第 6 回 東京エリア Debian 勉強会
事前資料*

Debian 勉強会会場係 上川純一†

2005 年 7 月 2 日

* 機密レベル public: 一般開示可能

† Debian Project Official Developer

目次

1 Introduction To Debian 勉強会

上川純一

今月の Debian 勉強会へようこそ．これから Debian のあやしい世界に入るという方も，すでにどっぷりとつかっているという方も，月に一回 Debian について語りませんか？

目的として下記の二つを考えています．

- メールではよみとれない，もしくはよみとってられないような情報を情報共有する場をつくる
- まとまっていない Debian を利用する際の情報をまとめて，ある程度の塊として出してみる

また，東京には Linux の勉強会はたくさんありますので，Debian に限定した勉強会にします．Linux の基本的な利用方法などが知りたい方は，他でがんばってください．Debian の勉強会ということで究極的には参加者全員が Debian Package をがりがりと作りながらスーパーハッカーになれるような姿を妄想しています．

Debian をこれからどうするという能動的な展開への土台としての空間を提供し，情報の共有をしたい，というのが目的です．次回は違うこと言ってるかもしれませんが，御容赦を．

1.1 講師紹介

- gotom さん Debian の glibc とかをやってます．
- 上川純一 宴会の幹事です．Debian Developer です．元超並列計算機やっていて，今は音楽関係とか，気づいたら canna とか．あと，pbuilder や，libpkg-guide などを通して，Debian の品質向上を目指してます．

1.2 事前課題紹介

今回の事前課題は「いままで toolchain をつかってみて/今後 toolchain に期待すること」というタイトルで 200-800 文字程度の文章を書いてください．というものでした．その課題に対して下記の内容を提出いただきました．

1.2.1 makoyuki さん

えー、すいません、当方、そもそも「toolchain」といわれて何のことかわかりませんでした。大体「開発環境一式」とかかなぁ?という感じだったのですが、正確に理解しているかわかりません。

んで、仮にそうだとして、ある程度のコンパイルとかはしていても、当方コード書きではまったく無い上、少なくとも Debian においては一度も手動でソースコードからコンパイルしたことがありませんし、プログラミング周りのことについてはド素人なのでド素人なりのことを書いてみます。

開発環境であるという前提で書きますが…正直に包み隠さず言いますと、当方は実際のところプログラミン

グっぽいことがしたくないから Debian を使っています。コードは書きたくないのです。万が一それが楽しいと思うことがあれば今後絶対にやらないということはありませんが、少なくとも今、私がやりたいのはプログラミングではありません。なので、「toolchain を使う」ということは、当面ないと思うのです。いままでも。そしてこれから当分の間は。

勿論、勉強にもなりますし、理解を深める一助にもなるでしょうから、それについての勉強をしたいとは思います。その上で、限度はあるにせよ、ドキュメントの充実、ツールやインターフェースの充実により、レベルは下げずに敷居を下げるものであって欲しいという浅はかな望みはありますが、それはなんとなく今回のお題にはそぐわない気もします。

そもそもが toolchain が何のことか理解していないかもしれませんので完全にハズしているかもしれませんがご勘弁を。

1.2.2 **澤田さん**

toolchain に期待すること、ということでいろいろ妄想してみた。

まず現実的なもの。最適化は結構ブラックボックスなイメージがあるので、最適化を行ったときにどの部分にどんな最適化を施したのか、逆にここはなんで最適化をかけなかったのか（時間と空間のトレードオフとか)、最適化をかけることでどんな効果が得られるのかのログがあるといいなと思った。

もうひとつまあまあ現実的なもの。リンク時にどのライブラリをリンクしないといけないのか考えるのが面倒なので、ライブラリが export しているシンボルを保存しておき、リンク時に参照する機能があるといいなと思った。この場合、どのライブラリを使っているかわからなくなってしまうので、デフォルトオフで–library-auto- detect とかいうオプションがあるといいだろうか。

次に実現できるかよくわからないもの。コンパイル時には inline 展開なるものがあるわけだが、リンク時（共有ライブラリ使ってる場合はロード時）にも inline 展開してくれるとうれしいことがあるかもしれないと思った。

最後にかなり非現実的なもの。最適化を行う際に一番問題なのはやはりどの最適化オプションを使えばいいのかわからないということであろう。これを解決する方法として、こんなコードに対してはこんな最適化が有効だったというデータベースを構築し、最適化時にそれを参照、実行してデータベースに反映とかすると何かができるのではないかと思った。

1.2.3 KATO **さん**

Toolchain = glibc/binutils/gcc ということで, glibc はもちろん, gcc, binutils を普通に使ってきましたが今まで特に「toolchain」という形で特に意識したことはありませんでした. ですから「使ってみて」といって実は何を書いたらよいのか良くわかりません. Toolchain の構成要素はそれぞれ互いに密接に関連していて, うまく連携がとれている必要があると思うのですが, 今後に期待するのは, (興味のない人は)toolchain のことを特に意識しないでも安定して Debian を使い続けられるようにきちんと連携たとれた状態にメンテナンスされているように, ということです.

1.2.4 **中島　清貴さん**

なかなかとても良い。このような使いやすいシステムがあったとは今まで全然知らなかった。ぜひわたしの友人、知人その他の世話になった方々に使ってみてもらいたい。と心底思います。toolchain を知って本当に良かったです。もし、まだ使っていないのなら、すぐに使うべきです。断言します。とてもおすすめです。こ

れは絶対に使うべきです。私がこのようなシステムを使えるということは、なによりの喜びであり使ってみて本当に感激いたしました。これは使わなければ大損です。

1.2.5 Hidetaka Iwai さん

今回のお題は toolchain とのことですが、個人的には toolchain に関して特に要望等はありません。debuan 2003 年夏号に顛末が書いてあった VIA C3 での libssl 動作問題のように、glibc の upstream maintainer 達や RedHat 等他の distribution の人々と協力関係を保って今後もメンテされていけば素晴しいのではないかと思います。強いて言えば、toolchain 単独の問題ではないのですが、auto builder や pbuilder, dpkg-cross と協調して package を cross compile できるようになれば、速くて安価な x86 マシンを buildd として投入できて、遅い buildd しかない architecture (m68k 等) の security update 等が速くなって皆幸せになれるような気がします。

1.2.6 Nobuhiro Iwamatsu さん

1. linux のカーネルヘッダが必要なのはどうかと思う。(BSD とかはどうやってるんだろう？)
2. toolchain は アーキテクチャによって gcc と glibc , binutils のバージョンの組み合わせで動いたり動かなかったりするのが問題と思う。しかし、Debian ではサポートされているアーキテクチャで全て同じ組み合わせというのがすばらしい。
3. コンパイラがコンパイルできても動くとは限らないのが問題だと思う。Debian でも毎回テストユニットを使ってテストしてから upload してるんですか？
4. SuperH サポートしてください。

1.2.7 上川

etch にむけて g++ の transition があるらしい．ABI がまた変わるとのこと．gcc 3.0 になるときに「これで最後だから」といってみておきながら，また浮気するのですか？あぁ...

woody から sarge にあがる際に，c++ の ABI が一旦変更になっています．アップグレードの際にこまったりしませんでしたか？

2 Debian Weekly News trivia quiz

上川純一

ところで，Debian Weekly News (DWN) は読んでいますか？Debian 界隈でおきていることについて書いている Debian Weekly News. 毎回読んでいるといろいろと分かって来ますが，一人で読んでいても，解説が少ないので，意味がわからないところもあるかも知れません．みんなで DWN を読んでみましょう．

