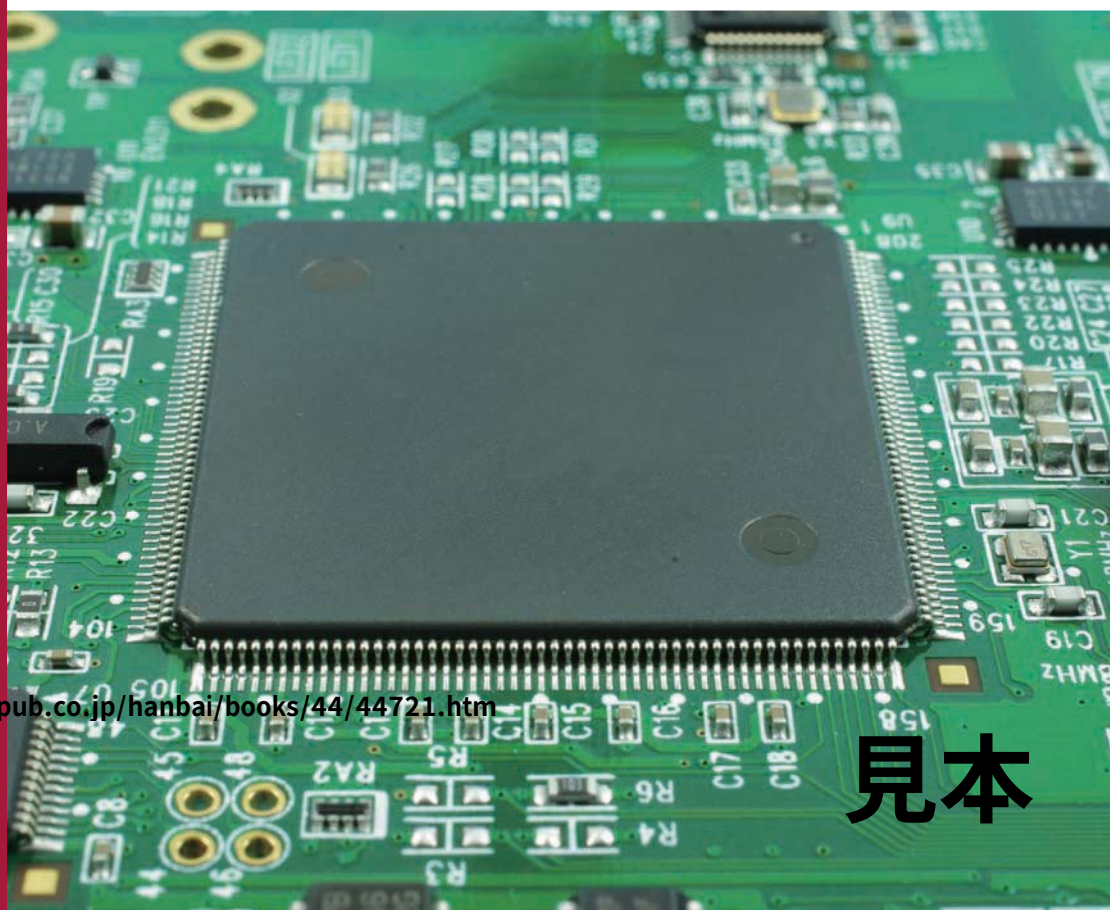
動くメカニズムを図解＆実験！

Linux超入門

My Linux
シリーズ

コンピュータの性能を100%引き出すために

宗像 尚郎
海老原 祐太郎
[共著]



CQ出版社

見本

CQ出版社

はじめに

ヘルシンキ大学の学生だったリーナス・トーバルズ氏が，Linuxカーネルの原型となるUNIX互換オープンOSのアイデアをメーリング・リストに発表したのは，1991年の夏です．当時はまだ商用インターネット・サービスが始まっていなかったので，Linuxカーネルの開発は，開発に参加したいメンバ同士で大量のフロッピ・ディスクを郵送し，回覧するという地道な方法でスタートしました．それから4半世紀も開発が継続し，のちに大きく発展することになることを当時誰が想像できたでしょうか？　Linuxは，インターネット通信で使われるTCP/IPプロトコルへの対応に優れていました．Linuxの開発時期が，インターネットの爆発的な普及のタイミングと重なったこともあり，当時のWebサーバやメール・サーバ向けなどに多く採用されていきました．

このような経緯を反映してか，現在入手可能なLinux関連の技術書籍や専門月刊誌の多くは，いわゆるプログラマではなく，企業などで情報システムを構築運用するIT系技術者を主な読者対象としてきました．これらの書籍では，OSのインストール，ユーザ・アカウントやアクセス権の設定，RAIDの構築などストレージ・デバイスの管理，スクリプトによる自動バックアップなど，Linuxサーバの保守管理機能が中心的に取り上げられていました．また，最近のスマートフォンやPC環境向けのアプリケーション開発現場では，実行環境に依存した記述をより積極的にプログラム実行環境を隠ぺいする，C++やJavaなどのオブジェクト指向プログラミング技法がよく使われます．これらはプログラミング環境が提供するライブラリを利用するので，カーネルの機能を直接利用しません．そのため，Linuxカーネルをブラックボックスとして取り扱うことに特別な違和感はなく，実際使うだけなら特に困ることもありませんでした．

