6

UNIXで構築する リアルタイム制御システム設計についての一考察

A study of the design of real-time control systems built with UNIX

日本アイ・ビー・エム システムズ・エンジニアリング株式会社
第二システム・センター
エンタープライズ・システム部
ICP－ITアーキテクト
中本 雅寛

Masahiro Nakamoto
IT Architect
Enterprise Systems
System Center No.2
IBM Japan System Engineering Co., Ltd.

Linuxは、フリーウェア、オープン・ソースという特長を武器に、インターネット・サーバー分野のみならず、計測・制御システムのオペレーティング・システムへも着実に応用されつつあります。一方、IBMのAIX®はV3.1からリアルタイム・タスクをサポートしていましたが、本当にリアルタイム制御用のオペレーティング・システムとして利用できるのでしょうか。現在のAIXは、エンタープライズ・クラスのUNIX®サーバーであるpSeriesの64ビットUNIXオペレーティング・システムとして、先進性・信頼性・スケーラビリティーにおいて優れていますが、適用用途が昨今、e-businessに集中し、そのため技術情報に偏りがちです。
本論文では、筆者の活動を通して得たリアルタイム制御システムの技術的課題、構築する際の方法論、プラットフォームの選択についての評価、リアルタイム・システムとして採用を決定する際の検証方法の考え方について解説します。そして、それを踏まえて典型的なリアルタイム制御システムのアーキテクチャー・モデルを提案しています。

Armed with features such as freeware and open sources, Linux is steadily expanding its applications not only in the field of Internet servers but also to the operating systems used by measuring and control systems. On the other hand, IBM's AIX® has announced its support for real-time tasks from V3.1, but can one really say that this can be used as an operating system for real-time control? The current AIX is outstanding in terms of its advanced qualities, reliability and scalability as a p Series 64-bit UNIX® operating system, but its uses have been concentrated recently in the field of e-business, and there is thus a tendency for there to be a bias toward technical information. In this paper I take a look at the technical issues involved in real-time control system on the basis of the experience I have gained through my own activities, methodology and platforms used when engaged in building, assessment in regard to the selection of platforms, and approaches to inspection methods when deciding on adoption of a real-time system. I then go on to propose architecture models for typical real-time control systems on this basis.

1. はじめに

e-businessの多くは典型的なオンライン・トランザクション・システムであり、スループット、スケーラビリティー、信頼性が重要ですが、一方で、制約時間内での処理の保証というリアルタイム制御システムも重要な課題です。この分野では、多くのリアルタイム・オペレーティング・システムが存在します。

このような動向の中、筆者は、リアルタイム制御が必要であるシステムのデザインを検討する機会を得て、UNIX®を利用したリアルタイム制御システム構築の検討を行いました。

2. 要件の確認

「リアルタイム・システム」「リアルタイムに～するシステム」という言葉が氾濫しています。銀行オンラインのリアルタイム・システム、リアルタイム・オークションなど…、さまざまな事例があります。Webサイト、e-コマースなど、バッチ・システムでないシステムのことを、リアルタイム・システムと呼ぶ場合もありますが、こうしたシステムは、本論文が対象とするリアルタイム・システムとは違います。

リアルタイム・システムは、処理時間についての要件が厳格なもの（ハード・リアルタイム）と、緩やかなもの（ソフト・リアルタイム）の二つに分類できます。例えば、プラントの制御システムのようなものでは、時間に厳格な処理が実施あるいは完了しなければ、大災害を招くようなものもあります。一方、制約条件が緩やかな例としては、マルチメディアの再生のようなものがあります。こうしたタイプのものは、制約条件の時間を守れなかったとしても許容されるようなアプリケーションです。例えば、音声やビデオの再生では、時間までに処理できなければ、音飛びや再生画像の歪みなどが発生しますが、通常、人間が鑑賞する限りにおいては一定の範囲で許容されます。