漫然と読むだけではおもしろくないので，DWN の記事から出題した以下の質問にこたえてみてください．後で内容は解説します．

2.1 2005 年 24 号

6 月 14 日版です．

問題 1. Alioth が新しいサーバに移動するということが各ユーザ宛のダイレクトメールで出た．Branden Robinson が Alioth のアナウンスメールについてコメントを出した．その主旨は．

□ A 匿名で各ユーザにメールを出すのではなく，debian-devel-announce に出そう，さらに誰が出したのかを明確にして出そう

□ B メールは信頼できないプロトコルなので WEB ページに掲示したらよいだろう

□ C SMTP より，SNMP トラップのほうがみんな気づく

問題 2. Andreas Barth は etch に向けてリリースポリシーについてどういう変更が必要だと説明したか

□ A ダメな人禁止

□ B ダメなコード禁止

□ C 共有ライブラリはダイナミックにリンクすべきであり，ソースコードをコピーするのは禁止しよう．

問題 3. etch にむけて c++ の ABI が変更になるため，Matthias Klose は何を宣言したか

□ A C++ でかいたプログラムの粛正

□ B とりあえず今 C++ のライブラリであたらしい ABI バージョンのものについてはアップロードしないようにフリーズする

□ C java の利用禁止

問題 4. Debconf での講演の予定の組み方について今回はどういうこころみがなされたか

□ A 意志決定方法としてさいころを導入してみた

□ B 参加者の都合が悪い日をできるだけ選んだ

□ C 参加したい講演の投票をおこない，参加者ができるだけ参加できるようにスケジューリングをする

問題 5. Scott James Remnant 曰く，dpkg 1.13 で新しく追加されたアーキテクチャ変数名は
□ A DPKG_HOST_ARCH_OS
□ B DEBIAN-ARCHITECTURE
□ C uname -a

問題 6. Roberto Sanchez の作成した FAQ には何がかいてあるか
□ A Debian パッケージをインストールする方法がかいてある
□ B Debian パッケージをカスタマイズしてビルドする手順がかいてある
□ C Debian メンテナを招集する方法がかいてある

問題 7. selinux の進捗についてはどうなっているか
□ A coreutils などのパッケージが libselinux1 に依存するようになってきた
□ B NSA の陰謀であることが判明したので排除
□ C もうほとんどのユーザが SELINUX を使うようになった

2.2 2005 年 25 号

6 月 21 日版です.
問題 8. Oskuro が GNOME について宣言したのは何か
□ A 今後は KDE に移行する
□ B Gnome 2.10 が unstable に入った
□ C GNOME3 がもうすぐリリース

問題 9. woody から sarge にアップグレードするのにもっとも大きい問題は何か
□ A 相互に依存するような依存関係があるパッケージのアップグレード
□ B woody では存在したけど sarge で消えたパッケージの処理
□ C etch に無いパッケージ

問題 10. Aurelien Jarno が BSD 移植版について報告した問題は
□ A GNU Hurd についてはもうあきらめたので考えなくて良い
□ B FreeBSD についてはもうあきらめた.
□ C libselinux1 に依存するパッケージは，selinux が linux のみなので，Linux のみでリンクするようにしてほしい

問題 11. Bill Allombert が menu について報告したのは
□ A menu のセクションの部分については国際化できるようになった
□ B menu はもう使われていないので廃止する
□ C 対応する Window Manager が増えた

問題 12. 矢吹さん曰く大阪でなにかあるらしい，何？
□ A 10 月に mini Debian Conference を開催する
□ B 9 月にたこやきはやぐい競争
□ C 阪神優勝祈念会

2.3 2005 年 26 号

6 月 28 日版です.

問題 13. Debian Policy Committee を Branden Robinson が発表しました．チェアマンは誰？
□ A Manoj Srivastava
□ B Andreas Barth
□ C Matt Zimmerman

問題 14. Andreas Barth は etch 向けのリリースポリシーを発表しました．そのなかにあった記述は
□ A debian/changelog や debian/control ファイルを通常のビルド処理で変更することを禁止する
□ B 二年以内にはリリースしない
□ C パッケージの数は 10 万を越えることを目標にする

問題 15. XML 形式のインタプリタ言語 CYBOP を Debian に含める場合，プログラムはどこにおいたらよい？
□ A ユーザが実行できるなら/usr/bin 以下，ライブラリモジュールなどなら/usr/share/cybop のような場所を指定する
□ B /srv
□ C XML ベースの言語なんてとんでもない

問題 16. cogito というプログラムと git というプログラムが，/usr/bin/git を両方保持していて，conflict: していたことに関しての反論は
□ A /usr/bin/git の機能が全く違う
□ B /usr/bin/git は GNU の商標です
□ C cogito の/usr/bin/git が一番よく使われているから GIT の git なんか不要

問題 17. openafs パッケージは全アーキテクチャではビルドできない．しかし，Architecture: all のパッケージがあるために，各アーキテクチャでビルドが試行されてしまう．そのような状況を表現する最良の方法は何？
□ A Build-Depends: type-handling でアーキテクチャを指定
□ B Packages-arch-specific を使う
□ C buildd のメンテナに個人的にメールを送る

問題 18. tetex がすでにあるのに TeXlive をパッケージする理由としては

□ A たくさんの TeX パッケージがあり，年に一回リリースしている

□ B tetex は使えない

□ C そこに TeXlive があるから

3 最近の Debian 関連のミーティング報告

上川純一

3.1 東京エリア Debian 勉強会 5 回目報告

前回開催した第 5 回目の勉強会の報告をします.

DWN クイズを実施しました．今回は 3 週間分だったので，分量がすくなかったかな.

えとーさんが update-alternatives と dsys について説明しました．update-alternatives が 実は root 権限がなくても使えるようになっていたり，新しい発見がありました．dsys については，alternatives の Description(説明文) というものがないのでその情報が欲しい，さらに国際化するには何が必要だろう，という話題が出ました.

武藤さんが debian-installer の構造について話をしました．国際化はまだ etch にむけて改善できるだろう，ということです．udeb をつかっていたり，anna をつかっていたり，cdebconf を使っているところとか．1st-stage と 2nd-stage があって，全然違うとか．インストーラが動いているときには DEBCONF_PRIORITY は HIGH なのにインストール後は MEDIUM に強制変更するためいろいろとひずみがでているとか．なお，自動認識されないハードウェア情報については，バグレポートが重要なので，lspci, lspci -n の出力とともにおくってくださいとのこと．lspci -n の出力はしらなかったけど便利.

GPGKeysigning party をしました．キーサイニングに参加するのは 10 人なので，あまり時間かからないかなぁ，と思ったら 30 分くらいかかってしまいました.

はなの舞で宴会，デニーズでデザート.