ところが，SoCに組み込まれたチップ特有の周辺機能を活用する場合や，組み込み機器制御用のハードウェア制御プログラムを開発する場合は異なります．CPUの処理能力が潤沢とは言えない組み込み機器向けアプリケーションでパフォーマンスを追求するには，Linuxカーネルの機能を呼び出すカーネルAPIを使ったプログラム記述が今でも欠かせません．このようなコーディングを行うには，通常のC言語プログラミング技法の理解に加えて，複数のプログラムを並列動作させるスレッド・プログラミング，スレッド間の動作を調停させるプロセス間通信や，同期の使いかた，ネットワークを利用するソケット通信など，カーネルが提供する機能を直接利用する方法を理解する必要があります．

CPUの動作速度，メモリやHDDの容量や速度，各種周辺機器を接続するPCI，USB，SATAなどのインターフェースは，過去20年で劇的に進化しました．Linuxカーネルはその都度，最新規格にいち早く対応してきました．例えば，HDD上のファイルからデータを100バイト読むようにアプリケーションからリクエストした場合を考えてみます．するとカーネルはHDDからの読み込み単位1セクタ分512バイトのデータを読んで512バイト分

のデータをメモリ上に保存しておきます．これは，次に続きの100バイトのリード要求が来たとき，あらためてHDDをアクセスするのではなく，メモリ上でデータを高速に返すための工夫です．さらにアプリが大きな連続データを読んでいると判断した場合にはカーネルが独自の判断で複数セクタ分のデータを投機的に先読みすることもあります．このようなカーネル内部のインテリジェンスは，アプリケーションからは見えません．OS上でファイルを読んだ方がスループットが上がるという結果だけが見えることになります．

本書はこのような外から見えないカーネル内部の動きに注目します．カーネルにはいろいろな最適化のしくみが組み込まれていますが，ユーザの指示と無関係に自律的に動いているわけではありません．あくまでアプリケーションからの要求を細かく分析した結果に基づいた最適化を行っています．そのため結果としてユーザがカーネルに対してどのようなリクエストを投げるかによって性能最適化の有効性が変わる可能性があります．カリカリに性能を最適化したい組み込み機器などでは，カーネルの気持ち（＝内部最適化処理）を理解してプログラミングすることが重要です．

本書ではカーネルAPI呼び出しの最適化のヒントとなるよう，現時点で最新のカーネル実装を参考にしながら，カーネルAPIが呼ばれたときのカーネル内部の動作を図なども使ってわかりやすく解説すること，さらに実際に実験プログラムを動かしながらカーネルの動きを確認することで，肌感覚でカーネルのインテリジェンスを体験することを目指しました．Linuxカーネルの主要な機構に対して，カーネルが内部でどのような制御を行っているのかを紹介します．またその上で，実際にカーネル機能を確認するためのテスト・プログラムを使った実験を行いました．

各章はそれぞれ独立した記事になっているので，興味のあるトピックから読んでいただいてもよいですし，全体像をつかむために順番通り読んでいただいても問題はありません．Linuxカーネルは，今やソースコード行数が2,000万行を超える膨大なプログラムです．カーネルの機能について網羅的に詳細な説明を行うのは現実的ではありませんが，ここで取り上げているトピックを把握しておけば役に立つことが多いと思います．

Linuxカーネル内のインテリジェント処理がブラックボックスの場合，アプリケーション・プログラムからカーネルAPIを利用してカーネルに処理をお願いしたら，結果が戻ってくるまでひたすら待つしかなかったのですが，本書の内容を理解すればカーネルがどのように要求をさばいているのかを感覚的に共感できる，高感度プログラマとなれるでしょう．これはクールで高性能なシステム開発を目指すエンジニアにとって，大きな武器となります．

2016年3月　**宗像 尚郎**

基本をおさえておけばメカニズムも長所短所も合点！

第1章 Linuxカーネルの設計思想

Linuxは多くの組み込み機器で活用されるようになっていますが，本来は機器制御用途向けに開発されたソフトウェア（OS）ではありません．

Linuxカーネルの設計思想を理解しないまま，組み込み機器向けのいわゆるRTOS（Real Time Operating System）で構築されたソフトウェア資産をLinux環境に移植しようと試みて，まったく期待した性能が出せない事例はいくらでもあります．

ここでは，Linuxがさまざまなプログラムの集まりであり，スケジューリングなどを調整することで，全体として効率良く動こうとするOSであることを説明します（**図1**）．厳密な時間管理が必要なハード・リアルタイム制御を求めて，ユーザ・プログラムがOSの処理を止めるような使い方をするものではありません．**タスク・スケジューリングはカーネルに任せる**というのがLinuxの基本的な考え方です．