まず、対象となるシステムがどちらのタイプなのかを、顧客の要件から明確にする必要があります。また、適用業務によっては、IBMとしては対応できないものも存在するので注意が必要です。

筆者が取り組んだシステムの主な制約条件を挙げます。

- *x*ミリ秒ごとの時刻同期処理
- 時刻同期処理とは別に、*yyy*／秒のトランザクション発生
- システム障害発生時、*z* 秒以内の復旧・業務回復

ここでは、典型的なモデル・システムとして、他システムとのタイマー同期のための入力と、非制御機器との入出力を行うリアルタイム制御システムを考えます（図1参照）。

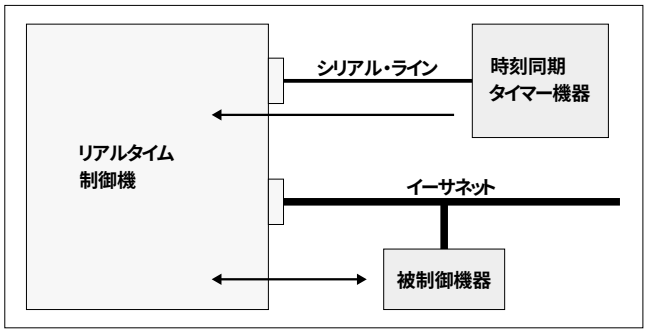

図1. リアルタイム制御システム

ハード・リアルタイム制御システムでは、求められる制約時間内に処理が完了できない場合は重大な影響が発生するので、厳密に設計・開発を行う必要があります。

リアルタイム・システムとビジネス業務アプリケーションとの設計段階での違いは、制約条件が守られるかどうかという点です。分析レベルでは、オペレーティング・システムが取り扱う処理単位、すなわちタスクやスレッド、プロセス、あるいは取り扱うハードウェアやソフトウェアの割り込み処理や、それらの優先順位などが必要になる点が大きく異なります。

通常、ビジネス業務アプリケーションでは、多くの場合、トランザクション・モニターやデータベース・マネジャーといったミドルウェアが介在し、システムとして関係するコンポーネントが多くて複雑であり、トータル・スループット重視です。このため、このような低レベルの分析を設計段階で行う必要は通常はありませんし、オペレーティング・システムがハンドリングするハードウェアの割り込みを考慮することもありません。リアルタイム・システムでは、制約時間を厳守できるか否かという点で、こういったレベルの検討が必要となります。

また、リアルタイム・システム機器が接続して利用するI/Oを明確にしておくことも必要です。処理内容を明確にすることは当然ですが、ここでの目的は、処理のデッドラインを明確にするために、外部要因を明確にすることにあります。例えば、GUIが必要であれば、ポインティング・デバイスからの割り込みが発生します。

つまり、次に挙げる項目を要件定義作業で明確にする必要があります。

- リアルタイムの要件が、“ハード”が“ソフト”か確認する。
- リアルタイム処理の処理内容を明確にする。
- 接続されるデバイスなど、I/Oを明確にする

3. デザイン方法論

Functional Aspects（アプリケーション）のデザイン・フェーズとして、リアルタイム制御システムの業務を分析し、設計を詳細に実施します。

この局面では、アプリケーションのモデリングを実施します。IBM Global Services Method（GSMethod）ではUMLなどを利用して、アプリケーションの静的モデルと動的モデルを確立することになります。動的モデルはシーケンス図とコラボレーション図を作成・利用し、作業を進めるとよいでしょう。

リアルタイム制御システムでは、動的モデルがデザインのポイントとなります。オブジェクト指向開発、特にGSMethodでは、アプリケーション・デザインを実施する場合に、その振る舞い、つまり動的モデルの確立を重視します。段階を追って、この動的モデル作成作業を繰り返し、精度を上げ、物理的／内部（詳細）デザインにまで進めます。