4 Debian ツールチェインと glibc, linux-kernel-headers パッケージ

後藤正徳 [*1]

4.1 はじめに

先ほどリリースされた Debian sarge では 11 ものアーキテクチャをサポートしている。これら異なるアーキテクチャが同時に使い物になるためには、当然各アーキテクチャ用バイナリを生成するツールチェインも十分整備されていなければならない。

本文書では、ツールチェインとは何かを簡単に紹介した後、筆者がメンテナンスしている glibc, linux-kernel-headers パッケージに関して説明を行う。

4.1.1 ツールチェインとは

ツールチェイン (toolchain) という用語に正確な定義があるわけではないが、一般にマシンネイティブなバイナリを生成・加工するための一連のツール群を指すことが多い。本節では、まず Debian での代表的なツールチェインパッケージを紹介する。

4.1.2 gcc

GNU Compiler Collection の略称。ツールチェインのコアパッケージであり、ソースコードからアーキテクチャ毎のバイナリへ変換する役割を持つ。なお、gcc 自体はソースコードからアセンブラコードを出力するまでの機能しか持っておらず、そこから先のバイナリ生成部分は後述する binutils のツールを内部で呼び出している。現在サポートするプログラム言語としては C (gcc), C++ (g++), Java (gcj), Fortran (g77, g90) などがある。

Debian パッケージメンテナは Debian-gcc チーム (Matthias Klose, Gerhard Tonn) であるが、事実上 Matthias (doko) 一人がメンテナンスを担っている。

4.1.3 binutils

binutils には、バイナリを生成・操作するツール群が入っている。例えば、gcc によって出力されたアセンブラコードをアセンブルしてオブジェクトコード化するアセンブラ as、複数のオブジェクトコードをリンクしてバイナリを生成するリンカ ld、オブジェクトコードを逆アセンブルしたり解析したりするツール objdump などがある。

Debian パッケージメンテナは James Troup である。最近では C++ transition for etch に絡んで Matthias Klose が主な実作業にあたっている。また、binutils の上流開発者の中で Debian 開発者も兼任している人物に Daniel Jacobowitz がいる。

[*1] GOTO Masanori, gotom@debian.org, Debian Project, 2005-07-02

4.1.4 glibc, linux-kernel-headers

glibc は GNU C Library の略称。gcc, binutils がバイナリを生成するものであるのに対し、glibc は生成したバイナリ実行時に使用される C 言語ライブラリである。C 言語の関数やバイナリ実行時に最初に必要となる初期実行コードなどを含んでいる。なお、C ライブラリは実行環境に依存するため、システムによっては glibc 以外の libc が使用されることもある[*2]。

linux-kernel-headers は glibc のうち /usr/include/linux など Linux カーネルソース起源のヘッダを扱うパッケージである。元々 libc6-dev パッケージにマージされていたが、ソースが別々であることから 2 つに分離された。

4.1.5 gdb

gdb はバイナリ実行時に、ユーザからデバッグを可能とするための機能を提供する。

Debian パッケージメンテナは上流開発者でもある Daniel Jacobowitz である。

4.1.6 **その他**

その他のものとして C++ 向けライブラリである libstdc++ などがある。また、bfd といったバックエンドライブラリや dejagnu といったツール群が存在しているが、本文書では割愛させて頂く。

4.2 glibc, linux-kernel-headers **パッケージの概要**

4.2.1 **歴史**

過去をひもとくと、元々 Linux の C ライブラリとしては H. J. Lu を中心に開発されていた libc4, libc5 が使用されていた。これは GNU C Library バージョン 1 をベースに Linux 向け改良が追加されたものであった。しかし、GNU C Library 自体も Roland McGrath, Ulrich Drepper, Andreas Jaeger を中心に別途開発が続けられており、古い libc5 に代わってより新しい現在の glibc バージョン 2 が libc6 として置き換わる移行が行われた (libc6 transition)。

Debian における glibc パッケージは、当初 Joel Klecker がメンテナンスしていた (1999〜2000)。しかし Joel が病気で逝去したため、代って Ben Collins がメンテナンスを開始した (2000〜2002)。だが、やがて Ben 自身の興味が薄れて放置されるようになってきたため、現在のメンバから構成されるメンテナンスチームが組まれた (2002〜)。その後 Jeff Bailey, Branden Robinson, Daniel Jacobowitz を中心に古い Debian パッケージから debhelper ベースへ完全に書き直され、同時に linux-kernel-headers パッケージが libc6-dev パッケージから分離した (2003)。現在は筆者を中心にメンテナンスが続けられている。

4.2.2 **パッケージメンテナ**

パッケージメンテナは Debian-glibc チーム。構成メンバは以下の通り。

- Ben Collins (benc) … 元 Debian Project Leader。Debian/SPARC ポートメンバ。svn 開発者、Linux ieee1394 カーネルサブシステムメンテナ。
- GOTO Masanori (gotom) … 筆者。glibc 開発者、Linux SCSI カーネルドライバメンテナ。

[*2] 例えば組み込み環境では GNU newlib、Debian netbsd-i386 では、NetBSD 用 libc など。

- Philip Blundell (pb) … Debian/ARM ポートメンバ。glibc を含む上流 ARM ツールチェインや Linux/ARM カーネルのメンテナ。
- Jeff Bailey (jbailey) … Debian/GNU hurd-i386 ポートメンバ。現在 Ubuntu にて ppc64 など先進的なツールチェインメンテナンスを行っている。
- Daniel Jacobowitz (drow) … Debian/PowerPC ポートメンバ。glibc, gdb, gcc, binutils などをフルタイムでハック。最近の glibc まわりでは MIPS TLS や ARM new EABI などをアクティブに作業している。

この他にも以下のメンバが活躍している。移植周り: Bastian Blank、debian-installer 関連: Colin Watson、locale: Petter Reinholdtsen, Denis Barbier、MIPS: Thiemo Seufer, Guido Guenther、hppa: Carlos O'Donell, amd64: Andreas Jochens, Goswin von Brederlow, ia64: David Mosberger, Randolph Chung, s390: Gerhard Tonn。

4.2.3 開発体制

開発は主に debian-glibc@lists.debian.org メーリングリストを中心に行われている。また、時々 IRC にて議論を行って今後の方針などを決定している。パッケージ管理は元々 cvs で行っていたが、現在は alioth の svn ベースに切り替わっている。レポジトリは以下の通り。

```
svn://svn.debian.org/svn/pkg-glibc/glibc-package/trunk
svn://svn.debian.org/svn/pkg-glibc/linux-kernel-headers/trunk
```

4.3 glibc バイナリパッケージとその中身

本節では、glibc の各種バイナリパッケージとそれに含まれるファイルを紹介していく。各ファイルの解説によって、glibc がどのような機能を提供しているかの一端が明らかになるだろう。

ところで、パッケージの中には同じ機能を提供しているにも関わらずアーキテクチャによってついている名前が異なることがある。具体的に i386 での名前を挙げると libc6, libc6-dev, libc6-dbg, libc6-pic, libc6-prof, libc6-udeb がそうである。libc6 を例にアーキテクチャとパッケージ名の対応を表 1 に挙げる。以後では、一番親しんでいるであろう i386 での名前を使用して説明を進める。

パッケージ名	アーキテクチャ
libc6	amd64, arm, i386, m68k, mips, mipsel, powerpc, sparc, s390, hppa, sh3, sh4, sh3eb, sh4eb
libc6.1	alpha, ia64
libc0.3	hurd-i386
libc1	freebsd-i386