その1：ユーザは直接ハードを制御できない

● マイコン用リアルタイムOS：厳密に時間を守ってハードウェア制御

はじめにOSの概念について説明します．精密な実時間制御が要求される機器制御の世界では，OSを使わないのが普通でした．プログラムの規模が大きくなるにつれて，

- キー/リモコン入力受け付け
- 計算処理
- アクチュエータ制御
- 表示制御

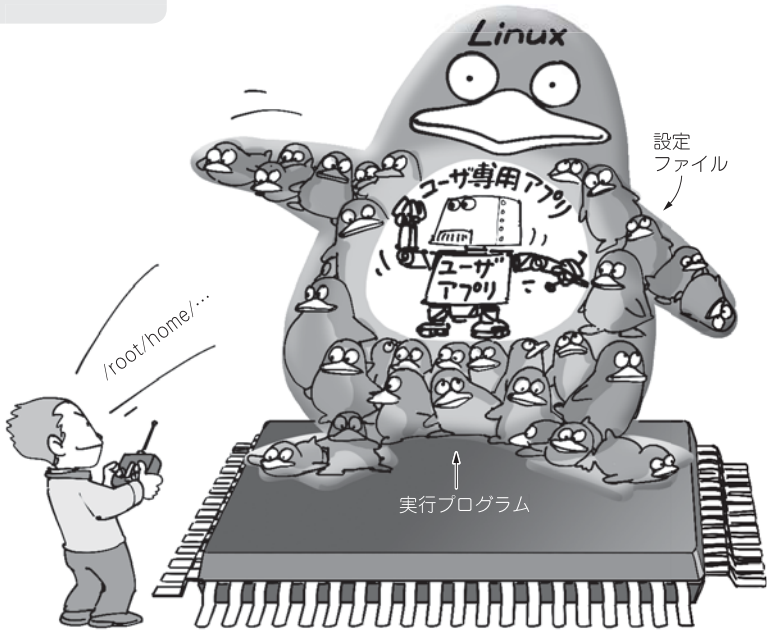


図1　Linuxは全体として最適に動こうとするOSなので，厳密に動きを管理することはできない
RTOSとは全然違って，ハード・リアルタイム制御には基本向かない

など複数の処理を行う必要が出てきました．機能別にタスク分割を行ってソフトウェアを開発するようになると，組み込み機器制御プログラミングの世界にリアルタイムOS（=RTOS）が導入されます．

RTOSは割り込み受け付けルーチン（ISR）から一定の時間内に特定の処理を完了させるなど，決定論的なプロセスの実行順序設計による実時間（＝リアルタイム）性の管理ができる構造になっています．

RTOS本体は小さくてサクサク動くプログラムです．マイコン・メーカなどがチップごとに専用のRTOSを提供（販売）したり，ユーザの要求によってはカスタマイズも行ったりしていました．

● 汎用OS Linux：ソフト屋向け…制御はOSに任せてハードを意識しない

Linuxはワークステーション用OSであるUNIXのクローンです．UNIX互換のアプリケーション-OSカーネル間インターフェースをもった本格OSとして開発されました．OSカーネルとアプリケーションの境界は**図2**に示すPOSIX（Portable Operating System Interface）と呼ばれるインターフェースで規定されています．

ファイル・システムやネットワーク・プロトコルなどの機能がOS内部に組み込まれた，**図3**に示すような全部入りモノリシック構造が採用されています．アドレス・マッピングや割り込み番号の割り付けなど，機器ごとのハードウェア実装の違いを意識せずに，どのマシンでも同じアプリケーションを実行できる統合的

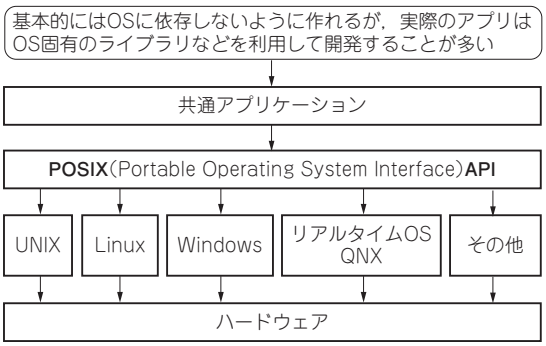
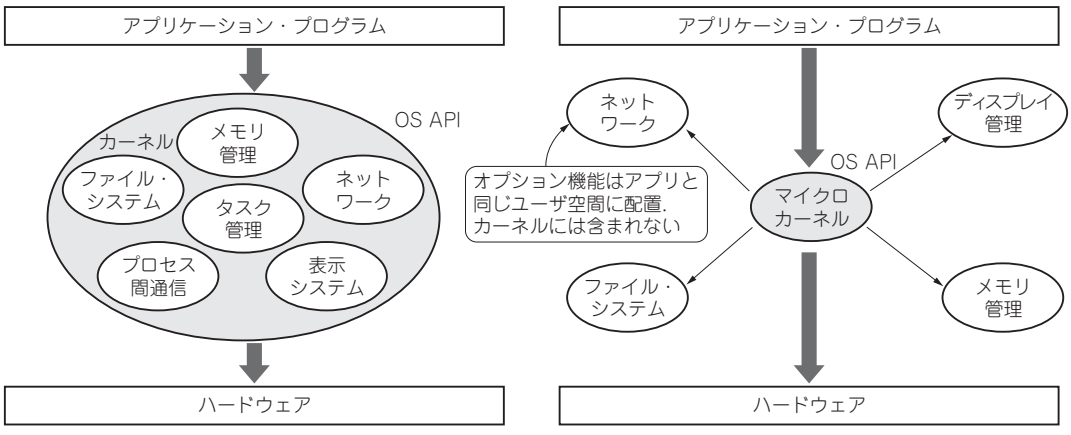


