次世代GRUBブートローダの基本設計と実装

Designing and Implementing the next generation of the boot loader GRUB

奥地 秀則 [1)]　　　松嶋 明宏 [2)]
Yoshinori OKUJI　Akihiro MATSUSHIMA

1) フリーランス (〒600-8142 京都市下京区土手町通七条上る納屋町 406 E-mail: okuji@enbug.org)

2) 北陸先端科学技術大学院大学 情報科学研究科 (〒923-1292 石川県能美郡辰口町旭台 1-1 E-mail: amatsus@jaist.ac.jp)

ABSTRACT. The boot loader GRUB is widely used nowadays, but it has many defects in the code design, because it has been developed too fast with no refactoring effort. For the next version, we refine the basic design, improve the implementation, enhance the reliability and the maintainability, and give effort to frequently requested features. As a result, we created basic frameworks and other fancy features, such as internationalization.

1 背景

GRUB は GNU プロジェクトの一部として開発されているブートローダである。当初 Hurd のためのブートローダとして開発が始まったが、複数の OS モジュールを任意にロードできるように、UNIX シェルのような対話的インターフェースや、ファイルシステムを理解し動的にファイルを読み出す能力が求められた。さらに、この OS とブートローダの間のプロトコルを “Multiboot Specification” という中立的な仕様書にまとめたため、その他の OS でも利用可能な汎用ブートローダとなった。また、既存の OS との互換性も重視し、Linux、FreeBSD、NetBSD、OpenBSD などは直接ロードすることができ、可読性に富む設定ファイルやメニュー形式のインターフェースなど、一般的なユーザの利便性も考慮されている。

こういった Open Firmware のようなブートモニターにも匹敵する能力や、使いやすいユーザインターフェースのため、各種 GNU/Linux ディストリビューションや OS 開発プロジェクトの標準的なブートローダとして広く採用されている。今のところ、GRUB は IA-32 アーキテクチャに特化されているものの、汎用 PC だけではなく、IA-32 ベースの組み込み機器や大規模 PC クラスタにも普及しつつある。

しかしながら、急速に広がるニーズに応えるため、数多くの機能が追加され、次第に機能の追加や保守が困難になりつつある。例えば、独自に地域化を行ったバージョンを使用しているベンダがいくつか存在しているが、場当たり的な解決が行われている。これは GRUB が国際化に必要な基盤を備えていないためである。また、より柔軟に設定が行えるよう、以前からスクリプト言語のような機能が切望されているが、真のメモリ管理がないため、このような動的に対処すべき機能は不可能といってよい。

2 目的

本プロジェクトでは、基本設計を一から見直し、基盤となるフレームワークの実装を行う。ブートローダはサイズの制限が厳しいため、最小限の機能を持つコアをパーティション外に置き、安全性を高め、残りの機能は動的にファイルシステムから呼び出すことで、同時に拡張性も向上させる。また、高位の目標として、スクリプト言語、国際化、他アーキテクチャへの移植の三点を掲げ、UTF-8 のような非 ASCII 文字コードへの対応、様々なスクリプトの表示のためのフォントエンジン、機種依存部分の明確な切り分けなど、それらに必要な基本機能の開発も並行して行う。

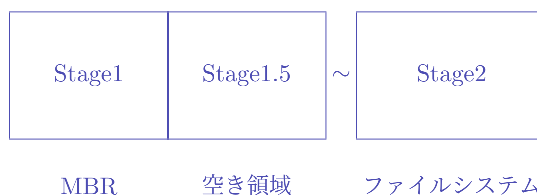

図 1: 従来の GRUB の典型的な物理的構造

なお、本プロジェクトのコード名として、“PUPA” が採用された。以下では、“PUPA” を GRUB の次期バージョンの呼称として使用する。

また、PUPA 以外にも、GRUB や PUPA の開発に有用なソフトウェアの開発も付加的に行った。これについては、第 6 章に別途記載した。

3 開発内容

3.1 ブートストラップ

PUPA 自身を起動・開始するための機能である。基本的な仕組は GRUB とほぼ同じであるが、安全性を向上させるため、いくつかの工夫が行われた。

従来の GRUB では (図 1 参照)、三つのバイナリイメージ、Stage1、Stage1.5、Stage2 が使われている。PC では最初に BIOS によってロードされる MBR と呼ばれる領域があり、そこに Stage1 が埋め込まれる。また、MBR の直後には通常 62 セクタ (31KB) の空き領域があり、ここに Stage1.5 が埋め込まれる。Stage2 は GRUB の本体と呼ぶべきものであるが、サイズの都合上、あるパーティション (ファイルシステム) 上に置かれる。

この方法では、Stage2 が置かれたファイルシステムが破損したり、パーティションが削除されたとき、全く起動しなくなるという脆弱性が存在する。そのため、PUPA では

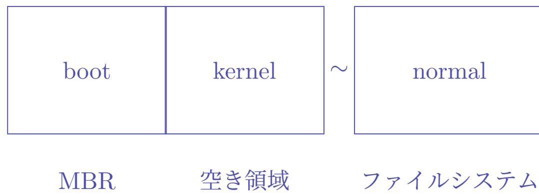

図 2: PUPA の物理的構造

図 2 の構造を考案した。

ここで、“boot” というイメージは GRUB の Stage1 に相当しているが、Stage1.5 のような中間的イメージは排除し、直ちに何らかの機能を提供できる、“kernel” と呼ばれるイメージを MBR の直後に埋め込んでいる。

また、“normal” と呼んでいるのは、後述するモジュールの一つである。このモジュールは GRUB の Stage2 に相当する機能を提供するもので、設置されているファイルシステムやパーティションに問題が発生しなければ、必ずロードされる仕組になっている。もし何らかの理由で “normal” がロードできなければ、“kernel” 自身に組み込まれた救援モードによって、最小限度の機能が提供される。この手法によって、安全性や信頼性を向上させている。

3.2 メモリ管理

従来の GRUB では、動的なメモリ割り合てがサポートされていなかった。PUPA では、ランタイムに発生する様々なメモリ要求に対応するため、libc の `malloc`、`realloc`、`free` 等に相当する機能を開発した。静的なメモリ管理は必要になるかもしれない最大限の領域を確保するため、非常にメモリの使用効率が悪い。しかし、動的なメモリ管理を併用することで、拡張性の向上だけでなく、メモリ使用効率も向上すると考えられる。

ただし、PUPA の主目的は OS をロード・起動することであるから、OS ファイルを読み込むために必要なメモリを消費してしまってはならない。そこで、PUPA 自身が使用するメモリ領域と、OS に必要なメモリ領域との間で、バランスを取る必要がある。これが一つの課題となった。