表 1　呼称が変化するパッケージ名とアーキテクチャとの対応関係

4.3.1 libc6

C ライブラリとダイナミックローダ等を提供。Priority: required, Section: base。

sid/i386 では約 9100 バイナリパッケージが依存している Debian のコアパッケージ。本パッケージは以下のファイルを含む。

/lib/*.so	libc のコアとなる動的ライブラリ[*3]。動的ローダ (i386 では /lib/ld-linux.so.2) と libc (i386 では libc.so.6)、そして周辺ライブラリ (libm, libnss_*, ...) を含む。スレッドシステムにはカーネル 2.4 でも動作する linuxthreads を採用。
/lib/tls/*.so	/lib/*.so と同様だが、TLS (Thread Local Storage) が有効になり、デフォルトのスレッドシステムが規格により正しく準拠した NPTL (Native Posix Thread Library) になっている。カーネル 2.6 使用時には、デフォルトの /lib/*.so 動的ライブラリに代わってロードされる。
/lib/*.so.*, /lib/tls/*.so.*	各ライブラリのバージョン番号を指し示すためのシンボリックリンクファイル。
/usr/lib/gconv/*	gconv (各文字コード・エンコーディング毎に用意されているコード変換 iconv モジュール) 動的ライブラリと gconv-modules (エイリアスファイル)。
/usr/lib/*	pt_chown をインストール。
/usr/share/zoneinfo/*	コンパイル済タイムゾーンデータ。環境変数 TZ に指定されたタイムゾーンに応じた時刻を返すために使用される。
/usr/bin/	各種設定・変換・表示ツール (iconv, locale, localedef, getent, getconf, catchsegv, tzselect, ldd, lddlibc4, zdump, rpcinfo) をインストール。
/usr/sbin/*	設定ツール (zic, iconvconfig, tzconfig) をインストール。
/sbin/*	動的ライブラリキャッシュファイル作成ツール ldconfig をインストール。
/sys	sysfs のためのマウントポイント。歴史的事情により存在。

4.3.2 libc6-sparc64, libc6-s390x

64 ビット biarch[*4]向けに作成されている libc6 パッケージ。Priority: standard, Section: base。

Debian sarge では sparc64, s390x にのみ提供されている。基本的にパッケージの中身は libc6 パッケージと同等であるが、64 ビット化した動的ライブラリを /lib の代わりに /lib64, /usr/lib64 配下へインストールする。

4.3.3 libc6-i686, libc6-sparcv9, libc6-sparcv9b

アーキテクチャ依存で提供される最適化パッケージ。Priority: extra, Section: base。

opt パッケージ、hwcap パッケージとも呼ばれる。基本的にパッケージの中身は libc6 パッケージと同等であるが、例えば libc6-i686 では /lib のかわりに /lib/tls/i686/cmov というディレクトリの下に最適化さ

[*3] 動的ライブラリ (dynamic linked library) とは .so で終わる名前を持ち、バイナリ実行時に動的ローダ (dynamic loader) によって必要に応じてロードされるライブラリファイル。これに対し、静的ライブラリ (static linked library) とは .a で終わる名前を持ち、コンパイル時にバイナリへライブラリコードを直接埋め込んでしまうために用いられるファイル。

[*4] bi: 2 つの、arch: アーキテクチャ。biarch とは、2 つのアーキテクチャを同時に使用可能なこと。

れた動的ライブラリがインストールされる。この最適化動的ライブラリは、アーキテクチャが i686 以上かつカーネルが 2.6 以上でプロセッサに cmov 拡張がある[*5]場合、バイナリ実行時に自動的に使用される。なお libc6-i686 は 686 用に最適化しているだけにとどまらず、スレッドサポートが大きく改善されているという重要な特徴がある。

4.3.4 libc6-dev

開発向けパッケージ。Priority: standard, Section: libdevel, Build-Essential。

C, C++ での開発を行うためには必ず必要になる各種ファイルがインストールされる。本パッケージは以下のファイルを含む。

/usr/include/* ヘッダファイル一式を格納。

/usr/bin/* gencat, mtrace, rpcgen といった開発用ツールを提供。

/usr/lib/*.a, /usr/lib/*.o libc6 で提供される動的ライブラリのうちのいくつかが静的ライブラリとしてインストールされる。また、コンパイル時に必ず静的にリンクされるオブジェクトコードが含まれる。

/usr/lib/*.so libc6 に含まれる /lib/*.so 動的ライブラリへのシンボリックリンクファイル。コンパイル用。

4.3.5 libc6-dev-sparc64, libc6-dev-s390x

64 ビット biarch 向けに作成されている libc6-dev パッケージ。Priority: standard, Section: libdevel。

Debian sarge では sparc64, s390x にのみ提供されている。基本的にパッケージの中身は libc6-dev パッケージと同等であるが、64 ビット化した動的ライブラリを通常とは別パスへインストールする。

4.3.6 libc6-dbg

デバッグライブラリパッケージ。Priority: extra, Section: libdevel。

libc6 や libc6-i686 パッケージで提供される動的ライブラリは strip されてシンボル情報が落とされているが、本パッケージで提供される動的ライブラリはシンボル情報も含んだものが/usr/lib/debug 配下にインストールされる。

本パッケージは、開発時など glibc も含めた gdb デバッグを行う際に使用する。LD_LIBRARY_PATH 環境変数に /usr/lib/debug パスを含めてアプリを起動すると利用できる。

4.3.7 libc6-prof

プロファイルライブラリパッケージ。Priority: extra, Section: libdevel。

libc6-dev で提供される静的ライブラリとほぼ同じだが、プロファイルオプション (-pg) つきでコンパイルされている。gprof を使ってアプリの性能プロファイリングを行いたいときに利用する。

4.3.8 locales

国際化機能 “ロケール” 用データを提供するパッケージ。Priority: standard, Section: base。

[*5] cmov 拡張は、VIA C3 以外の全ての 686 クラスプロセッサが持っている。

Debian で日本語を使いたい場合には必ずインストールされている必要がある。本パッケージは以下のファイルを含む。

/usr/share/locale/* 国際化 (gettext 化) した glibc の出力メッセージデータ (.po)。
/usr/share/i18n/charmaps/* ロケールデータのうち文字コードに関するデータ。
/usr/share/i18n/locales/* ロケールデータのうちロケール情報に関するデータ。
/usr/sbin/locale-gen ロケールデータ生成スクリプト。/etc/locale.gen に基づいてロケールデータを生成し/usr/lib/locale へインストールする。

元々locales パッケージには、コンパイル済のロケールバイナリデータ全てが含まれていたが、使いもしないロケール情報にサイズを食われることに強い反対意見があった。そこで現在は、各ユーザが charmaps (ja_JP.eucJP の eucJP) と locales (ja_JP.eucJP のうち ja_JP) データを使用してユーザが必要とする分だけ手元で生成出来るようになっている。“dpkg-reconfigure locales” によって生成するロケールデータを随時設定変更可能。

4.3.9 nscd

NSCD (Name Server Cache Daemon) パッケージ。Priority: optional, Section: admin。

デフォルトで提供される glibc ネームサービススイッチライブラリ (libnss_*) は、毎回関数を呼び出す度にネットワークへ問い合わせに行く仕様になっている。例えば libnss_dns は、DNS を引くための機能を提供するが、DNS を引く関数を呼ぶ度にネームサーバへ通信が発生してしまう。nscd を使うと、ローカルへ問い合わせをキャッシュするので、その分速度を速めることが可能になる。

4.3.10 glibc-doc

ドキュメントパッケージ。Priority: optional, Section: doc。

glibc に関するドキュメント (/usr/share/doc/glibc-doc/*) と、公式マニュアルである info (/usr/share/info/libc.info*)、そしていくつかのマニュアルページ (/usr/share/man/man3/*) が提供される。

なお、manpages-dev で提供されるマニュアルページと glibc-doc の info ドキュメントは、互いに提供元も内容も異なる全くの別物である。

4.3.11 libc6-pic

PIC アーカイブパッケージ。Priority: extra, Section: libdevel。

元々 boot-floppies のために用意されていたパッケージ。フロッピに入れるには大きすぎる glibc ライブラリから必要な関数だけ抽出し、小さくカスタマイズするために使用する。現在も debian-installer の mklibs で使用されている。

