QNX ソフトウエア システムズ 株式会社
〒 102-0075
東京都千代田区三番町六番地
三番町 KB-6 ビル 四階
電話：03-3511-6450
ファックス：03-3511-6451
E メール：japan_info@qnx.com
ウェブサイト：www.qnx.co.jp

リアルタイム OS それとも Linux?

-現実的なアプローチ-

Paul Leroux
QNX Software Systems Ltd.
テクノロジー アナリスト

組み込みシステムの設計者にとって、Linux はジレンマの元です。Linux を採用すれば、層の厚いプログラマ人口や豊富なソースコード、業界標準の POSIX API が使えます。しかし、標準の Linux カーネルでは、多くの組み込みデバイスで不可欠な、応答時間の保証やマイクロ秒単位の遅延といった、真のリアルタイム機能を実現することができません。

この問題は、Linux の汎用アーキテクチャに由来しています。例として、プロセスのスケジューリングを見てみましょう。リアルタイム OS（RTOS）が使用するプリエンプティブな優先度ベースのスケジューラとは対照的に、Linux では、すべてのプロセスに「公平な」実行機会を与えるポリシーが実装されています。そのため、優先度の高い、タイミングが極めて重要なプロセスが、すぐに CPU にアクセスできるとは限らなくなります。事実、Linux では、時により OS が優先度の高いプロセスに割り込み、より優先度の低いプロセスに CPU 時間を割り当てることがあります。

また、標準の Linux カーネルは、プリエント可能ではありません。優先度の高いプロセスでも、カーネルコールをプリエンプトすることは絶対にできません。そのため、カーネルコールが優

先度の低いプロセスに呼び出されたものであっても、コール全体が終了するのを待機しなければなりません。これでは、重要なイベントが、短くしかも予測可能な時間内で確実に処理されるようなシステム設計を行うことは、不可能とまでは言わないまでも、非常に困難です。

念のために言っておくと、このスケジューリング モデルを Linux の欠陥であると見なすのは間違いです。Linux のモデルは、デスクトップやサーバー アプリケーションなどで全体的な処理能力を向上させるには、非常に有効なものです。ただ、ネットワーク ルータ、工場用ロボット、継続的に使用されるメディア アプリケーションなど、確実な反応が要求される環境には適していないということです。

このような状況の中、Linux のリアルタイム性能での欠点を補う製品が出現しつつあります。これらの製品の中には、商用目的でソフトウエア ベンダが開発したものもあれば、オープンソースの研究プロジェクトもあり、中には、両者の協力から生まれた製品もあります。しかしながら、どのアプローチも、「標準」として生まれたものではありません。実際、これらの製品のアプローチには、POSIX/Linux の標準プログラミング モデルから大幅に逸脱しているものがあります。

Linux の外に実装されるリアルタイム性

たとえば、多くのベンダは、「リアルタイム Linux 」というかぎカッコつきの製品を提供しています。これは、リアルタイム カーネル上に、Linux を一つのタスクとして実行する形を取るものです（図１参照）。予測可能なスケジューリングを必要とするタスクは、このプリエンプト可能なカーネルにより、Linux よりも高い優先度で実行されます。したがって、これらのタスクは、実行が必要になると Linux をプリエンプトし、仕事が終わるまで CPU を Linux に解放しません。