3.3 動的結合

PUPA をランタイムに拡張するための仕組である。

動的結合とは、“kernel” イメージにモジュールを結合させ、動的に新たな関数や変数群を追加することを指す。モジュールには、汎用性の視点から、ELF の再配置可能オブジェクト (relocatable object) 形式が採用された。

モジュール間のシンボルの解決は ELF の機能だけで十分だが、“kernel” は裸のバイナリイメージとして作成しなくてはならないので、そのままでは “kernel” とモジュールを結合することが出来ない。ゆえに、“kernel” は自身がエクスポートするシンボルを、ELF に頼らずに保持しなければならない。この問題は後述する構築システムで自動化を行うことによって解消した。

なお、動的結合は再コンパイルなしに機能の追加や削除を可能とするため、C プリプロセッサ等による条件コンパイルが不必要となり、単一のバイナリだけを管理すれば良いので、保守性の向上が見込まれる。

しかし、このやり方では、モジュールをファイルシステム上に配置した場合、そのファイルシステムを理解するためのモジュールがロードできないと、何もロードできないという問題が起こる。これは、“preloaded modules” という仕組で解決した。このことは第 4 章で詳しく論じる。

3.4 オブジェクト指向フレームワーク

PUPA に存在する様々なオブジェクトを構造化する。

従来の GRUB では、これらのオブジェクトは大域的な状態によって保持され、構造化が十分ではなかった。そのため、ファイルが一度に一つしか開けない等、機能を追加する上で、多くの難点が露出している。また、構造化が十分でないと、コードの見通しが悪く、保守が困難である。

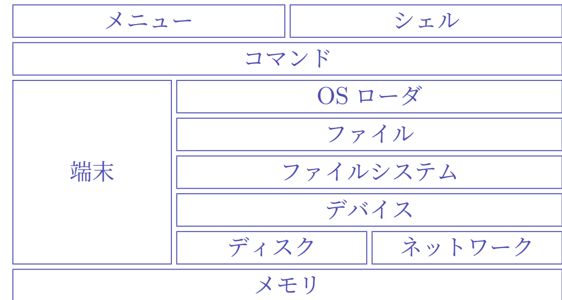

図 3: PUPA のフレームワーク

PUPA では図 3 に示した構造を用いている。これらの階層を守り、階層をまたぐような状態参照や変更を最小化し、機能モジュールの独立性を高めている。さらに、各階層のインターフェースを明確化し、動的結合と組み合わせることによって、機能の拡張が容易になった。

3.5 後方互換性

従来の GRUB に存在する機能を確保することが目的である。

PUPA の基本設計は GRUB とは大きく異なるだけでなく、移植性を高めなければならないので、それらの機能はほとんど一から書き直さなければならない。

計画段階では、具体的にどの程度の機能を再実装するかを決定していなかったが、打ち合わせの際に、以下の機能を実現することを目標に定めた。

ファイルシステム FAT ファイルシステム。簡単であるため。

ディスク FDD と HDD。

端末 コンソールとシリアル。

OS ローダ チェインローダ (他のブートローダをロードすること)、Multiboot、及び、Linux。

コマンド デバッグに有用なものと、OS の起動に最低限必要なもの。

3.6 クロスプラットホーム・インストール

異なるアーキテクチャ上で、PUPA をインストールするための機能である。

従来の GRUB では、あらゆるコードが IA-32 を想定して書かれているため、実際の動作だけでなく、インストール作業も IA-32 上で行わなければならない。これは組み込み機器の開発のように、異なるマシン上で開発を行う場合には、極めて不便であると言わざるを得ない。

そこで、PUPA のコードは、ターゲットとホストのアーキテクチャが異なる場合を考慮した。C プリプロセッサの力を借りて、データ構造の差異を吸収し、エンディアンの違いやワードサイズの違いを注意深く解決した。

3.7 非 ASCII コード処理

PUPA の国際化の一貫として、ASCII コード以外の文字コードを扱えるようにする。

様々な文字体系に対応させるため、Unicode が選択された。エンコーディングには、ASCII コードとの親和性のため、UTF-8 を中心に扱い、端末中では UCS-4 を用いている。

3.8 フォント管理

様々な文字体系を表示するために、フォントを管理する。

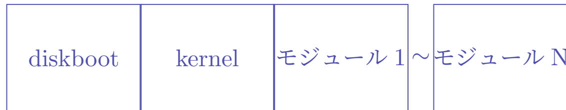

図 4: “core” イメージの形式

PUPA では異なる解像度に対応させる意味はあまりないので、ビットマップ形式のフォントを使用することにした。

フォントのデータ形式は BDF や PSF 等も検討したが、PUPA が持つメモリ制限を考慮し、任意のグリフだけを必要に応じてロードしやすくしなければならない。そのため、今回は独自の形式を選択している。

フォントのデータは既存の自由なフォントを流用し、独自形式に対応させるため、変換ツールを作成する。

3.9 グラフィックスコンソール

PC のテキストモードでは、扱えるグリフの数が少ない。そのため、グラフィックスモードに切り換え、フォントを描画するための端末を作成する。

3.10 メッセージ翻訳

PUPA が内蔵するメッセージを各言語に翻訳するためのシステムである。

PUPA はエラー・メッセージをはじめ、いくつものメッセージを内蔵している。これらは英語で書かれているが、英語を母国語としないユーザの利便性を向上させるため、メッセージを表示の際に翻訳されたテキストに変換する。

3.11 日本語化

上記の各機能を利用して、地域化を行う。地域化の対象として、開発者にとって最も重要な日本語を選択した。

3.12 構築システム

PUPA をコンパイル・インストールするための基盤である。

GRUB では、GNU Automake を Makefile のジェネレータとして採用しているが、Automake を使い続けることは困難であった。なぜならば、Automake は汎用目的であるにもかかわらず、利用するソフトウェアの都合に応じて、拡張する手段を提供していないからである。

PUPA では、独自の Makefile ジェネレータを作成し、“kernel” のシンボルの解決やモジュール間の依存関係の解決の自動化を図った。これらの自動化によって、データの重複を排し、人間が管理しなければならない情報量を大幅に削減することに成功した。

4 結果

4.1 物理的な構成

3.2 節で記したように、“boot”、“kernel” の二つのイメージを作成した。モジュールとしては、主要な機能を提供する “normal” だけでなく、OS をロードするためのチェインローダと Linux ローダ、及び、FAT ファイルシステム用のモジュール群を作成した。

実際には、もう一つのバイナリイメージが必要とされ、これは “diskboot” と呼ばれている。“diskboot” が必要となった理由は、“boot” だけでは、MBR(512 バイト) のサイズの制限のため、“kernel” をロードするコードを全て含むことが不可能だからである。“diskboot” のサイズは 512 バイトで、“kernel” をロードするだけの機能を持つ。