リアルタイム制御システムでは、すべてが動的です。つまり、ネットワークからのI/Oのために、オペレーティング・システムは割り込みハンドラーをスケジューリングし、タイマー割り込みが発生し、多くの事象はイベント・ドリブンとなります。こういった各種の割り込みを、オペレーティング・システムがサポートする一つの処理として、リアルタイム・タスクが存在します。

このリアルタイム・タスクは、ある時間内に処理が完了しなければなりません。静的モデルの分析を助けるものとしてクラス図を作成するのも確かによい方法です。しかし、ここで注意しなければならないのは、最終的にUNIXベースで実装を行う場合に、恐らくC++ではなくC言語を採用することになることです。つまり、実装レベルでは、オブジェクト指向は採用できないのです。このため、リアルタイム・タスクは、あくまでも分析手段の一つの方法と位置付けて利用することになります。

ただし、一般にはリアルタイム制御システムにおける静的モデルは、ある程度シンプルになります。分析の主体は、動的モデリングの確立なのです。

3.1. リアルタイム・システムに求められる性能要件

UNIXをリアルタイム・システムのプラットフォームで利用するメリットは、主に開発生産性と保守性の良さにあると考えています。C言語やPOSIXなどで標準化されているライブラリー関数などが用意されているため、プログラム開発者の確保が容易かつ品質の面でも有利となります。リアルタイム・システムが専業の開発ベンダーは、リアルタイム・タスク専用のオペレーティング・システムなど、得意なプラットフォームがあると思いますが、主にPCを含む汎用コンピューターで企業の基幹業務を取り扱っているものにとっては、Windows®とUNIXが身近な選択肢となります。

POSIXはアプリケーションのポータビリティーをメインテーマにして開発された標準です。UNIXをベースに標準化され、現在、スレッドやリアルタイム・タスクについても標準が策定されています。リアルタイム・タスクについては、IEEE POSIX 1003.1b-1993と1003.1i-1995にて、リアルタイム拡張が規定されています。The Open GroupのThe Single UNIX Specification Version2に記述されているリアルタイム機能から各項目を列記すると、以下のようになります。

- セマフォー
- プロセス・メモリー・ロッキング
- メモリー・マップ・ファイル＆共有メモリー・オブジェクト
- プライオリティー・スケジューリング
- リアルタイム・シグナル拡張
- タイマー
- プロセス間通信
- 同期I/O
- 非同期I/O

実際には、こうした機能を実現するにはオペレーティング・システムの性能や機能に依存する部分も多いため、本論文ではオペレーティング・システムに求められる要件も含めて以下の整理で分析を進めることにします。

- マルチタスクおよびタスク間通信機能
- 高速なコンテキスト・スイッチ
- メモリー管理システム
- 高精度タイマー・サービス
- タスクの固定優先順位
- 「割り込み」に対する短い遅延

このほかにNon-Functional Requirementとして、開発生産性、保守性、信頼性、ポータビリティー、問題判別の容易性、サポート体制といった項目も忘れてはならない重要な点です。

3.2. プラットフォームとしての適性

検討の対象としてわれわれの身近にあるUNIX準拠のオペレーティング・システムであるLinuxとAIX®について検討結果をまとめます。

3.2.1. Linux

Linuxには幾つかの特長があります。

- フリーウェアである。
- オープン・ソース・コードである。
- 最も普及しているPC/AT互換機で普及している。
- 多くのプラットフォームにポーティングされている。

- 「Linux」とは、厳密には「カーネル」を意味し、多くの場合、ディストリビューターによってパッケージされたものを利用する。
- 非常に小さいカーネルとシステムを構築することが可能。
- カーネルのモジュールを動的にロード／アンロードできる。
- サポートはコミュニティーを中心にボランティア・ベースで行われている（ディストリビューターなどを中心に有料サポートも展開されつつある）。