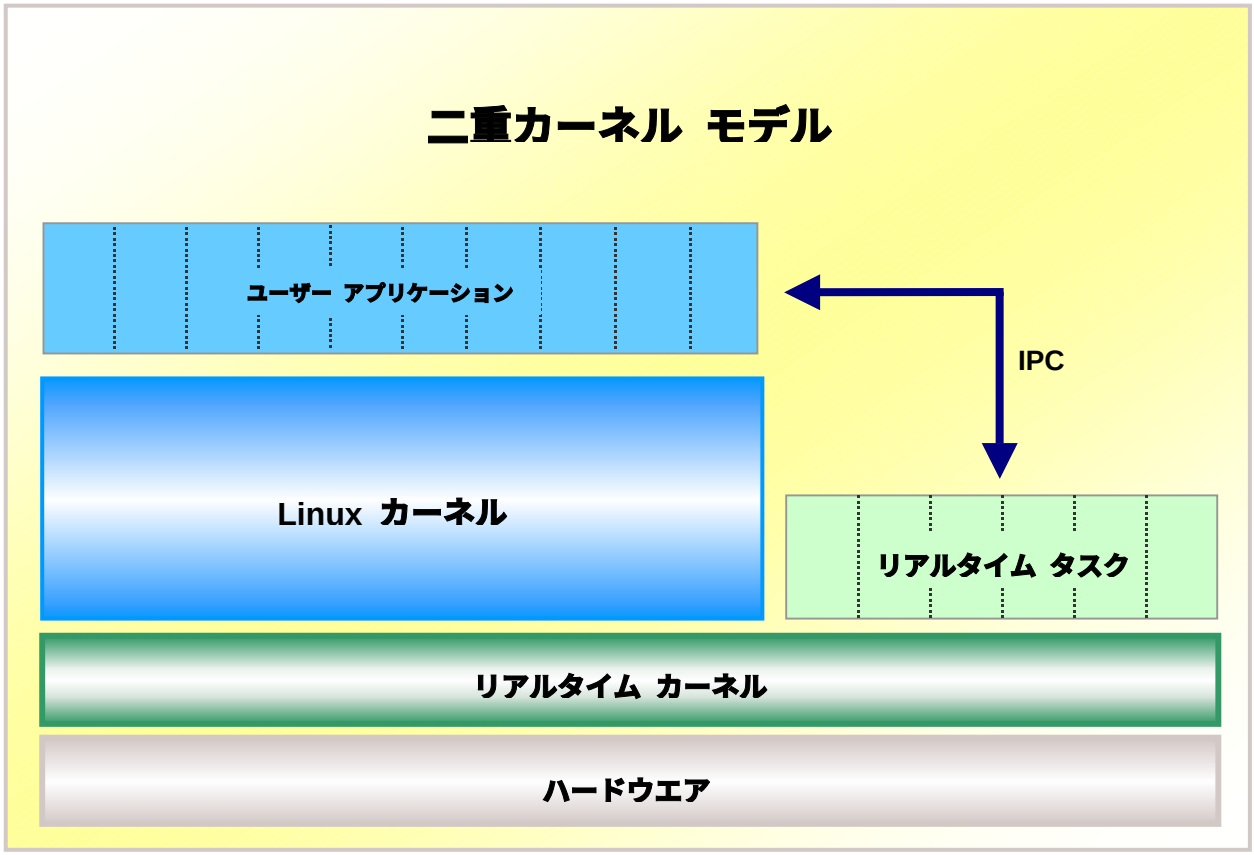

図 1：二重カーネル モデルでは、Linux は、別個のリアルタイム カーネル上で一番低い優先度で実行されます。

この二重カーネル モデルでは、リアルタイム カーネルが常にハードウエア割り込みを優先的に取得します。割り込みがリアルタイム タスクに対応するものとしてフラグされていると、カーネルはそのタスクが実行されるようスケジューリングを行います。割り込みがリアルタイム タスク対応としてフラグされていなければ、それは Linux に渡されて処理されます。このアプローチの優れている点は、リアルタイム処理が、Linux 環境で動いているアプリケーションとは関係なく実行できるということです（もちろん、リアルタイム カーネルに CPU サイクルが奪われるということは抜きにしてですが。）それでも、このアプローチには、いくつかの欠点があります。

プログラミングの二度手間

リアルタイム カーネルで実行されるタスクは、ファイルシステムやネットワーキングなど、既存の Linux サービスを最大限に活用することができません。実際、リアルタイム タスクが*いかなる* Linux サービスを呼び出しても、Linux のプロセス処理にリアルタイム性が無いという、同じプリエンプトの問題に直面します。