図2　OSに関係なく動くアプリを作るために定められたソフトウェア・インターフェース…POSIX
LinuxはPOSIX準拠OS



（a）モノリシック・カーネル方式
（=全部入りカーネル）…Linuxが採用

（b）マイクロカーネル方式
（=軽量タスク管理+拡張機能）

図3　OSには全部入りタイプとコンパクト・タイプがある
Linuxは全部入り

第2章 Linux起動のしくみ

ボード用Linuxとパソコン用Linuxの起動条件の違い

● 動作環境…パソコン用はBIOSでチップの違いを吸収してもらっている

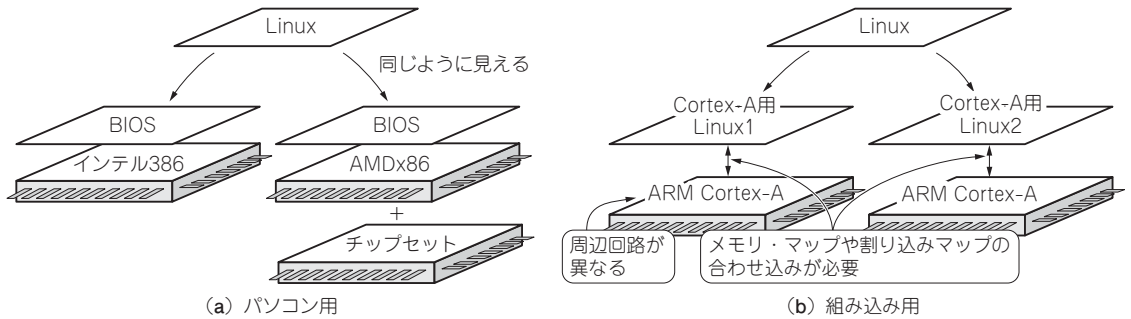
Linuxは当初i386搭載パソコン上で開発されました．Linuxカーネルは，**図1**に示すように，パソコンのBIOS（Basic Input/Output System）プログラムによって初期化済みのマザーボードから起動するように作られています．

BIOSはマザーボードのチップセットや割り込みマップの違いを吸収して，抽象化されたPCハードウェアをOSに見せます．

一方，組み込み機器で多く使われるARM LinuxボードにはBIOSはありません．メモリ・マップや割り込みマップはボードごとに異なり，抽象化されないままの生の状態でOSに渡されます．