PCで利用されているオペレーティング・システムとしては、Microsoft®社のWindows®が広く普及して久しいのですが、少なくともLinuxは、インターネット分野におけるアプライアンスとしては破竹の勢いで普及しています。

Linuxのソース・コードはGPL（The GNU General Public License）というライセンスに基づいて公開されており、ソース・コードなどをダウンロードなどで入手し利用することが可能です。

Linuxがフリーウェアで、かつ稼働するコンピューターが最も普及していて、安価に入手できるPCということであれば、より広範囲な応用分野で活用することも当然のごとく考えられます。Linuxは、その容量の小ささや信頼性などから、組み込み機器のオペレーティング・システムとして利用が広まりつつあります。こういった流れの中で、リアルタイム・オペレーティング・システムとしての応用も進められています。

3.2.2. Linuxベースのリアルタイム OS

Linuxをベースにしたリアルタイム・オペレーティング・システムが開発されています。そのうち、RTLinuxなど、Webサイトなどから入手できるものもあるようです［参考文献4、5］。

RTLinuxは、リアルタイム・スケジューラーとタスク優先度の管理といった機能を提供するリアルタイム・カーネルなどを提供し、リアルタイム・カーネルで割り込み、あるいはLinuxからの割り込み制御をフックすることによって、リアルタイム性に関する割り込みをRTLinuxで処理・管理します。リアルタイム・タスクと非リアルタイム処理（Linuxプロセス）の通信を処理するものとしてFIFOが提供されており、リアルタイム処理と非リアルタイム処理を連携し、並行して処理することが可能です。また、RTLinuxではタイマー精度もナノ秒精度でカウントしているので、より細かい間隔の制御が可能です。

現行のRTLinuxではPOSIX1b準拠のAPIと独自拡張のAPIを備えているためUNIX/Linuxプログラマーが開発を行うことについて敷居は高くないと思われるので、プラットフォームへの投資を低く抑えたい場合には有効なソリューションと思われます。

LinuxあるいはLinuxをベースにしたフリーウェアのパッケージを採用する場合の注意点としては、カーネルやディストリビューションのバージョン・アップなどに対応するための情報収集や適用作業を、原則としてすべてユーザーが負うことです。顧客の環境で長期にわたる利用を前提とした場合は、この点についてサポート体制を確保するよう考慮しなければならないのは言うまでもないことでしょう。開発フェーズでのサポート体制、および、運用フェーズでのサポート体制の確保が、利用製品の確定までにめどが付く必要があることから、現段階で日本アイ・ビー・エムにて実施するSIプロジェクトなどでの利用についてはリスクを考慮しなければならないでしょう。

3.3. AIX

AIXはIBMの提供するUNIXです。最新版のAIX5Lは64ビット・サポートのカーネルであり、Linuxとの親和性が高くなっています。AIXは、The Open GroupのUNIX98ブランドを取得しており、文字通りUNIXです。よってマルチタスク、タスク間通信の条件は当然満たされています。

AIXではバージョン4.1以降、カーネルとしてスレッドをサポートしており、現行のバージョンではM:Nスレッド・モデルが実装されています。M:Nスレッド・モデルは、カーネルの実態であるカーネル・スレッドと、ユーザー・スペースに存在するスレッドとの対応を意味しており、カーネル・スレッドと1:1で対応するユーザー・スレッドと、一つのカーネル・スレッドに対して複数のユーザー・スレッドが対応するものとが存在する実装です。1カーネル・スレッドに対して複数のユーザー・スレッドがマップされるものは、pthreadライブラリーがスレッドのスケジュールを実行するので高速にスレッドのCPUディスパッチが可能です。

AIXのカーネルはプリエンプティブです。また、最近は忘れられていますが、AIXはリアルタイム処理のサポートをうたっているオペレーティング・システムです［参考文献7］。

