序章

何がわからないのか？
何を理解すれば良いのか？

真にLinuxを理解するために

西田 亙

● 何がわからないのか？

昨今，組み込み業界への注目が高まりつつありますが，組み込みOSとしてのLinuxに興味をもち始めている技術者の方々はかなりの数にのぼることでしょう．

しかし，いざ未知のテーマに取り組むとなると，なかなか最初の一歩を踏み出せないものです．「わからないもの」を恐れとして感じる人間の本能が，踏みとどまらせているのかもしれません．

それでは，Linuxに挑戦する人々は一体，「何を恐れている」のでしょうか？　いいかたを変えれば，「何がわからない」のでしょうか？　その理由さえわかれば，恐怖への正しい対処方法と理解への道のりが見つかるはずです．

● 恐れの対象を知る

筆者自身がそうでしたが，DOSの世界しか知らなかった人が初めてPC-UNIXに触ると，目にするものすべてがブラック・ボックスであり，途方に暮れるものです．最初の2年間は，果てしないインストールの繰り返しと，システム管理だけで過ぎ去りました．「UNIXのしくみを知りたい，使いこなしたい」という欲求とは裏腹に，どこから学習を始めればよいのか，わけがわからず，悶々とした日々を過ごしたことを覚えています．

数年間に及ぶ逡巡を打破するきっかけは，恐れの対象を絞り込むことから始まりました．「一体自分はLinuxの何がわからないのだろう？」としばらく考え続けた末に，筆者が出した結論は次のようなものでした．

(1) DOS上でのプログラムの正体はファンクション・コール(INT 21h)や，BIOS呼び出し，もしくはI/Oポートの直接操作であり，すみずみまで理解できる．しかし，Linux上のプログラムの実体はまったくわからない．

(2) DOS上の実行プログラム(COM型式)はセグメント内のオフセット0x100番地から始まる単純なものだが，Linuxはそうではないらしい．EXE型式に近いELF(Executable and Linking Format)というものらしいが，その正体は？

(3) DOSは単純なFAT(File Allocation Table)システム上でファイルを管理していた．それに対して，Linuxのファイル・システムはかなり複雑化しているらしい．しかも，DOSのようにフォーマットするだけで起動システムができあがるわけではないようだ．Linuxの起動には，ルート・ファイル・システムなるものが必要らしいが，その中身とは？

(4) Linux上ではプログラミング作法がDOSとはまったく違うらしい．Cプログラムをビルドするためには，gccというコマンドを使うそうだが，アセンブラやリンカはどうなっているのだろう？　スタンド・アロンのプログラムを作るためにLinux用crtファイルのソースを知りたいのだけれど，これらはどこにあるのだろう？

以上，四つの疑問にまで対象が絞られると，目ざすべき道はおぼろげながらも見えてきました．

● GNU開発ツールを知る

ここで，上記の四つの疑問のうちの三つ(3番目以外)の背景には，GNUの開発ツールが存在するという点に着目してください．

プログラム構造が開発ツールと密接に関連しているのはもちろんのこと，ELF実行可能ファイル型式も開発ツールに深く依存しています．

すなわち，Linuxを知るとは，GNUの開発ツールを知ることとほとんど同義だともいえるでしょう．

それでは，GNUの開発ツールとは何物なのでしょうか？一般的に知られているのはGCCですが，このコマンドは数多くのツール群の一つにすぎません．GNU開発ツールは大きく，binutils(BINary UTILitieS)パッケージ，およびGCC(GNU Compiler Collection)パッケージに分類されます(**図1**)．

開発の基本を構成するのは，binutilsに含まれるアセンブラas，そしてリンカldです．asとldさえあれば，プログラムを作成することができるので，両者はLinux界のアダムとイブだともいえるでしょう．

Cコンパイラ(cc1)は，GCCパッケージの中に含まれるコンパイラの一つです．Cソースは同パッケージの中に含まれるCプリプロセッサ(cpp0)で前処理され，cc1によりコンパイルされてアセンブリ・ソースに変換されます．

アセンブリ・ソースはasによってアセンブルされ，出力されたオブジェクト・ファイルをldが，crt(C Run Time startup)ファイルおよびライブラリとリンクすることで，実行可能ファイルが完成します(**図2**)．

実行可能ファイル型式には，DOSのCOM型式に相当する単純なバイナリもサポートされていますが，LinuxはELF型式のプログラムしか起動できないため(正確にはa.out型式も可能)，さまざまなヘッダやセクション情報，デバッグ情報，シンボル情報などが添付され，肥大化したプログラム・ファイルができあがります(**図3**)．

これらの情報を管理するためのコマンドが，binutilsに

含まれる，objdump(OBJect DUMP)，nm(symbol NaMe)，size，strip，readelf などです．

このように，GNU 開発ツールは，さまざまなコマンドの集合体から構成されているにもかかわらず，多くのプログラマは全自動でビルドを請け負ってくれるドライバ・プログラムである gcc に頼り切っているというのが現状です．