このため、同じサービスがすでに Linux で存在している場合でも、リアルタイム カーネル用に、新しいドライバとシステム サービスをプログラミングしなければなりません。このようなドライバの既製品を売っているベンダは少ないので、通常、Linux プログラマは、不慣れな API を使って自分で一からドライバを記述することを余儀なくされます。

脆弱な実行環境

リアルタイム カーネルで実行されるタスクは、通常の非リアルタイム Linux プロセスに提供される堅牢な MMU 保護環境の恩恵を受けることができず、保護のないカーネル スペースで実行されます。その結果、リアルタイム タスクに無効な C ポインタなど、一般にありがちなコード エラーが含まれていると、致命的なカーネル障害を引き起こす原因となります。非常に高い信頼性が必要とされることが多いリアルタイム システムにとっては、これは由々しき問題です。

移植性の低さ

二重カーネルのアプローチでは、リアルタイム タスクは、Linux プロセスとはまったく無関係であり、スレッドとシグナル ハンドラは、POSIX API の小さなサブセット、あるいは、標準とは異なる API で記述されます。したがって、既存の Linux コードとアプリケーションをリアルタイム環境に移植することは非常に困難です。

もっと面倒なことに、二重カーネル アプローチは、ベンダごとに使っている API が異なります。特定のベンダのリアルタイム拡張機能用に記述されたリアルタイム タスクは、ほかのベンダのそれでは使えません。組み込みデバイスのメーカーは、幅広くサポートされている Linux の API を使いたくとも、競合するベンダごとに違う「スタンダード」のうち、どれか一つを選ばなければなりません。

Linux *のアプリケーションとドライバの非リアルタイム性*

Linux プロセスはリアルタイム カーネルで実行されないので、予測可能なタイミングでの動作は期待できません。すべてのプロセスは Linux の「公平な」アルゴリズムでスケジューリングされます。

設計オプションの限界

すでに言及したように、リアルタイム カーネルがサポートする API は、標準 POSIX および Linux が提供するサービスのサブセットしか提供しません。したがって、Linux あるいは成熟した RTOS のどちらを使った場合よりも、設計オプションが少なくなります。

「ピュアな」リアルタイム Linux?

二重カーネル アプローチの様々な欠点を考えると、Linux そのものをリアルタイムにしてしまったほうがよくないだろうか、と考えられます。これはそれほど難しいことではないはずです。少なくとも一つのベンダは、ハードなリアルタイム性能を可能にする「ピュアな」Linux カーネルを実現したと主張しています。簡単に言うと、このベンダは、Linux の SMP ロッキング機構を単一プロセッサで使用し、カーネルをプリエンプト可能にしたのです。

このアプローチには、確かに利点があります。プログラマが、標準の Linux カーネルとプログラミング モデルを使用することができるからです。これは、高分解能タイマの追加など、Linux のリアルタイム性を高めるための一般的アプローチと同様、反応性に優れたイベント駆動型のシステムに必要な遅延時間の短縮を実現します。残念ながら、このような低遅延を実現するためのパッチでは、単純なイベント駆動型のシステムに比べて、リアルタイム タスクが大幅なタイムスパンに渡って実行され、システム サービスやプロセス同士の依存性の高い複雑なリアルタイム環境を処理することができません。

たとえば、リアルタイム タスクがデバイス ドライバやファイルシステムなどのサービスに依存するシステムでは、優先度の逆転という問題を、Linux ドライバやバーチャル ファイルシステム（VFS）モデルの中で解決しなければなりません。このため、デバイス ドライバやファイルシステムを構成しているフレームワークをすべて書き換えなければならなくなります。こうしなくては、呼び出したサービスがブロック状態になったときに、呼び出し元のリアルタイムタスクで予期せぬ遅延が発生する可能性があるからです。さらに、既存の Linux ドライバの多