これらのイメージやモジュールは、後述するインストール用プログラムによって、単一のバイナリイメージを形成する。このバイナリイメージを以下では “core” と呼ぶこととする。

“core” は図 4 で示したように、“diskboot” を先頭に、“kernel”、及び、任意のモジュール群を付加した形式である。モジュール群は全くなくてもよいし、メモリやディスクの制限を除けば、いくつあってもよい。これらのモジュールは起動時にあらかじめロードされ、決してアンロードされることがない。この手法を “preload” と呼び、“preload” が行われたモジュールを “preloaded module” と呼ぶ。

この “core” イメージはコンパイル時ではなく、インストール時に動的に作成される。そのため、さまざまな環境に適したイメージを再コンパイルなしに生成可能である。しかしながら、現実の利用では、緊急時に必要な機能、及び、新規のモジュールをロードするために必要な機能のみを選択することになるだろう。すなわち、OS ローダと、モジュール群を配置しているファイルシステムを理解するためのモジュールである。もしユーザがコンソールの無いシステムを利用している場合には、シリアル端末をサポートするためのモジュールをロードしておく、というようなことも考えられる。

	gzip	LZO
解凍するコードのサイズ	約 4KB	446 バイト
圧縮率	0.54	0.59

表 1: gzip と LZO の比較。圧縮率は、開発中に作成した 32448 バイトの “core” イメージで実験した結果である。

4.2 イメージのサイズの問題

4.1 節で述べたように、“core” イメージが PUPA においてプログラムの主要部分を提供するが、3.1 節で示した安全性を確保するには、そのサイズが高々 31KB でなければならない。開発を進める中で、このサイズ制限は非常に厳しいことが判明した。そこで、生成されるコードのサイズを減少させることが急務となった。

使用しているコンパイラ、GCC でサポートされているオプションに、`-mrtd` や`-mregparm` がある。`-mrtd` は関数呼び出しにおける、引数の扱いを変化させるもので、関数の引数の数が固定の場合、呼び出し側ではなく、呼ばれた側がスタックを調整するという規則を用いる。ある関数が複数箇所から呼び出される場合、スタックを調整するためのコードが一回しか現れないので、サイズの減少に役立つ。

また、`-mregparm` は IA-32 に特異的なオプションで、関数の引数をレジスタを使用して渡すようにする。何個のレジスタを使用するかは明示的に指定し、最大で三つまで使用可能である。スタックの使用が抑制されるので、小さな関数では、スタックからレジスタに転送するという余分な処理が排除されるわけである。PUPA では小さな関数が大部分を占めるので、これもサイズの減少に役立つ。

これらのオプションを使用することで、約 2KB の減少に成功したが、これでも確実に増え続けるコードに対処するには不十分であった。なぜなら、IA-32 のリアルモードで動作するコードは再配置が困難で、“kernel” 自身に組み込まざるを得ないので、“kernel” のサイズの増加を抑えることは実質的に不可能だからである。

そこで、Linux 等で用いられているような、イメージの一部を圧縮しておき、起動時に自己解凍を行うという手法を採用することにした。圧縮するアルゴリズムとして、当初 gzip を考えたが、LZO も候補に上がった。LZO は LZ77 の変種であるが、gzip より解凍が高速で、サイズが小さいという特徴を持つ。しかし、圧縮率は gzip より若干劣っている。表 1 は gzip と LZO を比較した結果である。

さて、31KB に詰め込むことが可能なサイズの上限は以下の数式で推定できる。

$$(31 \times 1024 - 512 - S)/R \tag{1}$$

ここで、除算している 512 は、圧縮するわけにはいかない “diskboot” のサイズであり、S は解凍するコード自身のサイズ、R は圧縮率である。それぞれのアルゴリズムをこの式で計算すると、gzip では約 49KB、LZO では約 51KB となる。

この結果、及び、高速性や、ライセンス条項が GPL で問題がないことから、LZO を採用することにした。実際の利用法としては、“kernel” の先頭部分に自己解凍に十分なコードを配置し、インストール時に、自己解凍コードの後方に位置するコードや “preload” されたモジュール群を圧縮しておく。起動時には、自分自身の圧縮されたデータを解凍し、適切な箇所に解凍データを再配置する。この方法によって、物理的な構成を外面的に変化させることなく、サイズの減少を行うことができた。コードの実質的に利用可能なサイズは約 20KB 増大し、現時点では非常に余裕のある状態となっている。

4.3 メモリ管理

空き領域を singly-linked list を用いて管理する、単純なデータ構造を用いて、実装を行った。空き領域と使用領域を異なる 32 ビットのマジックナンバーで印を付けることで、メモリの不正使用をある程度検出することができる。また、ELF の再配置可能オブジェクトをロードするために、`memalign` 関数を実装したが、libc のものは余分に確保する領域が大きく、無駄が多いが、PUPA では `memalign` を特別に扱い、必要最小限のメモリだけを確保するようにしている。

PUPA では仮想メモリを使用しないため、物理メモリ上に存在する memory hole の影響を無視できない。そのため、空き領域を任意の数だけ追加し、分断されたメモリ領域を処理できるように設計した。

メモリ管理における最大の問題は、OS イメージをどのようにロードするか、である。OS イメージは任意のアドレスにロードできるわけではないし、PUPA が OS イメージをロードするためのメモリを奪ってはならないからである。現時点では、静的に特定のメモリを OS 用に予約することで解決している。チェインロードで必要となる、`0x7C00` から `0x7DFF` は使用を避けている。通常の OS ローダのためには、1MB から始まる連続領域のうち、3/4 を予約している。この問題については第 5 章で論じる。

4.4 動的結合

ELF の再配置可能オブジェクトをランタイムにロードし、リンクする機構を実装した。モジュールが不必要になった時にメモリを解放できるよう、アンロードを行うこともできる。しかし、モジュール間に依存関係が存在することもあり得るので、あるモジュールが依存しているモジュール群をリスト構造で表現し、参照カウントを管理することによって、アンロードが不用意に行われないように設計した。

モジュールをどう管理すべきか。当初はファイル名に基いて、名前を付けることも考えたが、標準のモジュールではなく、ユーザ独自の代替モジュールを使用する場合、ファイル名では十分とは言えない。なぜなら、ファイル名が異なるが、同様の機能を提供するモジュールがロードされると、そのモジュールに依存しているモジュール群の依存関係を正しく計算できなくなるためである。そこで、モジュールは自分自身の名前を ELF のシンボルに関連付けることによって、ファイル名とは別に保持することにした。標準的なモジュールでは、ファイル名とモジュール名は一対一に対応しており、理解しやすい仕組になっている。