4.3.12 libc6-udeb, libnss-dns-udeb, libnss-files-udeb

debian-installer 用 udeb パッケージ。Priority: extra, Section: debian-installer。

libc6-udeb は debian-installer が動作するために必要な最低限の動的ライブラリのみ含む。libnss-*-udeb は ssh を動作させるために s390 用に導入されたもので、フロッピ容量の関係で libc6-udeb から外された動的ライブラリが入る。

```
/* All asm/ files are generated and point to the corresponding
 * file in asm-sparc or asm-sparc64.
 */

#ifdef __arch64__
# include <asm-sparc64/delay.h>
#else
# include <asm-sparc/delay.h>
#endif
```

図 1　sparc における /usr/include/asm/delay.h の例。#ifdef によって読込むファイルを切り替えている。

4.4　linux-kernel-headers バイナリパッケージとその中身

linux-kernel-headers は、libc6-dev とともに C, C++ 開発時に必要となるパッケージである。ソースは glibc とは異なり Linux カーネルを元にしている。Priority: standard, Section: devel, 事実上の Build-Essential。

中身のほとんどは Linux カーネルヘッダファイルで構成される。本パッケージは以下のファイルを含む。

/usr/include/linux/* Linux カーネルソースでいう include/linux ディレクトリにあるヘッダファイルをコピーしたもの。アーキテクチャ非依存。

/usr/include/asm/* include/asm-* ディレクトリにあるヘッダファイルをコピーしたもの。アーキテクチャ依存。

/usr/include/asm-generic/* include/asm-generic ディレクトリにあるヘッダファイルをコピーしたもの。アーキテクチャ非依存。

ちなみに、アーキテクチャによっては 32 ビットと 64 ビットのカーネルヘッダが別々になっているものがある (sparc, ppc など)。その場合、asm に 32 ビットアーキテクチャ用ヘッダだけが入っていると、64 ビットバイナリをコンパイル出来ないことになってしまう。そこで、asm 以下には振り分けのためのダミーファイルを入れておき、コンパイルオプションの #ifdef によって適宜読むファイルを切り替える仕組みになっている (例を図 1 に示す)。

4.5　glibc, linux-kernel-headers ソースパッケージとその中身

本節では、ハックの一助になるよう glibc, linux-kernel-headers ソースパッケージの中身を簡単に紹介する。

4.6　glibc

glibc ソースパッケージは以下のような構成になっている。ここでは、特に重要な役割を持つファイルを取り上げる。

***.tar.bz2** ソースパッケージ。

prep.sh .dsc ファイルがなくても .orig.tar.gz からソースを解凍可能にするスクリプト。

debian/control, debian/control.in/* control ファイルとその元ファイル control.in/*。debian/rules control として control.in/* から control ファイルを生成する。

debian/debhelper.in/* 各バイナリパッケージ用 debhelper ファイルの雛形を格納。

debian/po/* locales パッケージで使用する debconf フロントエンド用国際化メッセージファイル。

debian/local/* Debian ローカルでインストールするファイル (例えばマニュアルや設定ファイル等) を格納。

debian/patches/* Debian ローカルパッチ。dpatch をベースにした独自のパッチシステムで管理。最新版では 66 個、総計 452KB (!) ある。

debian/rules, debian/rules.d/* ルールファイル rules と、ビルドの各段階に分けてより細かく記述したルールファイル rules.d/* から成る。

debian/shlibver shlib バージョン定義ファイル。

debian/wrapper/* -dbg 生成時用ラッパスクリプト。

debian/sysdeps/* ビルド時に各アーキテクチャ依存の情報を切り替えるための Makefile 集。

4.7 linux-kernel-headers

linux-kernel-headers ソースパッケージは以下のような中身となっている。

autoconfs アーキテクチャ毎に用意されているデフォルトカーネルコンフィグファイル /usr/include/linux/autoconf.h を格納[*6]。

debian debian ディレクトリ。

debian/patches Debian ローカルパッチが入っている。dpatch にて管理。

others ia64, parisc 用の特別ファイル。

testsuite テストプログラム。gcc-2.95, gcc-3.3, g++-3.3 に対して -ansi などのフラグをチェックしてコンパイルエラーが出ないか調べる。

version.h 出自のバージョン番号。/usr/include/linux/version.h としてインストールされる。

なお、debian/patches 以下には現在 36 個、総計 61KB 分のローカルパッチが入っている。これをパッチ種別毎に分類したものが表 2 である。

4.8 苦労話

libc6 パッケージは、通常のパッケージと比較して以下の点が大きな違いである。

- ほぼ全ての Debian ユーザが使用。

[*6] 余談だが、本資料を書いている途中で autoconf.h は biarch に対応していないというバグに気付いた。

分類	パッチ数
GCC -ansi 問題関連	9 個
#ifdef __KERNEL__ を忘れたヘッダへの追加/削除	8 個
削除されたシステムコールを元に戻す	2 個
include すべきでないヘッダを取り込んだ時の対策	5 個
クリーンナップ (適切なヘッダを取り込むなど)	5 個
glibc にあわせるため	7 個

表 2　linux-kernel-headers パッケージに含まれるパッチの分類

- バイナリパッケージの大半 (約 9100 以上) が依存。
- カーネルなどと同様に、アーキテクチャ毎に使用するソースが大きく異なるにも関わらず、全アーキテクチャで同じバージョンを提供しなければならない。

そこで、ここでは実際に起きた問題の一つを取り上げ、メンテナンスにあたって発生する様々な苦労の一端を紹介したい。

4.8.1　glibc 2.3.4-1 (experimental) をインストールすると起動しない問題

発生した問題

これは glibc 2.3.4-1 を experimental に dupload した時に発生した問題である。

i386 アーキテクチャでは、libc6 の他に最適化ライブラリ libc6-i686 が用意されている。カーネル 2.6 + i686 アーキテクチャを使っている場合、libc6-i686 に入っているとバイナリ実行時に/lib/tls/i686/cmov 以下の最適化動的ライブラリが優先して使用されることは既に述べた。この「どの動的ライブラリをロードするのか」をバイナリ起動時に実際に判断するのは、libc6 パッケージに入っている動的ローダ ld.so である。

ところが glibc 2.3.4 になってから、ld.so と libc.so 共通で使用している内部構造体に変更が発生し、新旧の libc.so と ld.so を混ぜて使用できなくなってしまった。これが原因で、以下のようなケースに陥ると、全バイナリがクラッシュして動作しなくなってしまうようになった[*7]。

- sarge の旧 libc6 (2.3.2.ds1) と旧 libc6-i686 (2.3.2.ds1) パッケージを先にインストールした状態で、新 libc6 (2.3.4) にアップグレードする場合。この時、新 libc6 で提供される新しいバージョンの ld.so が、旧 libc6-i686 で提供される古いバージョンの libc.so を読み込んでしまい、クラッシュする。
- 新 libc6 + 新 libc6-i686 をインストールした環境に旧 libc6 を入れ直すとやはりクラッシュする。
- sparc には libc6 の他に libc6-sparcv9, libc6-sparcv9b という 2 種類の最適化パッケージがあり、それぞれを片方ずつもしくは両方をインストールしたり削除したりしても、クラッシュしてしまう。

状態遷移図の登場

[*7] この問題は experimental を常用する強者ユーザによって発覚した。もし、unstable に直接 glibc をアップロードしていたら、それこそ阿鼻叫喚の騒ぎになったことだろう…。