確かに，デスクトップ環境でのプログラミングでは，GCC が一つあれば事は足りるのですが，組み込み環境のようにターゲットのリソースやメモリ・マップに応じて臨機応変にコードを生成する必要がある場合には，このような安直な方法は通用しません．一つ一つのツールの特性を熟知し，それらを組み合わせて使う技術が必要になります．

第 1 章では，gcc ドライバの動きを解析しながら，cpp0，cc1，as，ld の四つの基本ツールの意義と，その使いこなし方を紹介します．

また，あわせて，binutils に含まれる強力な開発援助ツールを用いて，オブジェクト・ファイルや実行可能ファイルをさまざまな観点から解析してみます．第 1 章を読み通せば，GNU 開発ツールの全体像を俯瞰できるようになるでしょう．

● プロセスの正体を知る

開発ツールを使いこなせるようになったら，次は Linux プログラムの本質を探してみましょう．

C プログラミングの教科書に必ず登場する有名なプログラムとして，**リスト 1** に示す hello.c があります．

```
gcc hello.c
```

を実行すれば，ただちに実行可能ファイル a.out ができあがり，それを実行すれば画面上に“Hello, world!”のメッセージが表示されます．

上の事実からは，hello.c が a.out という 4K バイト前後の実行可能ファイルに変換されたことしかわかりません．問題は，メッセージが画面上に出力されるに至ったその「過程」にあるのです．

残念ながら，hello.c のソース・リストを穴があくほど見つめてみたところで，答は出てきません．すでに説明したとおり，開発ツールの基本は C コンパイラではなく，アセンブラとリンカにあるからです．

第 2 章では，hello.c プログラムの動きを，システム・コールのトレース・コマンドである strace を使いながら解析していき，Linux プロセスの本質がシステム・コールにあることを学びます．printf の正体は write システム・コールであり，return 文の戻り先は（main 関数の呼び出し元）は，_exit システム・コールを実行してプロセスは終了します（**図 4**）．

Linux システム・コールの実体はソフトウェア割り込み 128 番であり，引き数はレジスタを介してカーネルに渡されます．第 2 章では，実際にアセンブリ言語，および GCC の拡張インライン・アセンブラを用いて，システム・コール関数を手作業で作成してみます．

Linux プロセスの正体がシステム・コールにあることを見極めることができれば，もはや重厚長大な C ライブラリに頼る必要はありません．

第 2 章の後半では，簡単なライブラリを自作し，外部の crt ファイルや glibc に依存しない，完全に自立したプログラムをビルドします．

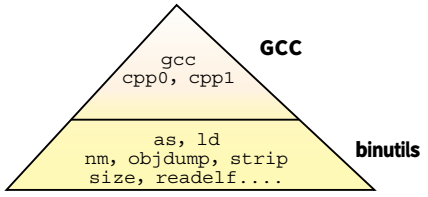

図 1
GNU 開発ツールの全体像

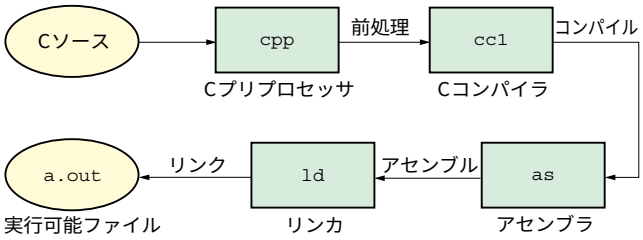

図 2　ビルドの工程

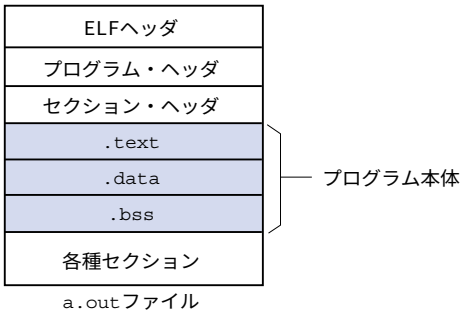

図 3
ELF 実行可能ファイル型式

リスト 1　hello.c

```
#include <stdio.h>

int main() {
  printf("Hello, world!¥n");
  return 1;
 }
```