くは、プリエンプト可能ではないという問題があります。このため、リアルタイム性を高めるには、システム中のドライバ全部にプリエンプション ポイントを挿入しなければなりません。これは、一般的な Linux プログラマにとっては、経験したことのない作業です。

いずれにしても、どんなリアルタイム修正が標準 Linux のカーネルに統合されるかは不明です（そもそも、リアルタイム性が、Linux の標準に統合されるかどうかさえ不明です。）結局のところ、ほとんどの Linux システムは、全体的なスループットの向上を目的とし、リアルタイムの予測可能性を犠牲にする現在のカーネルのアプローチに依存しているわけです。Linux をシステムに応用している開発者の多くは、平均反応時間が短い、確実性のより高いカーネルを別に求めていないのかもしれません。

ふたつの世界の特長を統合

上述の欠点を考慮すると、某 Linux ベンダが、リアルタイム システムの開発者ならば、Linux よりもリアルタイム専用に設計された OS を使ったほうがいいかもしれない、と言い出したのは、驚くべきことではありません。しかし、RTOS が以下の条件を満たしていない限り、Linux プログラマは安心できないでしょう。

- Linux の既存ツール、ソースコード、プログラミング モデルが使えること
- Linux オープンソース モデルの特長である、トラブルシューティングとカスタマイズの簡単さが維持されること
- Linux のリアルタイム拡張の問題点を解決すること

互換性の深さ

RTOS は、どこまで Linux のプログラミング モデルをサポートすることができるのでしょうか。この問に対する答えは、大部分のところ、Linux に採用されている POSIX API にかかっています。これらの API は、Unix の伝統に根付いていますが、「実装」ではなく「インターフェイス」の意味であると POSIX 作業委員会が定義しています。簡単に言うと、これらの API は、どの OS あるいは OS アーキテクチャにも縛られていないということです。その結果、RTOS は、Unix に類似したリアルタイム性のない Linux カーネルを採用しなくとも、Linux と同じ POSIX API（および Linux 互換性の微妙な点）をサポートすることができるのです。

QNX® RTOS は、こうした実装に依存しないアプローチの証明です。QNX は、リアルタイム マ

イクロカーネル アーキテクチャ上で、POSIX 準拠の API を提供しています（図 2 を参照）。ここでのポイントは、QNX は、POSIX をアドオン層として実装しているのではないということです。それとは対照的に、QNX RTOS の核である Neutrino® マイクロカーネルは、最初から POSIX のリアルタイム機能（スレッドを含む）をサポートするように設計されています。POSIX（従って、Linux）との互換性は、非常に深いところから始まっているのです。