モジュールはそれ自身で初期化や後処理が行えると都合が良い。そこで、モジュールがロードされたり、アンロードされたりする時に、特定の名前が付けられた関数を自動的に呼び出すことにした。それぞれ、`pupa_mod_init`、`pupa_mod_fini` と名付けられている。これらの関数をモジュール内に定義することによって、新しいコマンドを追加する、といったことが容易に行える。

モジュール間の依存関係については、4.7 節で述べるように、コンパイル時に自動生成している。

4.5 オブジェクト指向フレームワーク

3.4 節で書いた、階層的なフレームワークを実装した。これらの各階層は新しい機能を動的に追加・削除することをサポートしている。ただし、ネットワークについては、時間の都合上、実装を行っていない。

フレームワークのうち、メモリ、ディスク、ネットワーク、ファイルシステム、ファイル、OS ローダ、端末、メニュー、及び、シェルは何らかの実体を伴っているので、特に説明は不要であろう。そのため、ここでは残りの仮想的な階層について説明を加える。

デバイスとは、GRUB におけるドライブにおおよそ相当するもので、ファイルを扱う機能を持ったオブジェクトの総称である。PUPA では、GRUB と同様、物理的に存在するファイルシステムだけでなく、ディスク位置を直接指定するブロックリストや、ネットワーク・プロトコルをサポートしたい。したがって、デバイスは実際にはディスクかネットワークであり、底辺にあるオブジェクトがディスクであるか、ネットワークであるかを意識しないで記述するための階層である。

デバイスはユーザが直接指定可能である。その表記法は十分な自由度を設けており、“,”、“(”、“)” を除く、あらゆる文字が使用可能である。どういう表記が妥当であるかは、その底辺に存在するディスクやネットワークのドライバが任意に決定してよい。

コマンドについても若干の説明を行いたい。コマンドは実際には二つの種類があり、それぞれ、救援モード用と通常モード用である。これらのモードは幾分性質が異なるため、別個に管理しなくてはならなかった。というのも、救援モードは “kernel” に内蔵されるので、サイズの制限を受けるから、複雑なコマンドを実装するのは無理があると判断したためである。通常モードにはこのような制限はないので、コードサイズを気にせず、豊富な機能を実装してよい。しかし、この事については再考が必要だと思われる。詳しくは第 5 章で議論する。

さて、こうしたフレームワークは明らかに GRUB より優れた設計であるが、もう一つ、ディスク・アクセスの高速化は特筆すべきであろう。GRUB では、ディスクの読み込みを高速化するために、キャッシュを一応備えている。しかし、このキャッシュは 1 シリンダだけに留まっており、フラグメント化されたファイルにはほとんど効果が無い上、複数のデバイスへアクセスすると、すぐに無効になってしまう。

この問題を解決するため、PUPA では、複数のキャッシュを同時に保持することにした。極端に効率の良い手法は必要ないと考えられるので、昨今の CPU で頻繁に用いられているような、一次元配列を使った構造を利用している。この方法では、各ディスクに与えられた一意な識別子と、キャッシュされるデータの先頭セクタ位置から、ハッシュ値を計算し、そのハッシュ値に相当する配列要素へデータを保存する。すでに同じハッシュ値が使われている場合には、既存のキャッシュを解放するという、単純なアルゴリズムである。

現時点では、キャッシュは 8 セクタ (4096 バイト) 単位で行われ、最大で 1021 個のキャッシュを保持可能である。そのため、最大で約 4MB のデータをキャッシュすることができる。また、メモリ不足に陥った時には未使用のキャッシュを解放する機構も備えている。

4.6 互換機能の実装

本プロジェクトは次世代 GRUB の基盤を生み出すことが目的のため、利便性や実用性は後回しになった。このた

めに、3.5 節で上げた全ての項目を実装することはできなかったが、基盤が本当に有用であるかを試験するに十分なだけの機能を実装することには成功した。

FAT ファイルシステム、FDD や HDD へのアクセス、コンソール、チェインローダ、Linux ローダ等は基本的には実装が完了している。シリアル端末や Multiboot ローダは実装できなかった。実装しなかった理由は、単に他の開発の優先順位が高く、時間が足りなかったためである。

Linux ローダは完全ではない。initrd がサポートされていないためである。これも時間の制約上、やむを得ず、省略したものである。

4.7 構築システムと移植性の確保

PUPA の最大の目標の一つは、移植が容易なブートローダを設計することであった。これは構築システムと密接な関係にあるので、同時に記述することにする。

実際に移植が行われた訳ではない現状で、移植性について考えることは非常に難しいが、以下の項目を考慮しなくてはならないだろう。

- 異なるプラットホームでは、コンパイラに与えるオプションは異なるだろう。
- 異なるプラットホームでは、使用するソースファイルが異なるだろう。
- 異なるプラットホームでは、作成したいイメージが異なるだろう。
- 共有可能な箇所は、出来る限り、共有するべきだろう。

これらから導かれる結論は、プロジェクトを構築するためのルールはプラットホームに特異的にならざるを得ないということである。一つのルールにわずかな変更点を加えて、共通のルールから構築を行うというアイディアは興味深いが、現実味に乏しいと考えられる。

また、3.12 節で述べたように、GNU Automake は非常に便利な道具であるが、拡張性に乏しく、生のバイナリイメージの作成等、独自の規則を多く必要とする PUPA では利用が困難である。

そこで、オブジェクト指向スクリプト言語、Ruby を使用して、独自の Makefile ジェネレータを作成した。このジェネレータは極めて小さいものであるが、GNU Automake に似せた文法に従い、ソースファイル間の依存関係を自動計算する規則や、プログラムやイメージを作成する規則等を生成することが可能である。

このジェネレータを使って作成された Makefile は、共通の Makefile にインクルードされ、プラットホーム毎に別個管理することができる。

この Makefile は他にもいくつかの面倒な作業を自動化する機能を持っている。

上述した “kernel” がエクスポートするシンボルを決定するのに、ヘッダファイルに印を付けるという方法を採った。具体的には、関数には `EXPORT_FUNC`、変数には `EXPORT_VAR` というマクロを用いて、エクスポートしたいシンボルを指示する。これらのマクロは、プログラム自体の中で何の役目も行わず、C プリプロセッサによって除去される。しかし、Makefile から呼び出されるシェルスクリプトがヘッダファイルを走査し、これらのマクロを認識し、示された関数や変数をエクスポートするためのソースコードを自動生成する。これによって、エクスポートすべきシンボルを別に管理する負担が消失した。

