましたが、これはいずれも Linux カーネルの改善を目指す開発者達からなる、感動的なほど大きなグループに助けられたおかげです。このカーネルは、数千人もの人々が共通の目標に向かって共同作業を行ったときに達成可能なことの最も優れた成功例です。

しかし、開発者基盤が大きいほど、カーネルの受ける利益も大きくなります。やるべきことは常にあります。しかし、同様に重要なことは、Linux エコシステムへの参加者のほとんどは、それぞれカーネルへの貢献を通じて自分自身の利益が得られるということです。コードの品質向上、保守／配布費用の低減、カーネル開発の方向性に対する高いレベルの影響を与えることなど、いずれもコードをメインラインにマージすることが重要なキーになります。これにより、関係者の全員が勝利を得られます。さあ、皆さんのエディターを立ち上げ、参加してください。大歓迎です。