AIXにはカーネルを拡張(Kernel Extension)する機能があり、カーネル機能の変更・追加が可能な仕組みを備えています。デバイス・ドライバー、システム・コール、カーネル・サービス、プライベート・カーネル・ルーチンが、その対象となります。

カーネル・サービスは、カーネル・モードで実行されるプログラムに対してランタイム・カーネル環境を提供するルーチンであり、AIXカーネル機能そのものといってよいでしょう。カーネル拡張を実装する場合には、このカーネル・サービスを利用することが可能です。

最近では、Linuxがカーネル・モジュールの動的ロードを実現し、その柔軟性が注目されています。もともとAIXでは、カーネル拡張機能を備え、ハイパフーマンスの達成や、カーネルとユーザー・スペース間のパフォーマンス関係データの交換の実現に利用しています。

リアルタイム処理では、要件の制約時間が一般にミリ秒・マイクロ秒といった大変短い時間単位であるため、オペレーティング・システムの動作・性能としては、高速なコンテキスト・スイッチングや、できる限り小さい割り込み遅延が求められます。AIXのリアルタイム処理サポートは、この制約に対処するためカーネル・サービスとして多くの機能を提供しています。

3.3.1. AIXのリアルタイム処理サポートの実装

AIXはSystem Vリリース2を基に開発されています［参考文献7］。V3.1では、リアルタイム処理をサポートするために、固定優先順位やカーネルのプリエンプティブ化を実現しています。

3.3.2.「割り込み」に対する遅延最小化の実現

AIXの実行環境にはプロセス環境（Process environment）と割り込み環境（Interrupt environment）が存在します。割り込み環境は、外部イベントに対し例外ハンドラが処理を実行します。この二つの環境の基本的な違いは、割り込みハンドラーが、基本的にタイム・スライスされない、すなわちプリエンプトされないことにあります。AIXでは、この割り込みの制御をカーネル・モードにあるプロセスからi_sched( )やi_disable( )などのカーネル・サービスを利用して管理することが可能です。

割り込みを上げるデバイスとデバイス・ハンドラーが一定範囲の時間で処理を終了することが、リアルタイム・システムでは必要です。これが守られないと、ほかのすべての処理が予測通り動作したとしても、イベントの消失やシステム・パフォーマンスの低下につながり、結果としてハード・リアルタイムが保証できなくなります。

AIXでは、割り込みの優先順位に対する処理時間がルールとして存在しますが（**表1**）、割り当てられたサービス・タイムより長時間の処理時間が必要な割り込みに対する手段として、オフレベル割り込みという仕組みを提供しています。

AIXは、このような方法で、割り込み遅延の最小化を図り、また、長時間処理の割り込みに対する制御・処理を提供することによって、処理時間の予測を可能としたリアルタイム性を提供することが可能となっています。

3.3.3. メモリー管理システム

リアルタイム・システムでは、限られた時間内の確実な処理の実行や処理結果の送信が必要であり、保証できなければなりません。AIXは、ユーザー・プロセスに対して広大なアドレス空間を提供しており、仮想メモリー・マネジャーによってデマンド・ページングがサポートされています。処理に必要な実メモリーが存在しない場合、ページ・スティールなどで他プロセス

表1．割り込みの優先順位に対する処理時間

優先順位	サービス・タイム（マシン・サイクル）
INTCLASS0	200サイクル
INTCLASS1	400サイクル
INTCLASS2	600サイクル
INTCLASS3	800サイクル

から実メモリーを奪ったり、ページングによって必要なメモリーを確保しようとします。万が一、リアルタイム処理を行うプロセスがページングの対象となった場合、要求されたサービス時間を守ることができなくなる可能性が出てきます。安定したサービス時間を維持し保証するには、少なくともリアルタイム・タスクのアドレス空間についてのページング発生は回避しなければなりません。また、割り込みハンドラーのモジュールも同様の理由でページングが発生してはなりません。