モジュール間の依存関係も自動的に計算させている。各モジュールからエクスポートされているシンボルや、解決すべきシンボルのリストを抽出し、どのモジュールに解決すべきシンボルが含まれているかを検査することによって、依存関係を導出している。膨大なシンボル数でも高速に処理できるよう、C で記述したプログラムを併用した。そのため、この依存関係の計算は瞬間的に行うことができ、再構築の手間を削減することができた。

さて、移植性の向上には構築システムそのものだけでなく、ソースコードの中にエンディアンやワードサイズに対する依存性を混入させないことが肝心である。この問題に対処するため、プラットホームに依存しない、ビット数固定の整数型やポインタ型等を定義し、`int` 等の C に組み込まれた型を不用意に使わないようにしている。

また、ファイルシステム等、エンディアンに依存した処理が必要になる部分には、ターゲットの CPU とは無関係に特定のエンディアンに変換するマクロを使用している。

4.8 インストーラ

PUPA をインストールするため、二つのプログラムを作成した。

まず一つ目は、`pupa-mkimage` である。このプログラムは、与えられたモジュール名のリストを使って、“core” イメージを作成するのに使われる。内部では、“diskboot”、“kernel” の二つのイメージを読み込み、バイナリ中の固定アドレスにあるデータ変数を調整し、モジュール群を追加する。モジュール群は “preload” されるわけだから、“kernel” が正しく認識してロードできなければならない。これを補助するために、各モジュールにはヘッダが書き加えられる。

上記のように、“core” イメージは自己解凍に必須な先頭部分を除く、残りのコードやデータを LZO で圧縮する。この自己解凍コードはマシンの初期化の後、LZO のソースコードの一部を改変したコードによって、圧縮部分を解凍する、ということを行う。この箇所は 900 バイト程度であるが、将来いくらかコードを付け足す可能性を考慮し、現在は余分を見て、最初の 1KB は圧縮の対象にしないことにしている。

さて、こうして作成された “core” は、二つ目のプログラムの `pupa-setup` によって、ディスクからブート可能な状態にできる。`pupa-setup` のインターフェースは単純で、基本的にどこにインストールしたいかを指定するだけで良い。

内部は実際の動作時にも使われるコードを共有しているため、若干複雑になっている。共有するのはフレームワークの一部や、ファイルシステムのコードである。そこで、エンディアンやワードサイズの違いを解決するのに、二つの設定が必要となる。すなわち、構築に使っているマシンの設定と、ターゲットのマシンの設定である。

ターゲット側については、PUPA が特別にサポートしなくてはならないわけだから、エンディアンやワードサイズはヘッダファイルに手動で記述しておく。ターゲット用のコードを生成する時には、この設定が用いられる。

構築側のエンディアン等を管理するのは不可能に近いので、GNU Autoconf を利用した。Autoconf には最初からエンディアンや型の大きさを検査するマクロが付属しているので、それらをそのまま使っている。構築マシン用のコードをコンパイルしている時には、こちらの設定が用いられる。

4.9 国際化

ASCII コード以外の文字コードとして、各国語を統一的に扱える Unicode を用いた。外部では UTF-8 をエンコーディングとして利用し、内部では UCS-4 も利用する。

実際のところ、UTF-8 は ASCII コードとの互換性を維持しているので、大きな変更はさほど必要ではなかった。ただし、UTF-8 のバイト列を 1 バイト毎に出力することはできないので、ステート・マシンを使って、UCS-4 に変換する必要があった。UCS-4 に変換することで、内部的には 1 文字を 1 単位で扱うことが可能であり、可変複数バイトを扱うより、圧倒的に処理しやすい。

端末ドライバ毎にこの変換を行うのは効率的ではないので、端末を管理する上位層で UTF-8 から UCS-4 への変換

を行うことにした。各ドライバは UCS-4 だけを考えればよい。また、端末へのインターフェースに Unicode のコードポイントを直接出力できる関数を加え、任意の Unicode 中の文字を簡易に表示できるようにした。矢印等のように、頻繁に利用する文字群の扱いを簡略化するためである。

フォント管理では、ビットマップ形式が使われた。というのも、多くの解像度に対応させる必要性がないので、ベクトル形式には利点が無いからである。フォントのサイズは最も扱いやすい 8 × 16 ピクセルを基本単位としている。日本語のように、より大きく、正方形に近い文字を表すためには、この単位を横方向に 2 単位利用する。

フォントの形式について、BDF や PSF 等、既存の形式を検討したが、PUPA ではメモリをあまり消費しないようにする必要がある。そのため、必要に応じて、必要なグリフだけを得るとよいわけだが、速度性能に影響が出てはいけない。

この条件に見合う形式は見付からなかったので、今回は独自の形式を利用することにした。この形式を "PFF" (PUPA Fixed Font) と命名する。表 2、表 3、及び、表 4 にその構造の概観を示した。

識別子
オフセット・テーブルのサイズ
オフセット・テーブル
データ

表 2: PFF フォント形式の構造。

Unicode のコードポイント値
ファイル中のオフセット

表 3: PFF フォントのオフセット・テーブルに含まれる、各エントリの構造。

グリフの大きさ
ビットマップ

表 4: PFF フォントのデータに含まれる、各エントリの構造。

ビットマップを除く、各情報は 32 ビット、リトル・エンディアンの非負整数を使い、IA-32 上では極めて高速にアクセスできる。ビットマップは 8 × 16 ピクセルが一単位だから、16 の整数倍のバイト数であり、これも 32 ビット境界にアラインされている。

オフセット・テーブルが高速化に一役買っている。このテーブルによって、Unicode のコードポイント値から、それに対応するデータ位置を素早く割り出すことができる。テーブルは Unicode のコードポイント値に基いて、昇順にソートされており、二分探索法によって、$O(\log n)$ のオーダーで任意のグリフを検索できる。

実際のデータとして、GNU の unifont を利用した。このフォントはさまざまなフォントの集合体で、Unicode の広い範囲を網羅しているばかりか、8 × 16、及び、16 × 16 ピクセルのグリフを含んでいるので、PUPA の要件に非常に近い。

unifont は単純なテキスト形式で記述されているため、Ruby を使って、PFF に変換するツールを作成した。PUPA では、フロッピィのような、サイズの制限の厳しいディスクを使うことがあるので、特定の範囲のグリフだけを抜き出せるようなオプションも用意した。

次に、フォントを表示するため、グラフィックスコンソールを実装した。PC は通常テキストモードから開始するが、このモードでは同時に 256 個の文字しか扱えないので、日本語のような、文字数の多い文字体系には使いにくいからである。