上記のように、パッケージをインストールしたり削除したりといった状況に応じて、問題が起きたり起きなかったりする。そこで、結局最後は libc6 や libc6-i686 パッケージのインストール状況毎に応じて状態遷移図を書いて問題を整理した。それが、図 2 である。

例えば、Lo (libc6 2.3.2.ds1) + Po (libc6-i686 2.3.2.ds1) がインストールされている状態 Lo,Po から Ln (libc6 2.3.4) をインストールしようとすると、色付きの不整合状態 Ln,Po に陥ることが図から読み取ることができる。

解決策

この図 2 に基づき、インストールや削除、アップグレードやダウングレードでも問題が発生しないように工夫した glibc を作成した。具体的には、Debian ローカルパッチで使用可能になっている /etc/ld.so.nohwcap というファイルを場合に応じて制御するというものである。このファイルが存在すると、ld.so は最適化ライブラリを使用せず、libc6 に入っているデフォルトライブラリを使うようになる。つまり、ld.so と libc.so のバージョンはいつでも一致するようになってクラッシュしなくなるわけだ。そこで .preinst/.prerm に手を加え、インストールまたは削除される状態に応じてこのファイルを生成したり削除したりするような変更を行った。

i386 では、上記手法によって問題解決することを確認した。ただし、sparc 版では同様のテストを行える環境がないため、新しい glibc を sid にアップロードすると sparc ユーザによってはシステムクラッシュが経験できるかもしれない。

4.9 etch の TODO

glibc, linux-kernel-headers には、まだまだ様々な作業が積み残されている。本節では今後 etch にて行われる予定の作業のうち、Debian 全体に与える影響が比較的大きいものをピックアップして紹介したい。

4.9.1 etch におけるツールチェイン移行計画

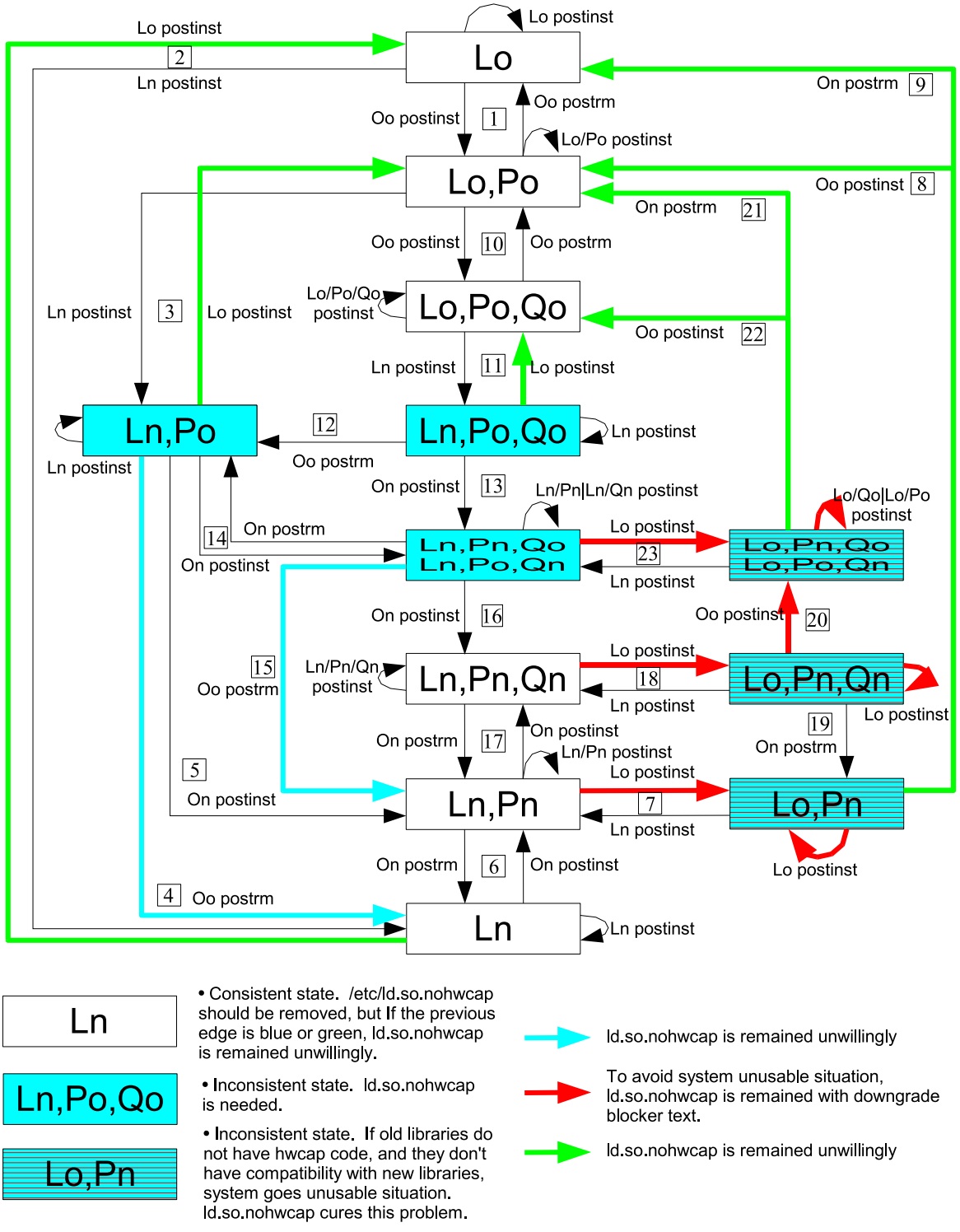
sarge がリリースするまでには非常に長い年月がかかったため、主要なツールチェインは全て昔のバージョンに塩漬けされた状態になってしまった。これを取り戻すべく、etch では最新版を 7 月に次々と投入予定である。最近の doko + jbailey + gotom の議論によって、次の順番でツールチェインを移行していこうという話になっている。

1. linux-kernel-headers: 2.6.0-test7 → 2.6.12 (sid では順次カーネルバージョンに追従予定)
2. binutils: 2.15 → 2.16
3. gcc: 3.3 → 4.0
4. glibc: 2.3.2.ds1 → 2.3.5 (experimental はさらに最新版へ追従予定)

この移行の中で最も影響が大きいのが gcc 4.0 である。gcc 4.0 では C++ ABI (Application Binary Interface) や、いくつかのアーキテクチャ(sparc, mips など) で関数の呼び出し規約に変更が加わっている。特に、C++ ABI 移行では、C++ を利用する全てのパッケージが一時的に名前を変えるなどの対処が必要になってくる。より詳細は以下の URL を参照のこと。

Debian Glibc package 2.3.2.ds1-20 (sarge) <=> 2.3.4-3 (etch) Diagram for breakable package upgrade/downgrade with sparcv9/sparcv9b optimized packages.

- gotom 2005-04-03 ver.5



Ln

• Consistent state. /etc/ld.so.nohwcap should be removed, but If the previous edge is blue or green, ld.so.nohwcap is remained unwillingly.

Ln,Po,Qo

• Inconsistent state. ld.so.nohwcap is needed.