POSIXでは、このような目的のためにmlock( )、munlock( )、mlockall( )、munlockall( )といったAPIが規定されていますが、現状のAIXではサポートしていません。しかし、AIXでは、pin( )、unpin( )、pincode( )、unpincode( )といったカーネル・サービスが同じ機能と、メモリー管理のための、より幅広く強力なサービスを提供しています。

pincode( )は、プロセスのコードとデータ・エリアを同時にページ固定することができるのでお勧めです。ページ固定について一つ注意しなければならない点として、共有メモリー・セグメントおよび共有ライブラリーのページ固定があります。ページ固定の目的として、不測のページング発生を防ぎ、要求仕様を保証することにあるので、リアルタイム・プロセスでこれらを利用する場合には、ページング発生対策を取る必要があります。具体的には、ここで紹介した一連のメモリー固定のサービスを利用することで、AIXではメモリー・ロッキングについて対応が可能です。

3.3.4. タスクの固定優先順位

AIXのプロセス（スレッド）・スケジューリングは、伝統的なUNIXのスケジューリング・アルゴリズムと同じく、マルチレベル・フィードバック付きラウンド・ロビンと呼ばれるものです［参考文献7］。128段階の優先度レベルを持ち、数字が小さいほど優先度が高くなります。割り込みハンドラーが最も高く、カーネル／リアルタイム・プロセス、通常ユーザー・プロセスの順で優先度が低くなります。

伝統的なUNIXおよびAIXでは、swapperという名称のプロセスが存在しますが、実はこれがスケジューラーです。AIXでは、スケジューラーが1秒ごとに全プロセスの優先度を計算し直し、ディスパッチャーは10ミリ秒ごとにプロセス／スレッドのCPUディス

パッチを実行します。nice( )や setpriority( )は、ユーザー・プロセスの優先順位を変更するためのコールですが、スケジューラーによって優先度が再計算され、実行時の優先度は可変となることからも分かるように、こうしたコールを発行したときの基本優先度を変更するだけです。

ユーザー・プロセスの優先度は「40～127」であり、それ以上はカーネルやリアルタイム・プロセスに割り当てられます。リアルタイム・プロセスは固定優先順位であるため、スケジューラーが対象にする必要はありません。ただし、AIXは基本的にはこのように動作するため、リアルタイム・プロセスに対しても割り込みが発生する可能性があります。このため、リアルタイム・プロセスの優先順位は、スケジューラー、つまりswapperプロセスよりも高く設定することが望まれます。AIXでは、これが可能です。swapperプロセスの優先順位は「16」となっているので、「15以上」ということになります。

AIXでは、スレッドがサポートされる前からプロセスの固定優先順位を独自のsetrpri( )コールによってサポートしてきました。スレッドをサポートするようになり、POSIX準拠のpthread_setschedparam( )などでプライオリティー・ポリシーと優先順位値を設定することが可能となっています。また、リアルタイム・タスクの優先順位を固定にしてswapper／スケジューラーの影響を除外するために、より高い実行優先順位に設定することが可能です。また、割り込みの制御も可能であるため、リアルタイム・タスクのスケジューリングと時間予測については高いレベルで実装可能と考えられます。

3.3.5. 高精度タイマー・サービス

AIXは独自の高精度タイマー・サービスをカーネル・サービスとして提供しています。alarm( )、getitimer( )、setitimer( )などを、System VやBSDとの互換性のためにサポートしていますが、その精度はマイクロ秒までです。POSIXではリアルタイム用途として別のAPIを規定していますが、それはメモリー管理と同様の理由（と推測）で実装されていません。

AIXでは高精度タイマー・サービスとして、talloc( )、tsrat( )、tstop( )、tfree( )が提供されています。ナノ秒レベルの高精度が要求される場合、クリティカルなタイミング管理が必要な場合、および絶対時間でのタイマーが必要な場合は、こうしたカーネル・サービスを利用して容易に作成することが可能です。