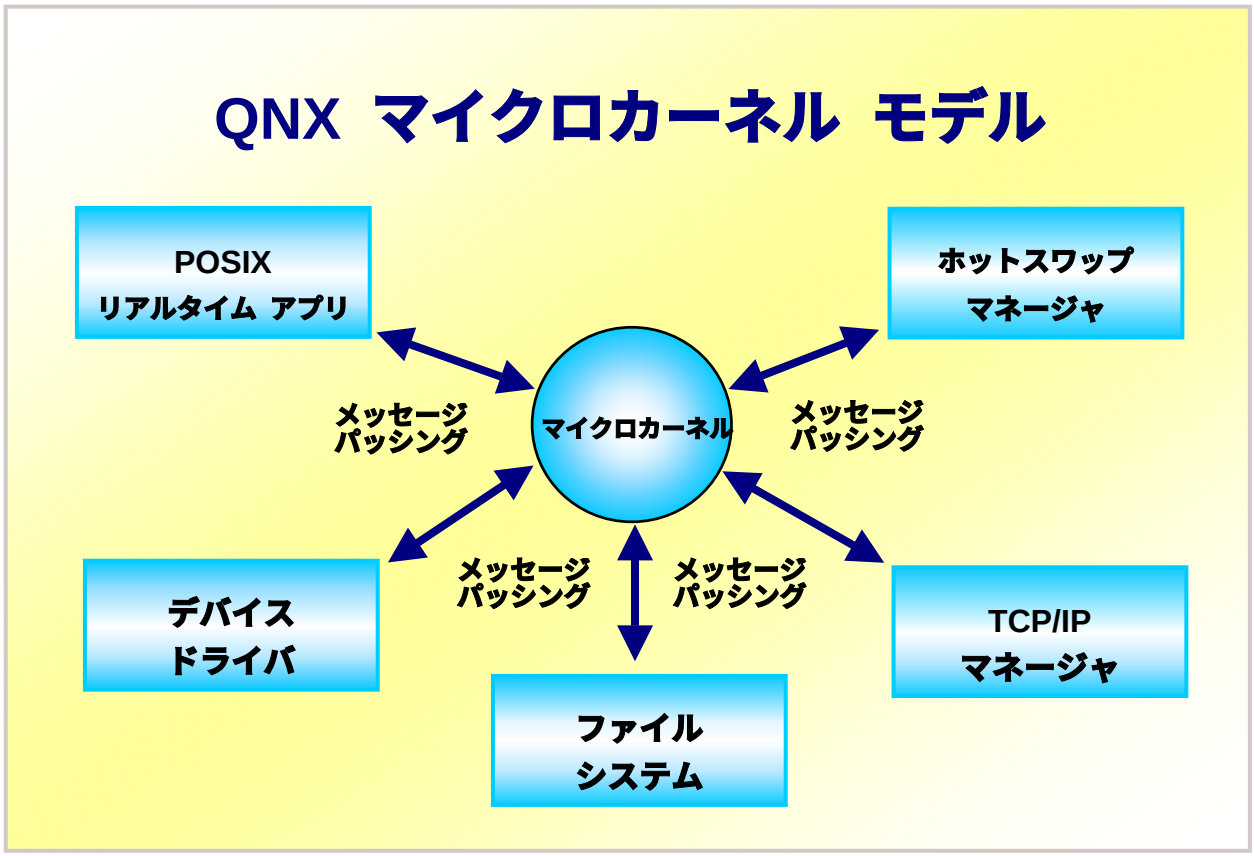

図 2: QNX RTOS では、マイクロカーネルはごく小さなコアサービスのみを処理します。ほかのシステムサービスは、オプショナルな、メモリ保護されたプロセスとして提供されます。

組み込み Linux *：新しい定義*

OEM 業者が組み込み製品への Linux 応用を試みるようになり、API 互換性の問題が驚くほど重視されるようになりました。最近まで、Linux は、主に x86 ハードウエア ベースのウェブ サーバーとワークステーションという、比較的狭い範囲の環境でソフトウエアを作成するプログラマのコミュニティ以外の何物でもありませんでした。開発の焦点が共通しているため、Linux プログラマは、バイナリの互換性、一貫性のある OS カーネル、すぐに移植できるソースコード資産など、Linux に付随する様々な利点の恩恵を受けることができました。

残念ながら、組み込み市場では、こうした「共通の開発焦点」というものが成立しません。これは、組み込み市場の多様性という単純な理由によるものです。組み込みシステムの多くは、用途が特殊なため、アプリケーション、ドライバ、OS サービス、ボード デザインなどを、すべてカスタマイズしなければなりません。多様性を抱える組み込み市場で Linux を応用した結果が、次第に明らかになってきています。それは、組み込み OEM のプログラマが、自分たちの目的に合わせて独自に Linux カーネルを変更するため、標準から逸脱した互換性のないバージョンが存在するようになったということです。Linux は、分裂の危機に瀕しているというより、現実問題として既に分裂が発生しているのです。