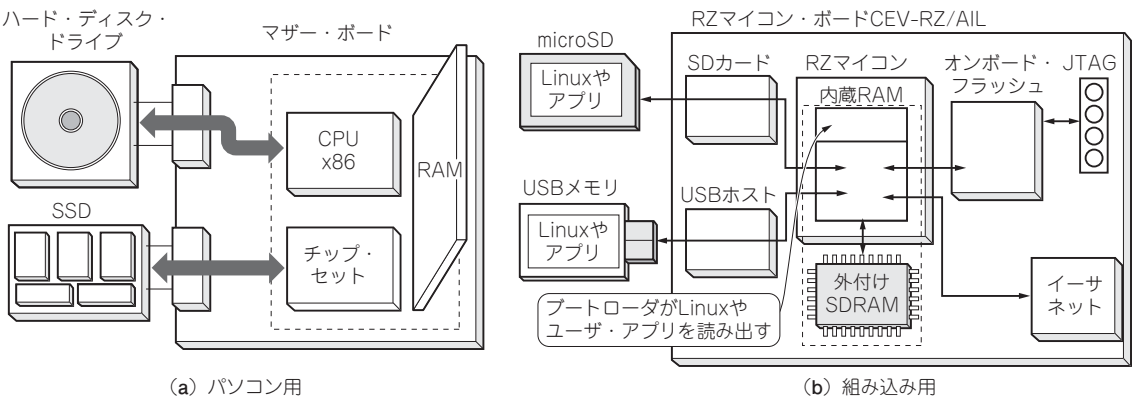
● 起動デバイス…ボードではハード・ディスク以外もよく使う

Linuxパソコンは，通常は**図2**に示すようにHDD/SSDから起動します．



（a）パソコン用　　（b）組み込み用

図1　組み込みLinuxがパソコンLinuxと違う点①…メモリ・マップや割り込みマップの合わせ込みが必要



（a）パソコン用　　（b）組み込み用

図2　組み込みLinuxがパソコンLinuxと違う点②…ハード・ディスクやSSDじゃない起動デバイスを使うことも多いのでセキュリティ対策が必要

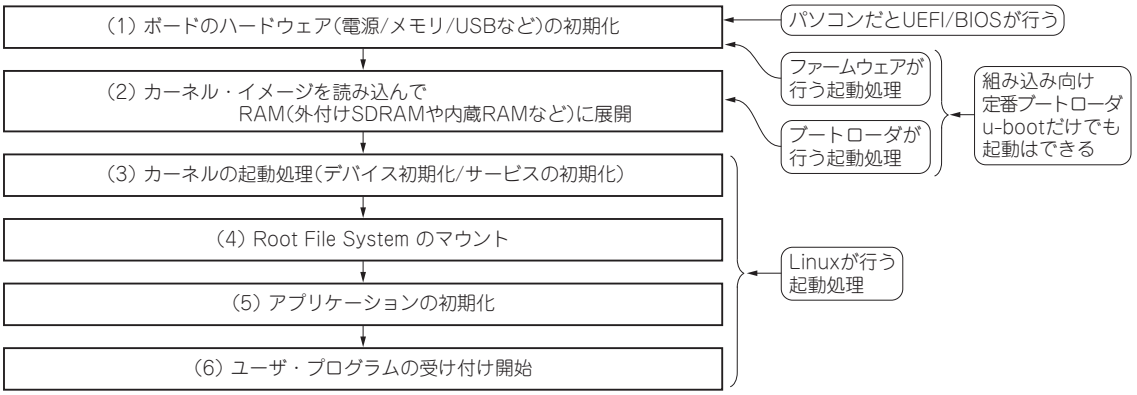


図3　組み込みLinuxの共通起動手順

組み込み機器では，開発時にはネットワークやUSBメモリ，SDメモリーカードから，製品組み込み時にはオンボード・フラッシュ・メモリなどから起動します．

組み込み機器の場合には，起動するカーネルのセキュリティを管理するために，セキュアブート機能への対応などボードごとに起動シーケンスの管理が必要になります．

共通の起動手順

ここでは**図3**に列挙したような組み込みLinuxシステムで共通の起動手順を紹介します．

この中で(1)はファームウェアが，(2)はブートローダと呼ばれるプログラムが受け持ちます．

(3)～(6)はLinuxカーネルの起動シーケンス内で処理されます．

● 初期化はディストリビューションによって異なる

UbuntuやFedoraなど，Linuxのパッケージングを行っている団体をディストリビュータといいます．実際のLinuxのパッケージをディストリビューションといいます．

カーネル起動後の初期化シーケンスは，CPUアーキテクチャには依存しませんが，Linuxのディストリビューションによって方法が異なります．

● 定番起動スクリプトSysVinit

本稿では`SysVinit`という現在多く使われている起動スクリプトの記述方法をベースに指定方法などを紹介します[注1]．

以下，組み込みシステム上でLinuxアプリケーションを動作させるための起動処理について順番に説明していきます．

手順1＆2：電源/メモリ/USBなどの初期化＆カーネル・イメージのRAM展開

● ファームウェアとブートローダがカーネル起動処理前の準備をする

ファームウェアはボードの電源が投入された直後から動き出すプログラムです．**図4**のような，Linuxカーネルを起動するための準備の前処理を行います．

CPUのバス・ステート・コントローラやDRAMのアクセス・タイミングなどの初期化は，カーネルを起動する前に設定します．

注1：SysVinitに変わる新しいsystemdという起動手順の記述方法がFedora15やArch Linuxなどで導入されていますが，ここでは定番の起動スクリプトSysVinitについて説明します．

第3章 仮想メモリ・アクセスのしくみ

本稿では，Linux上で物理メモリを抽象化して保護するための，メモリ管理（MM；Memory Management）機能について解説します（**図1**）．

Linux流！ 仮想アドレス空間のメリット

ワンチップ・マイコンを使ってOSレスでハード制御を行ったり，ITRONなどのRTOS（リアルタイムOS）アプリを作ったりするハードウェア制御プログラマが，Linuxソフトウェア開発を始めるときに困惑するトピックの一つが，仮想アドレス空間ではないでしょうか．

● その1：メモリ容量を最大限効率良く使える

一般的な組み込みシステムの物理メモリ空間レイアウトは，**図2**（**a**）に示すように，実ボード上のメモリICの配置そのものです．メモリ容量はシステムに搭載された物理メモリ・デバイスのサイズ以上にはなり得ません．

しかしLinuxのプログラムは，**図2**（**b**）に示すようにターゲット・ボード上の物理メモリ・サイズとは関係なく，32ビット版カーネルなら，32ビットで指定可能な最大サイズの4Gバイトのメモリ空間が利用可能であると仮定して動作します．この仮定アドレスを，仮想アドレス空間と呼びます．