どの環境でもサポートされていて、かつ、十分な数の文字を一画面に表示できるようなモードを使用しなくてはならない。そこで、通称 VGA16 と呼ばれている、VGA の 12h モードを使用した。このモードでは、解像度は 640 × 480 ピクセル、色数は 16 色である。フォントの大きさが 8 × 16 ピクセル単位なので、80 × 30 単位を扱うことができる。

メッセージ翻訳は、時間の都合上、関連するライブラリの調査や考察だけにとどめ、実装は行わなかった。

日本語化はメッセージ翻訳機能がないので、完全な形では実行できなかった。しかし、上述した unifont から日本語の部分だけを抜き出し、設定ファイルやプログラム中に書かれた日本語テキストを正常に表示できることを確認した。

ただし、グラフィックスコンソールの実装が不十分であり、まだ完全には動作しない。また、メニュー形式のユーザ・インターフェースも複数カラムに渡る文字に対応させていないので、動作に若干の問題がある。

5 考察

本章では、第 4 章で記述した結果を踏まえて、考慮すべき問題点や、今後の展望について詳述する。

5.1 メモリ管理

メモリ管理では、OS イメージに必要なメモリ領域が問題である。OS イメージは、多くの場合、アドレスが固定された形でコンパイルされている。そのため、PUPA は OS イメージをロードし、それを適切な箇所に再配置しなければならない。

再配置は PUPA が後処理を終えた後、起動する直前に再配置を行うことで実現している。この再配置用のコードが再配置の過程で破壊されないことを保証せねばならないが、既存の OS イメージは低位のメモリ位置を必要としないので、"kernel" の先頭付近に再配置用コードを配備すれば問題が生じない。

本当に難しいのは、OS イメージを保持するだけのメモリを予約しておく必要性である。上述した通り、1MB から始まる連続領域の 3/4 を予約しているが、これが最善であるとする根拠はあまりない。

仮に、実メモリが 4MB しかないシステムを考えてみよう。この場合、OS に残されたメモリ量は 2.25MB である。実メモリ量を考慮すると、これは理に適っているかもしれないが、ユーザの直感には反するかもしれない。

また、PUPA 自身が使える最大の連続領域は 0.75MB に制限される。これは圧縮アルゴリズムの bzip2 が最大で 900KB 必要とすることを考えると、かなり制限が厳しい。このような少ないメモリ量のシステムでは、十分に力を発揮できないのは明らかである。

この問題への対処として、OS イメージに必要な領域も動的に確保し、十分なメモリが得られない場合、必須でない機能に使用されているメモリを解放するという戦略が考えられる。

しかし、メモリはフラグメント化されるかもしれないので、必須でないメモリを解放したとしても、連続領域として十分な空き領域を作成できるという保証がない。また、OS イメージが、Linux の initrd や Multiboot のモジュールのように、複数に分かれており、しかも、それらの順序やメモリ位置が重要な場合、起動時に再配置を行うのが困難であろう。順序を入れ換えるのに、単純にコピーすると、他の OS イメージを上書きする危険性が存在するからである。

それ以外にも、何が必須でないかを決定すること自体が

難しい。例えば、コンソールを持つシステムでは、シリアル端末は必須とは言えないが、ディスプレイもないシステムでは、そうではない。また、全く英語が読めないユーザにとっては、国際化機能は必須といえる。
それゆえ、何が必須であるかを決定するのは自明ではない。それは環境やユーザに依存しており、機械的に判断することが不可能である。
この問題に対する完璧な解決法はないのかもしれない。

5.2 ディスク・キャッシュ

ディスク・キャッシュは低速なデバイスに対するアクセスを高速化する手法として重要だが、管理が簡単ではない。
必ずしも同一のディスクが付けられているとは限らないという問題がある。つまり、リムーバブルなディスクである。典型的には、フロッピィや Zip 等のメディアが考えられる。
これらのメディアはユーザが外部から抜き挿し可能なため、キャッシュしているディスクが途中で交換されると、キャッシュの内容とディスクの内容がもはや同一ではなくなってしまう。
GRUB では、キー入力待ちのように、一時停止を行う場合、必ずキャッシュを無効化している。実際的にいって、ユーザがメディアの交換を行うのは、明らかに動作が停まっている場合に限られるからである。しかし、このやり方は最善ではなく、交換を行わない場合にもキャッシュが追い出されてしまう。
PUPA では未だこの問題に対処していないが、ディスクのドライバがメディアの交換を認識できると良いだろう。ディスクへのメソッドとしてこの機能を実装し、適当なタイミングでこのメソッドを呼び出せばよい。もし交換されたことが判明したら、キャッシュを無効化するという動作を行うことになる。
問題は、どのタイミングでメソッドを呼び出すか、である。
一つは、GRUB と同様、一時停止を行う場所で呼び出す方法である。これは悪くない手法だが、一時停止が明らかでない場面で、呼び出しをやり損うかもしれない。例えば、ネットワークのレスポンスが悪い場合である。この場合、本当の意味で停止しているわけではないが、ユーザは停止していると捉えるかもしれない。このような箇所を全て洗い出すことは難しいと思われる。それに、そのような箇所全てにコードを挿入することは管理上の問題を引き起こすだろう。
もう一つの手法は、ディスクやディスク上のファイルをオープンする度に呼ぶというものだ。PUPA は原理的に一時停止中はディスクをオープンしたままにはできないから、ディスクをオープンするのは必ず動作再開の後になる。だから、このタイミングで呼び出すわけである。これなら、呼び出し箇所を集約可能だから、管理は非常に楽になる。しかし、呼び出しが頻発して、動作が遅くなる可能性もある。特に、メディアの変更を知らせるキャッシュ・ラインに問題がある場合、常にキャッシュを無効化してしまうことになり兼ねない。そのため、メディアの変更を検知する時、その時刻を記憶しておいて、あまりに頻繁に呼び出しがあったら、検知を実際には行わないというような工夫が必要かもしれない。