このため、組み込み Linux 開発者間の新しい「共通語」、つまり組み込み市場の多様性に対応する一方で、移植性と操作互換性という重要ポイントを提供するような、「組み込み Linux」の定義が必要になってきています。これこそ、「Embedded Linux Consortium (組み込み市場での Linux 応用を促進する、ベンダ中立の産業団体。略称 ELC)」が、まもなく提言される予定の ELC プラットフォーム仕様の中で定義しようとしているものです。ELC の仕様は、組み込み Linux を API のセットとして新しく定義し、仕様に準拠しているかどうかを調べるテスト スイートも作成しています。

このアプローチは、特に新しいものではありません。Unix の世界が分裂の危機に直面したとき、POSIX 仕様が同様のことを行いました。実際、ELC 仕様は、POSIX1003.1 など既存の POSIX 標準に基づくものになります。QNX RTOS は、すでにこれをサポートしています。したがって、QNX RTOS は、組み込み Linux アプリケーションを本質的にサポートでき、同時に、最初から組み込みシステム向けに設計された真のリアルタイム OS としての利点をすべて提供することができるのです。

QNX は、組み込み Linux アプリケーションのランタイム環境として使えるだけでなく、Linux

ホストの GNU ツールを提供しているので、既存の Linux ワークステーションからリアルタイム アプリケーションの作成が可能になります。したがって、開発者の観点から見れば、ツール、API、プログラミング モデルがまったく変わらないので、QNX も Linux もほぼ同様ということになります。Linux プログラマはまた、QNX のセルフホスト開発環境で作業を行うことができるので、QNX 環境への「引越し」も可能です。人気の高い DDD デバッガなど、Linux と共通の GNU ツールを使用していることからも、QNX 環境は Linux 開発者にとっては非常になじみやすいものであるはずです。

本質的なオープン性

それでも、なぜ多くの開発者が Linux を使いたがるのか、ということをよく考えなければ、Linux 互換性も、ただの混乱の元になってしまいます。Linux 人気の本質を理解するためには、オープンソース モデルの利点について考えてみなければならないでしょう。オープンソースにより、開発者は、OS のアーキテクチャを分析した上で自分のコードと統合したり、アプリケーションに特有な要求仕様に対して OS コンポーネントを適合させたりできます。また、自分のプログラムで問題が起きた場合だけでなく、基盤となる OS コードの結果として予期せぬ問題が生じた場合でも、トラブルシューティングの時間が節約できます。つまり、多くの市販 OS の、中身が見えない「ブラックボックス」モデルでは不可能な、ベンダに依存しない、維持管理のすべてを自分の手で行うような開発ができるわけです。

QNX RTOS は、「アクセシブル ソースモデル」および「マイクロカーネル アーキテクチャ」の二つの方法でオープンソースに対応しています。まず、QNX では、QNX RTOS のドライバ、ライブラリ、およびアプリケーションのソースコードを QNX 開発者のコミュニティが自由に使えるようにアクセスを提供しています。ソースのライセンスに関するモデルについては、この記事では詳しく取り上げませんが、ここで指摘すべき重要なことがひとつだけあります。それは、QNX が提供するソースコードは、多くの Linux ソースとは異なり、GPL として提供されるものではないということです。QNX では、独自のライセンス契約に基づいてソースを提供しているので、GPL の場合とは異なり、知的所有権を放棄することなく、自由にソースの改変ができます。つまり、QNX ソースは、オープンであるだけでなく、組み込みシステムの開発者が懸念するオープンソース GPL の知的所有権の問題とも無関係なのです。

第二に、マイクロカーネル RTOS である QNX は、基本的にカスタマイズに関してオープンです。カーネルスペース内で行われる数少ないコアサービス（例：スケジューリング、割り込みの処理）を除いては、ほとんどの OS レベルのサービス（ドライバ、ファイルシステム、GUI