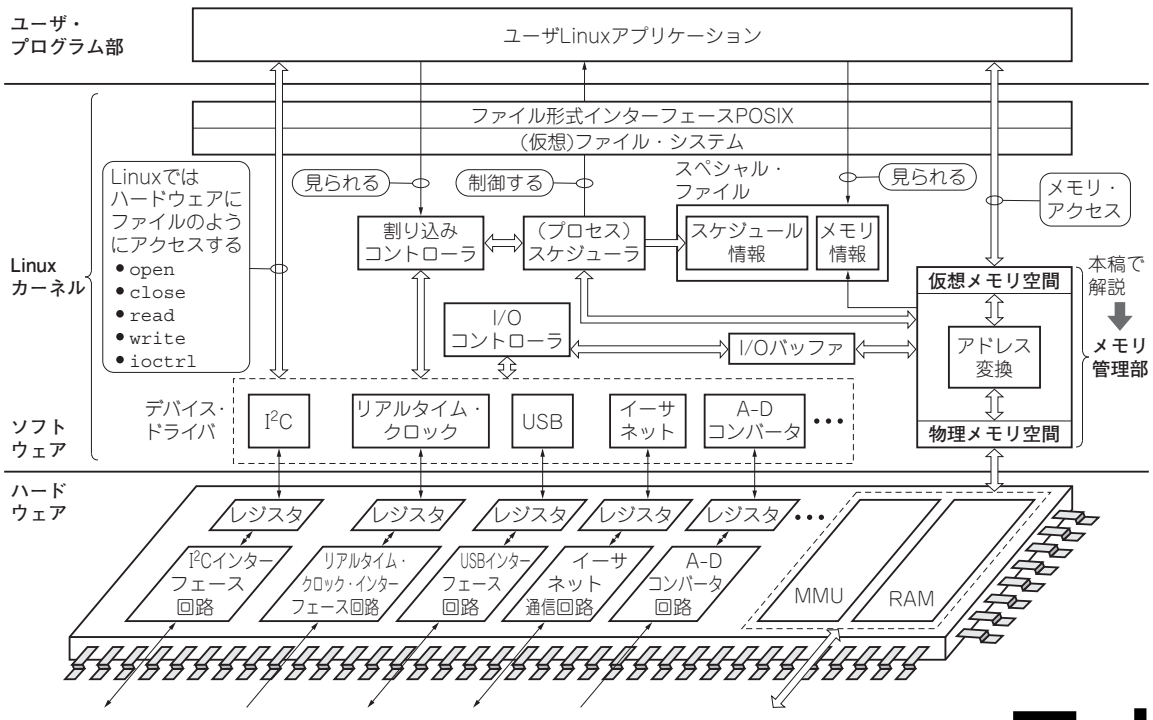


図1　本稿で解説すること…Linux上でRAM（メモリ）アドレスを簡単に扱える仮想アドレスのメカニズム

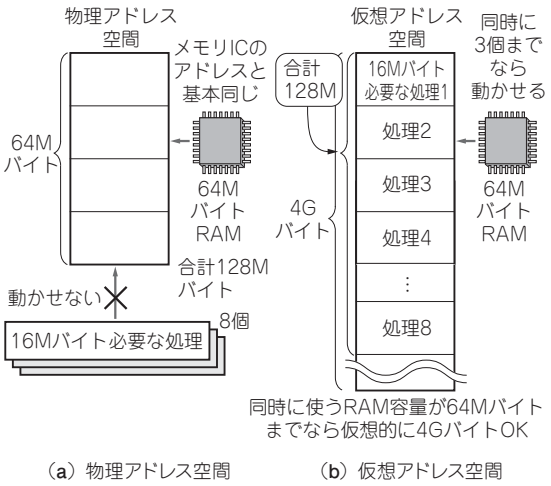


図2　仮想アドレス空間のメリット①：メモリ容量を最大限効率良く使える

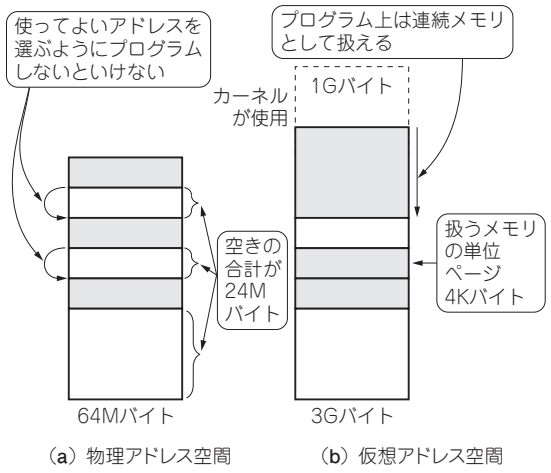


図3　仮想アドレス空間のメリット②：ポインタ指定するだけで大容量メモリが使える…Linuxプログラム側は今どの番地を使ってよいか管理しなくてよい

Linuxでは，RAMが64Mバイトしか搭載されていないCPUボードでも，仮想アドレス空間にはずっと大きな，例えば1Gバイトのメモリ容量があるかのように振る舞います．こんなインフレ的な考え方はOSレスやRTOSのプログラマにはなじまないかもしれません．しかし，Linuxには「同時に使わないメモリは共有可能と考えることによってメモリの利用効率を高めたほうがよい」という基本的な考え方があります．

▶ OSレス/RTOS流 固定的メモリ割り当て

例えば，RAMを16Mバイト使う処理が8個ある場合にメモリを固定的に割り当てれば16Mバイト×8=128Mバイトが必要になります．RAM容量が64MバイトしかないCPUボードではメモリ・サイズ不足になり動かせません．

▶ Linux流 投機的メモリ割り当て

しかし，このプログラムがもし「同時には一つしか動かない」と仮定できれば，必要なメモリ・サイズは16Mバイトで十分ということになります．前者の固定的割り当てがOSレス/RTOS的な発想とすれば，後者の動的投機的割り当てがLinux流の考え方です．