5.3 ネットワーク

今回はネットワークの実装を見合わせたが、若干の考察を行いたい。
ネットワークはデバイスの一種であるが、ディスクとはその性格を異にする。というのも、ディスクやその上に存在するパーティションは、必ず特定のファイルシステムを保持するのに対して、ネットワークは複数のプロトコルを持ち得るからである。
GRUB や PUPA は、ディスクやパーティションが与えられると、その中に存在するファイルシステムの自動検出を行う能力を持っている。これが可能なのは、デバイスの情報を完全に掌握しているからだ。
しかし、ネットワークではそうはいかない。あるプロトコルが使えるかどうかを容易には検出できないから、ユーザが指定するか、DHCP のようなプロトコルを用いて、サーバから情報を受け取らなければならない。
また、TFTP や NFS のようなプロトコルは同時に一つのデバイス上で共存可能なので、物理的なデバイスと論理的なデバイスが一対一に対応するとは限らない。
そのため、PUPA は内部的にネットワーク・デバイスとネットワーク・プロトコルをペアとして管理しなければならない。ユーザから見るデバイスは、物理的なデバイスやプロトコルではなく、ペアとして存在する仮想デバイスとなる。
このような仮想デバイスは組み合わせの数だけ存在するので、ユーザへの見せ方が厄介である。
ユーザやサーバによって、明示的に初期化されていないペアは全く存在しないとすることができるだろう。しかし、TFTP はネットワークからのブートにおいて事実上の標準プロトコルであるから、何もしなくても使える方がありがたい。
だからと言って、あらゆる組み合わせを最初から見せてしまうと、使いもしないペアが山のように表示されてしまうことになる。例えば、仮に AFS がサポートされたとしても、それを必要とするユーザは僅かであろう。
このように、デフォルトで作成すべきペアとそうでないペアが存在し、管理は複雑になってしまう。管理の難しさを軽減する方法として、ある物理デバイス上で使用できるプロトコルは同時に高々一つだけ、と規定することが考えられる。ある OS イメージは TFTP で、またある OS イメージは NFS でダウンロードする、といった使い方をすることはおそらくないだろうから、この方法でも良いのかもしれない。実際に試してみて、最善の道を模索するべきであろう。

5.4 コマンド

コマンドは現時点では二種類あり、一つは救援モード用、もう一つは通常モード用である。
救援モードはサイズの制限を大きく受けるため、別に管理すべきだと考えている。しかし、これは考え直した方が良いのかもしれない。
PUPA で実際に目にすることが多いのは圧倒的に通常モードの方であろう。だから、通常モード用のコマンドは使用頻度が高く、テストされる機会が多い。したがって、救援モード用のコマンドより早くデバッグされる傾向が生じるに違いない。
また、救援モードと通常モードで使われる機能には相当重複がある。これらの機能を別個に実装し、独立管理するのは無駄ではないか。
このような理由から、救援モードと通常モードのコマンドは統一する方が望ましいだろう。実のところ、コマンドの定義の方法はほとんど同一であるから、統一するのは簡単なはずである。
問題なのは、やはりサイズのことである。コマンドの中身だけなく、ヘルプ用の文字列があるためだ。
ヘルプ用の文字列は、ユーザがあるコマンドの使い方を忘れた場合に、シェル上で使用法を参照するためのものである。一つ一つの文字列は大した長さではないが、コマンド数が増えれば、段々無視できないサイズになっていく。
GRUB では、短いもので 20 文字、長いもので 500 文字程度であるから、仮に平均 100 文字としても、10 個のコマンドで 1KB 程度になってしまう。これが大きいか小さいかは判断に迷うところだ。現在では、LZO による圧縮があるので、十分無視できる程度かもしれない。今後の実験

が必要だろう。

5.5 インストーラ

インストーラについては、異なるアーキテクチャ上でインストールする機能を含め、一通りの機能を実装した。しかし、一般的な利用の観点からはまだまだ不十分である。

まず、デバイス・マップの自動作成がない。デバイス・マップとは、BIOS のドライブと OS の (デバイス) ファイルの対応を記述するための仕組である。GRUB ではエミュレータに内蔵させていたが、柔軟性を考え、PUPA ではデバイス・マップの作成部分を分離することにした。だが、実装が済んでおらず、現在は手動で作成してあげないといけない。これには多少の知識が必要となるので、幅広く利用されるためには、出来るだけ早く実装しなければならない。

次に、`pupa-mkimage` はナイーブ過ぎるという問題がある。どのモジュールを "core" に加えるかはユーザに任されており、自動的には決定されない。この決定には、PUPA の構造や機能に対する理解が不可欠であるから、とりあえず試してみたいというだけのユーザには負担が大き過ぎる。通常必要になるのは、OS をロードする機能と、モジュールが配置されているファイルシステムをサポートするためのモジュールなので、自動化は単純なはずである。おそらく、PUPA のインストールに使っている OS のローダ、及び、チェインローダを加え、さらに、モジュールの存在するファイルシステムを `mount` 等のコマンドで調べれば、必要なモジュールを決定できるだろう。

最後に、大抵の環境ではコマンド一発でインストールできるように、これら分割されたプログラムを組み合わせたプログラムを作成すると役立つだろう。PUPA を製品に組み込む場合、このプログラムをそのまま、あるいは、若干の変更を加えて配布するだけで、ユーザが即利用可能な状態に設定することが可能だろう。

5.6 国際化

国際化は当初の計画では松嶋が担当する予定だったが、進捗状況が捗々しくなかったため、奥地が支援する形となった。プロジェクト終了近くに奥地が実装を行ったため、不十分な点が多い。

まず、フォントの形式には十分な調査時間が割けなかったため、独自形式である PFF を考案した。無闇に独自形式を生み出すことは好ましいとは言えないから、もし PUPA に適切な既存の形式があるなら、そちらに乗り換えた方がよいだろう。

それ以外にも、フォント形式だけでなく、グラフィックスコンソール等にも言えることだが、多くの言語をこのままではサポートできないという問題がある。

欧米で使用されている言語や日本語、中国語等では、現在の実装でも十分であり、実際にその確認も行っている。しかし、インド系文字のような、合成文字を必要とする文字体系や、アラビア文字のような、右から左に記述する文字体系はサポートしていない。これらの問題は、他の多くのプロジェクトでも十分な解決が成されているとは言い難く、非常に挑戦的な課題だと言えるだろう。これらの文字体系にもビットマップ形式だけで対応できるとは考えにくい。おそらく、ビットマップを使うにしても、何らかのヒント情報を併用する必要があると思われる。

次に、フォント管理にはキャッシュ機構を備えることが望ましいが、まだ実装できていない。現実に使用するグリフの数はあまり多くないだろうから、使ったグリフはすべてメモリ上に保持してもよいかもしれない。いずれにせよ、何らかの高速アクセス機能を実装し、ユーザが不快感を味わずに済むようにすべきである。

最後に、メッセージ翻訳は調査のみとなった。時間の制約上、止むを得なかった。

さて、既存のメッセージ翻訳機能の有名なものとして、gettext、及び、catgets が挙げられる。

catgets は変わり易いプログラムでは極めて保守が困難である。というのも、catgets はプログラム中の翻訳したい各文字列に、その文字列自体とは無関係に、一意な識別子を割り振らねばならないからである。文字列を追加したり、削除したりする度に、どの識別子が現在使われているかを管理するのは厄介だ。