よって、AIXでは高精度タイマー・サービスについて十分な実装が可能であると判断します。

4. 評価

ここで、評価用環境を用意し、何をどう評価すべきかを議論します。

4.1. 何を検証するべきか？

まずはじめに、機能の検証を行う必要があります。特に、リアルタイム・カーネル用のパッケージを利用する場合には、タスクの生成、タスク間通信など、ベースとなる製品単体で使用する場合には一般的な機能であるものも、パッケージで提供されている独自APIであるならば、一通りの機能を確認するべきです。こういった過程でAPIの使い勝手も含めて評価する必要があります。また、割り込みの制御、ページ固定についても同様に調査・検証を実施します。

以下、オペレーティング・システム（カーネル）自身の性能の検証項目を列記します。

- コンテキスト・スイッチ時間
  - ― ユーザー・プロセス（スレッド）間
  - ― カーネル・プロセス（スレッド）間
  - ― 割り込みによるコンテキスト・スイッチ時間
- 割り込みの遅延
- タイマー・サービスのシステム・オーバーヘッド
- タイマーの精確度
- タスク間通信のオーバーヘッド、スループット
- そのほか、利用を予定するサービスのパフォーマンス（AIXの場合、GDLCのスループットなど）

例えば、タイマーの精度を検証する場合、そのばらつきを考慮し、リアルタイム・タスクの動作タイミングを設計で考慮する必要があります。ばらつきの原因は、タイマー・サービスを実行するためのカーネルのオーバーヘッドであったり、割り込み処理の遅延などです。アプリケーションとして、タイマーとして利用するサービスは、その提供精度（粒度）だけではなく、実際にタイマーが通知される精度（ばらつきの最大／最小）を考慮する必要が出てくるため、ここでの検証が意味を持ってきます。

例えば、ここでの検証値は、設計時に以下のように利用できます。

図2で示すタイマー精度の検証項目においては、CPU使用率やI/Oの発生率なども、併せて取得することが必要です。処理内容によっては、CPUクォンタム内に処理が終わらないこともあり得るので、その場合にはアプリケーション・デザインの見直しが発生します。ソフトウェアの選定作業を通して、ハードウェア・プラットフォームについても選定に必要なデータを入手します。

割り込みの遅延やコンテキスト・スイッチのオーバーヘッド

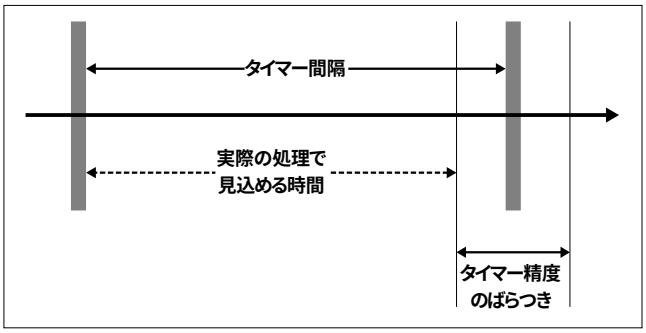

図2. タイマーの精度の検証

なども、タイマーの場合と同様です。割り込みについては、使用を予定するハードウェアが備えているデバイスを動作させ確認してください。すべてのデバイスの割り込みが予定されている時間内で完了しているのかを確認することは、リアルタイムの制約時間を保証できるかどうかの最大のポイントです。

このような積み上げで、要求仕様内で処理できることを設計時点で保証する必要があります。

4.2. 検証方法

各目的別にプログラムを開発します。オーバーヘッドなどの計測には精度の良いタイム・スタンプが必要ですが、AIXの場合にはナノ秒まで表示するトレース機能を使用できるので、システム内での遅延やオーバーヘッドの計測に利用できます。ほかのシステムの場合には、検証ツールも含めて調査の必要があります。I/Oポートに評価イベントごとに結果を出力しデジタル・オシロスコープにて表示するというのも一つの方法でしょう。計測・制御用途のリアルタイム・システムの場合には、こういった機器は通常必須であり、実業務アプリケーションのデータ・フローに近似できるので有効な方法です。