Lo,Pn

• Inconsistent state. If old libraries do not have hwcap code, and they don't have compatibility with new libraries, system goes unusable situation. ld.so.nohwcap cures this problem.

ld.so.nohwcap is remained unwillingly

To avoid system unusable situation, ld.so.nohwcap is remained with downgrade blocker text.

ld.so.nohwcap is remained unwillingly

図 2　libc6, libc6-i686, libc6-sparcv9, libc6-sparcv9b 全てを考慮したインストール・削除パス

"C++ ABI transition for etch"
http://lists.debian.org/debian-release/2005/04/msg00153.html

4.9.2 multiarch サポート

現在の Debian は、いわゆる 64 ビットの biarch を部分的にサポートしている。とは言え、64 ビットをサポートするパッケージは libc6 や libncurses など一部のライブラリパッケージのみで、それも debian/rules でインストール先を /lib から /lib64 に振り分けるといった仕掛けを作り込んで実現している。

しかし近年、amd64 や ppc64 など、32 ビットと 64 ビットを両方使えるアーキテクチャが急速に普及してきており、より簡単に両者のバイナリを扱えるパッケージフレームワークの整備が求められている。また、3 種類以上のアーキテクチャを同時に使える multiarch [*8] も可能にしたいという意見がある。

現時点でどう実装していくのかについて議論はまとまっていないが、動的ローダに変更を加える可能性は高い。より詳細は以下の URL を参照のこと。

"Multiple architecture problem and proposed solution"
http://www.linuxbase.org/futures/ideas/multiarch/

4.9.3 新しいアーキテクチャのサポート

Debian sarge は 11 + 1 アーキテクチャにポーティングされているが、さらにもっと多くのアーキテクチャへポーティングしたいという声がある。

ppc64

現在最もサポートが望まれているのは ppc64 である。Debian で最初にネイティブ ppc64 の開発作業を開始したのは amd64 サポートでも中心的な役割を果たした Andreas Jochens であった[*9]。その後 debian-powerpc メーリングリストにおける議論で、ppc64 は amd64 と比較してネイティブサポートのメリットがそれほど大きくない[*10]ため、現在は biarch サポートでいく方針に固まりつつある。

sh3, m32r, mips64, parisc64

既にいくつかポーティングされているものもあるが、積極的にサポートする方向にはなっていない。sh3 に関しては、マシンの整備が進めば状況はもう少し良くなると考えている。

[*8] 例えば mips にはエンディアン 2 種類とは別に o32, n32, n64 といった異なる ABI が存在する。
[*9] http://debian-ppc64.alioth.debian.org/ を参照のこと。
[*10] ppc32 から ppc64 へはアドレス空間の拡大が中心でありレジスタ数も変わっていないため、ビット幅が増えた分、性能的には遅くなってしまう。

4.9.4 locales-all パッケージの提供

locales パッケージの解説で述べたように、現在コンパイル済ロケールデータはパッケージとして提供されておらず、ユーザがインストール時に必要なものだけコンパイルするようになっている。しかし、このロケールデータのコンパイルにはかなりのメモリと時間を食うため、特に組込み向けシステムでは生成が大変厳しいというバグレポートが報告されている。

そこで、locales パッケージの他に locales-all パッケージを用意し、コンパイル済ロケールデータをあらかじめ用意しておくことで、ロケール生成にかかる時間を節約できるようになる予定である。

4.9.5 timezone パッケージの作成

timezone データは libc6 パッケージに統合されているが、必ずしも必要なデータではないため、組込み機器向けでは libc6 から切離してスリムにしたいという要望がある。また、timezone データそのものは glibc 開発者がメンテナンスしているわけではなく、glibc とは無関係に頻繁に更新されている。

そのため、etch では timezone データを libc6 パッケージから切り離して独立させていく予定である。これまで安定版では timezone データを更新したくても一緒に glibc 一式を再コンパイルしなければならないリスクがあったが、別パッケージにすることでそれを回避できるというメリットもある。

4.9.6 カーネル 2.2 サポートの廃止

現在の Debian glibc は、Linux カーネル 2.2 から 2.6 までをサポートしている。しかし、既にカーネル 2.2 シリーズはメンテナンスされなくなりつつあり、Debian から消えつつある。

glibc はカーネルによってコンパイルするソースを切り替えているが、最適化パッケージをインストールしていない限り、いつまでもカーネル 2.2 の頃にあった古いシステムコールを利用してしまう。そこで、etch ではカーネル 2.2 サポートを廃止し、カーネル 2.4 以上でなければ動作できなくする予定である。

4.9.7 kernel version detection

woody → sarge の移行では、mips, sun4m, hppa, hppa64, real-i386 などがカーネルを最新版へアップグレードしない限り glibc も入れ替えられない状態が発生した。これは、カーネルと glibc の仕様が一緒に変更されたためである。

しかし、一旦カーネルと glibc をアップグレードした後、カーネルを昔のバージョンに戻して再起動されるとシステムがうまく動作しなくなる可能性もある。

そこで、カーネル 2.2 サポート廃止にあわせて、古いカーネルを検出した場合は/etc/init.d/glibc-kernel-check といったスクリプトによってユーザに警告を発せられるような仕組みを整えていく予定である。

4.9.8 NPTL のデフォルト化

RedHat の RHEL4 や Ubuntu ia64, ppc 版では linuxthreads に代わって NPTL が標準 PThread ライブラリとして採用されている。既に上流開発者も linuxthreads は今後メンテナンスをほとんど行わない予定と宣言している。近い将来 Debian でも 2.6 カーネルが主流になってくれば、linuxthreads と 2.4 カーネルはサポートを止めていく方向になるだろう。

4.9.9 バージョンアップの高速化

Debian では sarge の大幅なリリース遅れにより、glibc や linux-kernel-headers がかなり古いバージョンになってしまった。これは、リリースマネージャによるベースパッケージフリーズの影響が大きい[*11]。etch では、SCC の導入によってマイナーアーキテクチャの FTBFS に足をひっぱられなくなることもあるので、出来るだけ最新版を unstable に積極投入していきたい。

また、upstream とのより緊密な開発体制を敷くためにも、experimental を積極活用し、cvs 最新版を出来るだけ experimental へ投入していきたいと考えている。

4.10 おわりに

本文書では主に glibc, linux-kernel-headers を中心に Debian ツールチェインの解説を行なった。良く使われてはいるものの、普段あまり中身を知る機会のないと思われるパッケージであるが、これを機会に関心を持っていただければ幸いである。

[*11] リリースマネージャも sarge リリース間近のときに、2004 年 8 月に実施したベースパッケージフリーズは、ライブラリを無意味に古くさせてしまったと回想している。

5 dpatch をつかってみよう

上川純一

5.1 dpatch とは

Debian のソースパッチを管理するツールです． Debian パッケージでは，ソースパッケージは以下の構成になっています．

- .orig.tar.gz: オリジナルの tarball
- .diff.gz: Debian で作成した差分
- .dsc: dpkg 用制御ファイル

この中で，.diff.gz は一つの大きな差分ファイルとして管理されるため， どの部分がどういうパッチであるか，ということを管理してはくれません． その部分を実装するのが dpatch です．

通常の.diff.gz であると， `debian/`ディレクトリ以下の Debian パッケージング用の情報と それ以外のソフトウェア自体への修正が混合しています． それを整理するというものです．

5.2 ファイル構成

dpatch では，それぞれの小さな変更をそれぞれ独立したパッチとして扱います．それぞれのパッチを `debian/patches/xx_patchname.dpatch` という名前 で管理します．例えば，`debian/patches/01_configure.dpatch` という名前になります． そして，パッチの一覧を `debian/patches/00list` に適用する順番に記述します．そこでは，`01_configure` というような形式で記述できます．

この形式を採用しているため，Debian においての変更点が debian/ディレクトリ以下に集まり， また変更点を debian/patches ディレクトリで分類して管理できる，という利点があります．

dpatch でのファイル名が数字ではじまるのは昔はその番号で適用順序を決定していたなごりです．今はあまり数字を利用する必要性はありません．`debian/patches/00list` ファイルに指定した順番でパッチは適用されます．