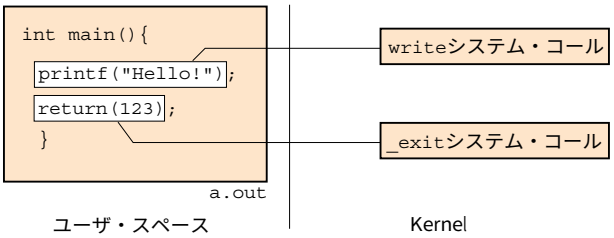

図 4　プロセスの本質

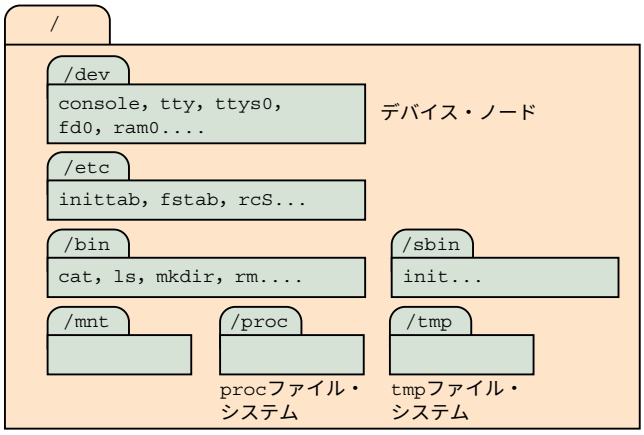

図5　ルート・ファイル・システム

● ルート・ファイル・システムを自作する

プログラムの本質が理解できれば，残すはUNIXの特徴の一つである，ルート・ファイル・システムだけです．

UNIXシステムは，大きく三つのパートから構成されます．一つはカーネル，もう一つは実装されるプログラムやライブラリ群，最後がルート・ファイル・システムです．

前二者の重要性は誰もが知るところですが，ルート・ファイル・システムの存在は，意外と見逃されていることが多いようです．

UNIX Kernelは，ルート・ファイル・システム上でなければ，活動することができません．そのもっとも大きな理由は，各種のデバイスが，ルート・ファイル・システムの/devディレクトリ中に格納された，デバイス・ノードを通じてアクセスされるからです．プログラマやユーザは，デバイスを名前で識別しますが，カーネル内部ではデバイス番号（メジャー，マイナ）を通じて，該当するデバイス・ドライバが選択されます．

つまり，デバイス名とデバイス番号の橋渡し役を演じるものが，/devディレクトリ中のデバイス・ノードなのです．

よって，ターゲットとなるシステムで必要になるすべてのデバイスを列挙し，対応するデバイス・ノードをあらかじめ作成しておく必要があります．

ルート・ファイル・システムには，このほかにも初期化ファイルや初期化スクリプトを格納する/etcディレクトリや，プログラムを格納する/bin，/sbinディレクトリなどが存在します．Linuxでは，さらにカーネル内部情報をユーザ・ツールに提示するための，/procディレクトリも必須です（**図5**）．

第3章では，公開されたばかりのLinux Kernel 2.6を用いて，フロッピ・ディスク版の最小linuxシステムを構築します．むだがまったく存在しないルート・ファイル・システムを構築することで，「何が必要で何が不要なのか？」，その判断を明解に下すことができるようになるでしょう．あわせて，Linux 2.6で新しく採用された諸機能についても簡単に紹介します．

● USBメモリ活用術

第3章までを通して，GNU開発ツールの使いこなし方，プログラムの本質，ルート・ファイル・システムの構築方法は理解してもらえると思います．最後の第4章では応用編として，今後の主流となるであろうUSBメモリ上に実用的なLinuxシステムを構築します．

従来のシリアル・ポートやコンパクト・フラッシュは，いずれUSBインターフェースさえ取って代わられていくものと思われます．実際，Linux Kernel 2.6では，USBシリアル上での接続もサポートしているため，組み込みボードがUSBインターフェースさえサポートしていれば，コンソール制御，無線LAN，USBメモリ，USBカメラなど，必要とされる機能をすべて実現することが可能になります．

ただし，コンパクト・フラッシュとは異なり，USBメモリからLinuxを起動するノウハウは，現時点ではほとんど公開されていないようです．そこで，第4章では高機能ブート・シェルであるGRUBを活用した，USBメモリ版Linuxシステムの構築方法を解説します．

USBメモリの劣化に対応するため，ルート・ファイル・システムはRAMディスク型式でマウントし，重要なファイルはジャーナリング・ファイル・システムのReiser FS上に保存し，WindowsやMac OS X上からデータ転送できるようにFAT型式のパーティションを用意し，syslogなどのログ・メッセージはRAMファイル・システムであるtmpfsに書き出すように工夫しています．

手軽に脱着できるUSBメモリ・システムは，ひとたび経験すると，これ以外の記憶装置は考えられないほど，便利なものです．皆さんも，ぜひオリジナルな可搬型USB-Linuxシステムの構築に挑戦してみてください．

● 幹を捉えること

本特集は「技術の幹を捉える」ことを念頭に置きながら，執筆しました．「木を見て森を見ず」という言葉がありますが，現実世界はその逆で，筆者を含め多くの人達は「葉を見て幹を見ず」の状態に置かれているように思います．

技術が進化し，メディアも多様化したため，ちまたにはデバイスや情報が溢れています．このような状況に浸っていると，時として「自分たちは日々進歩しているのだ」という錯覚にとらわれがちです．しかし，ふと我に返ると「基本が理解できていない」事実に驚かされるのです．

組み込みシステムは，利用できるリソースが限られている反面，一人の技術者が十分に理解できるサイズに収まっているともいえるでしょう．本特集が，組み込みLinuxの基本を理解し，そのおもしろさを知るための一助となれば幸いです．

にしだ・わたる