それに対し、gettext は翻訳したい文字列そのものを鍵にして、翻訳後のテキストを検索する。これは catgets より、プログラマの視点からは、圧倒的に扱いやすい。ただし、gettext にも弱点はあり、元の文字列がほんの僅かだけ変化しても、例えば、抜けていた読点を挿入するといった、内容に関係しない変更でも、翻訳物を見直さないといけなくなる。しかし、これは簡単に見て判ることだから、大きな問題ではない。

これらの事情を考えると、gettext を使用するのが最善であると考えられる。ただし、"kernel" 内に実装するのはサイズ的に好ましくないだろうから、"kernel" 外のモジュールとして実装せねばならない。すると、"kernel" 内に存在するメッセージはどのように翻訳するかを注意深く検討しないといけない。おそらく、ダミーの `gettext` 関数を "kernel" 内に作成し、本当の `gettext` 関数がロードされれば、本当の機能に切り換える、というような工夫が必要となるだろう。

5.7 実用化に向けて

今のところ、PUPA は一般利用を勧められる段階には達していない。なぜなら、インターフェースが完全ではなく、機能も不十分だからである。

インターフェースにおいては、コマンドラインを用いたシェル機能と、全画面方式のメニュー機能とがある。これらは双方共に "見せかけ" に近い状態で、さまざまな機能が未実装である。例えば、メニューでキーを押してもブートしないし、タイムアウト処理も無視されている。シェルでの補完や履歴等も、まだ存在していない。

機能でいうと、より多くのコマンドが必要である。GRUB には 61 のコマンドが存在するが、PUPA には比較的完成度の高い救援モードでさえ、10 のコマンドしか存在しない。GRUB に存在するコマンドが全て必要と言うわけではないが、それらの多くは有用なものであり、再実装を進めなくてはならない。特に、パーティションをアクティブにする機能はチェインロードに不可欠であり、早期の実装が望まれる。

これらの課題は基本的に時間と労力をかければ済む事であるから、さほど心配は要らないが、ファイルシステムは厄介だと思われる。

GRUB では、公式のものだけでも、八つのファイルシステムがサポートされている。PUPA では FAT ファイルシステムのみが実装されているが、ファイルシステムのサポートにはファイルシステムの内部構造に精通する必要があり、非常に手間がかかってしまう。これら全てを PUPA に移植するのは個人では時間がかかり過ぎるので、今後はボランティアの手を借りて、開発を促進する努力が不可欠になるだろう。

6 PUPA 以外のソフトウェア開発について (参考)

この章では、付加的に行われた、PUPA 以外のソフトウェアの開発について記述する。

6.1 BugCommunicator

BugCommunicator はバグ追跡システム (以下、BTS) である。

PUPA の元となっている GRUB では、PUPA と同様、GNU の Savannah というホスティング・サービスを利用

している。この Savannah は SourceForge のクローンで、統合型の開発プラットホームを提供している。Savannah では、SourceForge とは異なる、Xerox 社から寄付された、CodeX という名前の BTS を使用する。

GRUB での経験から、CodeX が持つ多くの欠点に気付いた。とりわけ、以下の項目が致命的であった。

- Savannah 上でユーザを作成し、そのユーザでログインしないと、ユーザの特定ができない。
- メールによる通知機能はあるが、メールで連絡を取り合うことができない。
- 新しい属性を任意に付加することができない。

CodeX の管理者と連絡を取り、これらの欠点を直せないかと相談をしたが、残念ながら、その試みは失敗に終わった。

そのため、他の BTS に乗り換えることを考えたが、BTS は一度使い始めると、そう簡単には乗り換えができない。なぜなら、既存のバグの情報を移し換えるのは困難だからである。よって、乗り換えを行うのであれば、出来る限り理想に近い BTS を採用しなくてはならなかった。

しかし、CodeX の長所である、WWW 上の分かりやすいインターフェースを同時に備えるものは皆無であった。例を挙げると、Debian BTS は拡張性に乏しいし、Bugzilla はあまりにインターフェースが複雑である。

こうした背景から、独自の BTS を一から作成することにした。こうして、BugCommunicator が誕生した。

BugCommunicator は以下の特徴を備えている。

- 自由なスクリプト言語 Ruby のみで書かれており、コードの拡張が容易である。
- 複数のプロジェクトを管理できる。
- WWW 上での検索や閲覧機能を持つ。
- ステートレスなので、リンクが張りやすい。
- 特定のバグを示す URI が不変である。
- フリーテキストだけでなく、選択方式の属性を任意に追加することができる。
- バグにファイルを添付することができる。
- バックエンドに MySQL を用いており、データ管理の分散が可能である。
- メールのみで情報交換が可能である。
- パスワードを一切用いないので、覚えないといけない情報が少ない。
- 内部を UTF-8 に統一することで、異なる言語環境に対応している。
- GPL に従って配布される、自由なソフトウェアである。

現時点での最新バージョンは 0.3 であり、そのホームページ
http://www.nongnu.org/pupa/bugcomm.html から入手できる。

すでに実用段階に到達しており、http://bugcomm.enbug.org/にて、BugCommunicator 自身、及び、GRUB の BTS として利用中である。

6.2 Ruby/Cache

Ruby/Cache は Ruby 用のライブラリで、オブジェクトをキャッシュするという目的を持つ。アルゴリズムは典型的な LRU である。

このライブラリは、上述した BugCommunicator のレスポンスを高速化するために作成された。多くのリクエストはリードオンリーであり、バグやプロジェクトの状態変更を伴わないと考えられるから、常にレスポンスの内容を計算するのは資源の無駄遣いであろう。従って、最近返したレスポンスの内容を、リクエストの内容と組にして保存しておくことで、再計算を防ぎ、レスポンスにかかる時間を減少できるかもしれない。

しかしながら、実際には BugCommunicator のレスポンスは予想外に速く、Ruby/Cache の組み込みは行っていない。そのため、現段階では独立のライブラリという形態でのみ、配布を行っている。今後検証すべきことは、BugCommunicator に登録されているバグの総数や、同時利用するユーザ数が増加した時に、BugCommunicator がその負荷に耐え得るか否かを確認することである。もし負荷が大き過ぎるようであれば、Ruby/Cache の利用を真剣に考慮することになるだろう。

Ruby/Cache のライセンスは Ruby と同様の自由なライセンスで、最新バージョンは 0.3 である。今までのバージョンは、
http://www.nongnu.org/pupa/ruby-cache.html からダウンロードできる。また、最新バージョンでは、新井康司氏によって書かれた FileCache というクラスを、Ruby/Cache の使用例として、同時配布している。

7 参加企業及び機関

なし

8 参考文献

なし