Linuxでは，8個のアプリが初期化処理でそれぞれ16Mバイトずつを要求した時点ではメモリ不足にはしません．実際にメモリ容量以上のプログラムを実行しようとしたときに初めてメモリ不足エラーの処理が起動しますが，同時に実行するプログラムが3個以下なら問題なく実行できてしまいます．

● その2：ポインタ指定一つで大容量メモリが使える

物理と仮想のもう一つの違いはアドレスの連続性（リニアリティ）です．**図3**（**a**）に示すように，物理メモリで24Mバイトの空きがあるといった場合，それは断片化したメモリを集めて合計すると24Mバイトになるという意味であって，24Mバイトの連続領域が空いているわけではないでしょう．

一方，仮想アドレス空間に対応したLinuxでは，**図3**（**b**）に示すように，ひとつひとつのプロセス[注1]ごとに，物理メモリの空きエリアとは無関係に連続した広大なメモリ空間を見せます．このため，アプリケーション・プログラムは，メモリ配置が非連続になっているかどうかを心配することなく，単純なポインタ操作でデータにアクセスできます．

注1：Linuxのプログラムの単位．

Linuxの性能を生かすにはできるだけ早くOSに権限を返すのが基本

第17章 割り込み処理を短時間で済ませるコツ…必要な処理をまず済ませる

Linuxの割り込み処理

■ ハード制御では特に重要！ 割り込み時間は短く！

割り込み処理はシステムのうち最優先で処理されるため，ほかのすべての処理が中断されます．Linuxに限らず，すべてのコンピュータ・システムでいえますが，割り込み処理ルーチン（interrupt service routine；ISR）は十分に短い時間で処理しなければなりません．

● 割り込み処理を短時間で終わらせるメカニズム

Linuxでは割り込みが発生すると，プログラムの現在の実行番地（PC）やCPU内レジスタの値をスタックに退避し，割り込みルーチン（トップ・ハーフ）が呼ばれます．

割り込みサービス・ルーチンが十分に短い時間で処理が終わるのであれば，トップ・ハーフのみで十分です．しかし，割り込みサービス・ルーチンの処理時間が長くかかりそうな場合は，割り込みルーチンを二つに分けて処理します．サム・チェックの計算やヘッダの確認など付随的な処理をボトム・ハーフに追い出し，トップ・ハーフを短く実装します．

例えばデータを受信する際に，周辺LSIからメイン・メモリに受信データを読みだす部分はトップ・ハーフで実行する必要がありますが，付随処理は必ずしもトップ・ハーフで実行する必要はありません．これらはボトム・ハーフに追い出せます．

本稿では，ボトム・ハーフを実現するしくみとしてタスクレット（taslket），ワーク・キュー（work queue），スレッド型割り込み（interrupt service thread；IST）の三つを解説し，それぞれを使った場合の処理時間を実験して測ってみます．

● 測定方法

プログラムは第16章と同様，TPU1（タイマ・パルス・ユニット1）を使った割り込み遅延時間計測プログラムです．これを改造して使います．TPU1とTPU0は完全に同期して動いており，トップ・ハーフまでの到達時間をTPU1，ボトム・ハーフ到達までの時間をTPU0を使って計測し，**写真1**に示すようにオシロスコープに波形表示します．

急がない処理を後まわしにする三つのしくみ

ボトム・ハーフを実現するしくみには

- タスクレット
- ワーク・キュー
- スレッド型割り込み

の三つがあります．

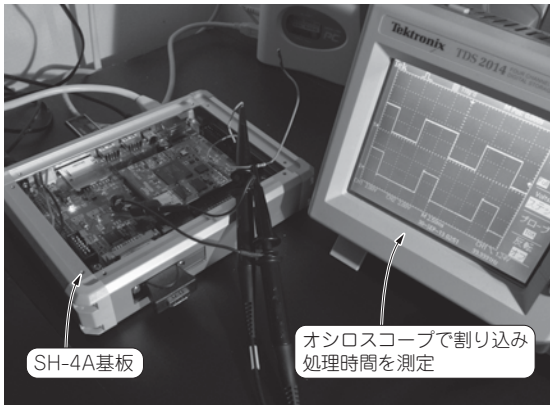


写真1　SH-4AボードのTPU1ピンで
割り込み処理時間を測定

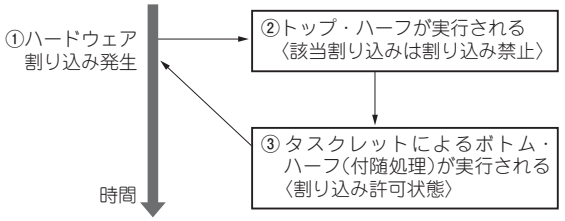


図1　しくみ1：タスクレット…トップ・ハーフ実行中に割り込み許可を出せる

● しくみ1：タスクレット… 割り込みの付随処理を追い出して処理時間を短くする

図1にタスクレットの概念を示します．トップ・ハーフ実行直後にタスクレットによるボトム・ハーフ関数がコールされます．

トップ・ハーフ実行期間は該当割り込みが割り込み禁止となっていますが，タスクレットによるボトム・ハーフ実行中は割り込み許可されています．すべての割り込み処理をトップ・ハーフに記すのではなく，タスクレットを用いてボトム・ハーフに付随処理を追い出してトップ・ハーフの実行時間を短くします．効果としては，次の割り込みを取りこぼす確率を下げることが期待できます．