サービスなど）は、カーネルの外側でユーザー スペースのアプリケーションとして存在します。その結果、カスタム ドライバやアプリケーションに特有な OS 拡張機能は、特殊なカーネル デバッガがなくても開発できますし、カーネルの専門家でなくても作業が可能です。実際、ユーザー スペースにある OS 機能の拡張は、Linux プログラマなら誰でも使ったことのあるソースレベルのツールでデバッグができるので、標準アプリケーションと同じように開発することができます。[1]

図 2 に示すように、QNX マイクロカーネル アーキテクチャは、主要な IPC 手段として、メッセージ パッシングを使用します。ユーザーが記述した OS 拡張機能も、QNX が与える OS モジュールと同じように、POSIX ベースの API を使用し、メッセージによって実装されるので、OS のカスタマイズがさらに簡素化されます。これはまた、トラブルシューティングの簡素化にもつながります。なぜなら、メッセージ パッシングにより、複雑なリアルタイム システムを、開発とテストがやりやすい、明確に定義された小さな機能ブロックに分けることが容易になるからです。さらに、メッセージ パッシングは、通信しあうプロセス群の実行を自動的に同期するので、多くの場合、個々のプロセスで複雑な同期サービスをハードコード化したり、デバッグしたりする必要がなくなります。ここで重要なのは、Linux の開発者は、QNX メッセージ パッシングを利用するために特別なコードを実装する必要はなく、POSIX コールでプログラミングを続けることができるということです。プロセス間での実際のメッセージ パッシングは、QNX の C ライブラリによって処理されます。

リアルタイムの利点

POSIX/Linux の API をマイクロカーネル アーキテクチャ上に実装することにより、前述のリアルタイム拡張機能による欠点も解決することができます。

より堅牢なランタイムモデル

モノリシック OS である Linux では、ほとんどのドライバ、ファイルシステム、プロトコルスタックが OS カーネルの中に統合されています。したがって、こうしたコンポーネントの中のたった一つのプログラミング エラーが、致命的なカーネル クラッシュの原因と

[1] マイクロカーネル アプローチには、もう一つ非常に重要な利点があります。多岐に渡る組み込みシステムで、同じカーネルを変更せずに使用できるということです。QNX RTOS では、各 CPU ファミリ間で、まったく同じマイクロカーネルのバイナリを使用することができます。QNX での社内テストを経て、さらに、顧客のミッション クリティカルなシステムでフィールドテストされたものと同じカーネルを使えるので、開発者に安心感を与えます。

なり得ます。QNX では、これらのコンポーネントはすべて個別のメモリ空間の中にあるので、ひとつのコンポーネントでのエラーがカーネルをクラッシュさせたり、あるいはほかのコンポーネントに障害を引き起こしたりすることはありません。したがって、QNX は、本質的に Linux よりも堅牢な環境をリアルタイム アプリケーションに提供することができるのです。言うまでもなく、二重カーネルアプローチで使用されている保護のないリアルタイム カーネルよりもずっとタフな環境です。

統一された環境

QNX では、「リアルタイム環境」「リアルタイムでない環境」という区別がなく、すべてが同じリアルタイム環境なのです。POSIX API のおかげで、リアルタイム アプリケーションは、GUI やファイルシステムといったシステム サービスへ直接アクセスすることができます。同様に、既存の POSIX/Linux アプリケーションも、すぐにリアルタイム的な働きをすることができるようになります。そして、リアルタイム アプリケーションも、非リアルタイム アプリケーションも、同様のメッセージ ベースの環境で動いているので、これらの間での IPC は大幅に簡素化されます。

二度手間を省く

前述のように、二重カーネルのアプローチを取ると、プログラマは不慣れな API を使用してカスタム ドライバをプログラミングしなければなりません。多くの OS では、ドライバを記述するには、使いにくいカーネル用デバッグ ツールが必要です。また、時間のかかるカーネルの再構築や、コストの高いカーネル専門のプログラマも必要になります。