5.3 道具

dpatch を利用するための道具を紹介します．

5.3.1 dpatch

dpatch コマンドは，パッチの適用とパッチをはずすという処理を実施してくれるコマンドです．従来は，makefile から include する Make スクリプトとして実装されていましたが，dpatch 2.0 からは実体が `/usr/bin/dpatch` シェルスクリプトになっています．

古い debian/rules では下記の内容を記述しています.

```
include /usr/share/dpatch/dpatch.make
```

最近の dpatch を利用するソースでは，dpatch コマンドを呼ぶように実装すればよいことになっています.
(/usr/share/doc/dpatch/examples/rules/rules.new.dh.gz から抜粋)

```
#!/usr/bin/make -f
#
# Sample dpatch rules file. Only example. Nothing else. :)
# This one uses the new way with dpatch from dpatch 2.x

export DH_COMPAT = 4

CFLAGS = -Wall -g
ifneq (,$(findstring noopt,$(DEB_BUILD_OPTIONS)))
CFLAGS += -O0
else
CFLAGS += -O2
endif

build: build-stamp
build-stamp: patch
        @echo "--- Compiling"
        dh_testdir
# Do something to build your package here
        touch build-stamp

clean: clean1 unpatch
clean1:
        @echo "--- Cleaning"
        dh_testdir
        dh_testroot
        dh_clean -k
# Clean your build tree

install: build-stamp
        dh_testdir
        dh_testroot
        dh_clean -k
        dh_installdirs
# Install it here

# Build architecture-independent files here.
binary-indep: build install

# Build architecture-dependent files here.
binary-arch: build install
        dh_testdir
        dh_testroot
        dh_installdocs
# And all the other dh_* stuff you need for your package.

# And now the simple things for dpatch. Here we only apply/unapply the patches.
# You can do more things with dpatch, like having patches only applied on
# a special architecture - see the non-dh version of the sample for this!
patch: patch-stamp
patch-stamp:
        dpatch apply-all
        #dpatch call-all -a=pkg-info >patch-stamp

unpatch:
        dpatch deapply-all
        rm -rf patch-stamp debian/patched

binary: binary-indep binary-arch
.PHONY: binary clean binary-indep binary-arch build install patch unpatch \
        clean1
```

5.3.2 dpatch-edit-patch

dpatch で利用するためのパッチを作成するコマンドです.

基本的な使い方は dpatch 管理下にあるソースコードのディレクトリで,

```
dpatch-edit-patch -d '説明文' 03_patchname 02_patchname
```

のように入力すると，二つ目のパラメータに指定したパッチまでのパッチを適用した状態で，一時ディレクトリにソースを展開してくれ，シェルが起動します．パッチ名には.dpatch 拡張子をつける必要はありません.

編集したのち，シェルから出ると，一つ目のオプションに指定した名前のパッチを作成してくれます.

debian/patches/00list ファイルは編集してくれるわけではないので，自分で編集する事になります.
debian/patches/00list ファイルを編集してパッチの名前 (.dpatch 拡張子をぬいたもの) を追加したらそのパッチが適用されるようになります.

5.3.3 dpatch.el

emacs 上で dpatch を使うために必要な，00list ファイルや，.dpatch ファイルの編集用のモードです.

まだまだ未完成です.

目標は dpatch-edit-patch の実装ですが，そこに至る前のコアの dpatch の部分の変更をしているだけで現在は時間が過ぎていってます.

5.4 作業フロー

作業のフローについては，これよりよい方法がある，などありましたら教えてください.

5.4.1 あたらしい upstream が出た時

Debian パッケージとして管理しているソフトウェアで，もととなっているパッケージ (upstream package) の新しいバージョンが出た場合の対応です.

- 前のバージョンから debian/ディレクトリをコピーしてくる
- `debian/patches/xx_patchname.dpatch` をそれぞれ `patch --dry-run -p1` で適用できるかどうか確認する
- `debian/patches/00list` を編集
- `debian/rules patch`
- **適用できないパッチの再作成**

5.4.2 あたらしく package をつくる時

新規にパッケージを作成する手順です.

- `debian/rules` を変更し，`/usr/share/dpatch/dpatch.make` を include し， clean と configure のルールで patch と unpatch ルールが呼ばれるようにする (patch-stamp 等を利用)
- touch コマンドなどで `debian/patches/00list` を作成する
- `dpatch-convert-diffgz` を実行して，とりあえず dpatch ファイルに変換する
- debian/patches/*dpatch ファイルを適当に編集して複数ファイルに分割
- `debian/rules patch` と `debian/rules unpatch` が成功することを確認
- `dpkg-buildpackage` で生成される diff.gz の diffstat をとり，debian 以下以外の場所が diff に含まれていないことを確認
- `dpatch-edit-patch` を利用して追加のパッチを作成

5.4.3 あたらしく patch を作成

- dpatch-edit-patch パッチ名とする.
  例： `dpatch-edit-patch automake`
  もし他のパッチに依存する変更をするなら，そのパッチ名を入力します.
  例 `dpatch-edit-patch automake autoconf`.

これを実施すると，/tmp 以下に適当なディレクトリでソースが編集できるようにシェルが起動します．

- ソースを適当に編集します．
- exit すると，パッチファイルが debian/patches ディレクトリ以下に生成されます．
- 生成されたパッチファイルに適当なコメントを書いておきます
- debian/patches/00list を適当に修正します
- debian/rules unpatch && debian/rules patch && debian/rules unpatch として 一応パッチが動作することを確認します

5.4.4 すでにあるパッチを編集

- dpatch-edit-patch パッチ番号 _ パッチ名とする．例：dpatch-edit-patch 03_automake
- 編集する
- exit

5.5 今後の開発

がんばれ．といいたいところですが，とりあえず現状の仕様をあらいだして，テストを作成して，全部の機能が動作することを確認するところからやろうと考えています．今は謎の機能が多すぎておいそれと触れません．

6 個人提案課題

名前 _____

下記の空欄を埋めてください:

Debian の toolchain の安定化のために，(　　)に注目し，今後の Debian の toolchain は (　　) します.

企画案の図：

7 Keysigning Party

上川純一

事前に必要なもの

- 自分の鍵の fingerprint を書いた紙 (`gpg --fingerprint XXXX` の出力．)
- 写真つきの公的機関の発行する身分証明書，fingerprint に書いてある名前が自分のものであると証明するもの

キーサインで確認する内容

- 相手が主張している名前の人物であることを信頼できる身分証明書で証明しているか[*12].
- 相手が fingerprint を自分のものだと主張しているか
- 相手の fingerprint に書いてあるメールアドレスにメールをおくって，その暗号鍵にて復号化することができるか

手順としては

- 相手の証明書を見て，相手だと確認
- fingerprint の書いてある紙をうけとり，これが自分の fingerprint だということを説明してもらう
- (後日) gpg 署名をしたあと，鍵のメールアドレスに対して暗号化して送付，相手が復号化してキーサーバにアップロードする (`gpg --sign-key XXXXX`，`gpg --export --armor XXXX`)

[*12] いままで見た事の無い種類の身分証明書を見せられてもその身分証明書の妥当性は判断しにくいため，学生証明書やなんとか技術者の証明書の利用範囲は制限される．運転免許証明書やパスポートが妥当と上川は判断している

8 次回

次回は 8 月 6 日土曜日の夜を予定しています．内容は本日決定予定です．
参加者募集はまた後程．