● しくみ2：ワーク・キュー… スリープ可能な割り込み処理

図2にワーク・キューの概念を示します．ワーカ・スレッドは，カーネル内部で普段はスリープしてイベントの発生を静かに待っている小さなタスク（スレッド）です．

（1）割り込みが発生するとトップ・ハーフが呼ばれる
（2）トップ・ハーフ内でワーク・キューをスケジュールすると，スリープしていた専用のワーカ・スレッドが起床する
（3）プログラムは割り込み発生前の元の場所に復帰する
（4）しばらくして汎用ワーカ・スレッドの実行順序がやってくると，キューに積まれた仕事（ワーク・キュー）を順次処理する．具体的には関数と引き数がペアになってキューイングされているので，これを順次実行する．ここでボトム・ハーフがコールされる

（2）で専用のワーカ・スレッドがスリープから起床するとしていますが，実際にはこの段階ではワーカ・スレッドのステータスがSLEEPからRUNに代わるだけなので，実際にワーカ・スレッドに処理が移るのは少々

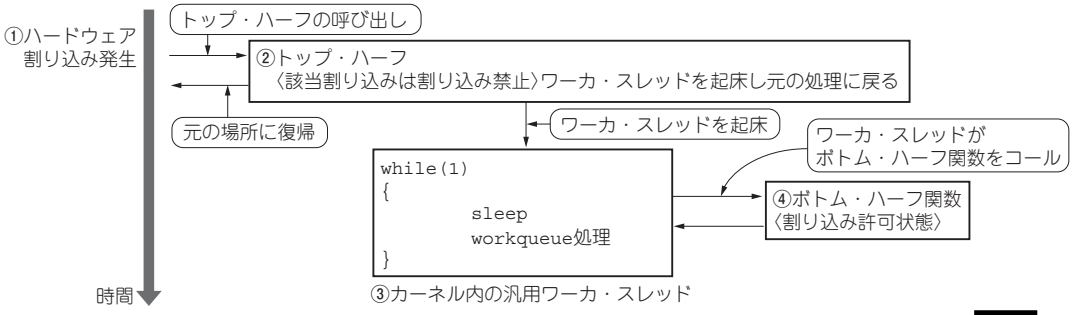


図2　しくみ2：ワーク・キュー…スリープ可能な割り込み処理

第20章 Appendix

便利で高速起動！定番ブートローダU-Bootを使う

第20章に引き続き，ブートローダの内容です．U-BootはGPLでライセンスされている多機能なブートローダです．多くのARM評価基板でもU-Bootを標準採用しています．本稿では定番U-Boot（正確にはDas U-Boot）で用いられるuImage形式について実験します．

実験内容

● U-Bootで使われる形式の起動時間を調べる

図1に実験の構成を示します．第20章で作ったSH-4Aマイコン搭載基板用のブートローダを利用し，外付けのフラッシュROMからCPUを起動し，圧縮されたLinuxカーネルのバイナリを呼び出します．以下の4種類について調べます．

- zImage形式
- 非圧縮カーネル
- uImage形式（gzip圧縮）
- uImage形式（非圧縮）

これらのロード時間やカーネルのファイル・サイズを調べます．その結果を**表1**に示します．

U-Bootで使う圧縮ファイルuImage

● 中身がわかるヘッダ付きの圧縮ファイル

定番ブートローダU-Bootで用いられるファイル形式がuImageです．uImageは，**図2**で示すようにシンプルな64バイトのファイル・ヘッダを圧縮したカーネル・データに付加したものです．

図1　定番ブートローダU-Bootで使われる圧縮ファイルuImageの起動時間を測った

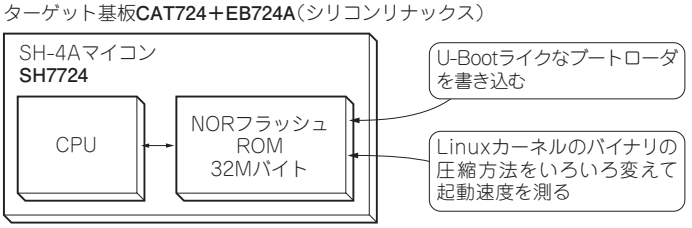


表1　ロード時間とカーネル・サイズを測定した

圧縮方式	ロード時間［ms］	zImageとの比較
zImage（gzip）標準	1,792	100%
非圧縮（vmlinux.bin）	613	34%
uImage（gzip）	562	31%
uImage（NONE）	1,792	100%

（a）ロード時間

圧縮方式	サイズ［バイト］	zImageとの比較
zImage（gzip）標準	2,531,360	100%
非圧縮（vmlinux.bin）	4,826,540	191%
uImage（gzip）	2,514,322	99%
uImage（NONE）	2,531,424	100%

（b）カーネルのファイル・サイズ

ISBN978-4-7898-4472-7

C3055 ¥3200E

CQ出版社

定価：本体3,200円（税別）

9784789844727

1923055032002

見本