QNX は、この問題をふたつの方法で解決しています。まず、多数のユーザーに使用されている確立された OS として、様々な標準ハードウエア用の市販ドライバをサポートしています。そして、すでに述べたように、QNX はすべてのドライバをユーザー スペースで動かします。したがって、ドライバの開発は、標準的なソースレベルのツールと技術でできるのです。

その他のマイクロカーネル サービス

組み込みシステムのニーズに応えるために設計されたマイクロカーネル OS である QNX は、Linux 開発者に標準 Linux やリアルタイム Linux 拡張機能にはないサービスを提供します。こうしたサービスには、以下のものが含まれます。

ネットワークの透過性

QNX のメッセージパッシングは、本質的にネットワーク分散型です。したがって、ネットワークに特有なプログラミングをしなくても、任意の数のノード間でアプリケーションを分配することができます。プロセッサ同士が、ネットワークを介して対話しているかどうかを「知る」必要はありません。プロセッサ同士は標準 POSIX 関数を使用してメッセージ パッシングを行うだけで、あとは全部 C ライブラリが処理を行います。

耐障害性のあるネットワーク機能

QNX のメッセージ パッシングにより、ネットワークは仮想化され見えなくなります。これにより、アプリケーションは冗長性を持ったネットワークリンクを介して透過的に対話ができます。ひとつのリンクが失敗しても、OS は自動的にトラフィックをほかのネットワークリンクにルーティングしなおします。ネットワークのトラフィックはまた、利用可能なネットワークリンク群を使って負荷バランスを取ることができるので、高い処理能力が実現できます。このサービスも、OS にビルトインされているので、アプリケーションで特別なコードを記述する必要はありません。

小さなメモリ要件

マイクロカーネル アーキテクチャのスケーラビリティにより、QNX RTOS は、Linux よりもずっと小さなランタイム環境を提供することができます。これは、情報アプライアンスなど、生産数の規模が大きい製品で決定的な利点となります。ユニットあたり 2 ドルのメモリ削減が、何万ドルという利益につながるからです。QNX 独自のウィンドウ システムである Photon® microGUI もマイクロカーネルのアーキテクチャを使用しているので、メモリ制限の厳しいデバイスでは、不要な GUI サービスを「アンプラグ」することができます。

相互作用

QNX のように、フィールドで鍛えられた RTOS は、リアルタイム アプリケーションの動作環境として優れたプラットフォームを提供しますが、Linux と RTOS のどちらを選ぶか、という選択は、二者択一を迫られるというものでもありません。実際、Linux と QNX の間ではアプリケーションの移動がすぐに可能なので、それぞれのノードが、目的に一番適した OS を搭載

しながら、すべてのノードで同じ API とアプリケーション コードを共用するような、分散リアルタイムシステムを構築することは、比較的簡単です。もちろん、システム間のコネクティビティや操作性が重要になります。このニーズに応えるため、QNX は TCP/IP やネットワーク ファイルシステム（NFS）、X サーバーなどの技術を完全にサポートしています。その結果、QNX のハードなリアルタイム環境でリアルタイム アプリケーションを実行し、これをネットワークで接続した Linux マシンで出力表示することが簡単にできます。要約すると、QNX と Linux は、単純な「共存」以上の可能性を秘めています。むしろ、このふたつの OS は、それぞれの開発者が同じ API や技能を使って、リアルタイム OS にしろ汎用 OS にしろ、ひとつの OS が単独で実現できる範囲よりもずっと広範な視野でアプリケーション開発を可能にする、相互作用を生み出すことができると言えます。

Printed in Canada. Part Number: 301527

お問い合わせ先：
QNX ソフトウエア システムズ 株式会社
〒 102-0075 東京都千代田区三番町六番地三番町 KB-6 ビル四階
Tel: 03-3511-6450 Fax:03-3511-6451
japan_info@qnx.com www.qnx.co.jp