以上のような方法で、利用するプラットフォームの性能、基礎データを収集し、アプリケーション・デザインの動的モデリングに反映します。

5. システム管理の方法論

ハード・リアルタイム制御システムにおいては、その適用業務の性格や担う社会的責任から、極めて高度な可用性と信頼性が求められることは言うまでもありません。

Webシステムを含む一般的なオンライン・トランザクション・システムなどでは、可用性の確保のために、プロセス監視（ヘルス・チェック）、CPU使用率、リアル・メモリーやページング・スペースの使用量、ワーク用ディスク容量使用量などのシステム・リソース監視を行うのが通常です。また、可用性についていえば、重要プロセスなどを定期的にヘルス・チェックして障害を検知した場合、即時に再起動するといった方法が採られます。また、業務を稼働する筐体（きょうたい）レベルでの可用性向上の手段としては、HACMPのようなクラスター・システムによるバックアップ構成を採ることは、今や常とう手段といえます。

ここで、リアルタイム制御システムの場合について検討することにします。

まず、システムのヘルス・チェックの観点からいえば、システム・リソースの監視、および業務を実行しているプロセス、あるいはプロセス群などを監視するという方法論は同様に採用すべきです。さらに、アプリケーション自身に監視機能を組み込むという手段も考えられますが、いずれにしてもプログラム・ロジックを実行している主体であるタスクやプロセスなどの稼働を常時監視することは、RAS確保のために必要最低限実施しなければなりません。同様に、ネットワーク、または筐体（きょうたい）レベルの耐障害性能を高めるためにバックアップ機を用意し、即時にスイッチ・オーバーするという方法も同様に採れると思われます。

ただし、通常のOLTPやバッチ処理の業務との最も大きく、かつ唯一の違いは、障害の検知および障害から業務回復までの時間を極めて短い時間内に完了する必要があることです。もちろん、回復時間については、業務の性格や要件によって大きく変わることは言うまでもありません。

ここでは、大きく三つの側面から、設計段階で考慮しなければならない点を説明します。

5.1. ハードウェアの側面

ハードウェアのRAS向上機能として、IBM pSeries にはCPU Dynamic Deallocationという機能があります。この機能はプロセッサーが2ウェイよりも多く搭載されている場合に威力を発揮します。つまり、あるプロセッサーの障害発生がしきい値を超えると、そのプロセッサーで処理しているタスクを正常なプロセッサーに移行して処理を継続させ、移行後、障害が多発しているプロセッサーを停止させ、システムから切り離します。

当然、こういった機能が動作する場合には、ハードウェアによる割り込みが発生すると思われるので、このような機能を持つSMPマシンをリアルタイム制御業務に用いる場合には、ハードウェア選定時に必要な情報を入手し、規定されている時間内に処理が継続されるか否かの確認が必要です。あるいは、制約時間内に処理が完了できないことが明らかな場合には、バックアップ機による処理の引き継ぎを検討しなければなりません。

ハードウェア機能によるリカバリーについては、エンド・ユーザーにより検証することが困難な場合があり得ます。技術情報が十分に入手できず、不確実な要素が残る場合には、プラット

フォームとして採用するかどうかの大きな判断が必要となります。

基本的には、本番機、ホット・スタンバイ機といったシステムで冗長構成を採ることを基本とすべきでしょう。

5.2. ネットークの側面

一般にネットーク管理技法としては、二つの方法が採られます。一つは、SNMPマネジャーが存在するシステムがSNMPトラップを受信し、監視対象となるシステム障害を検知する方法です。もう一つは、管理マネジャーから監視対象となるシステムのネットワーク・インターフェースに定期的に状態監視のポーリングを行って稼働状態か否かの監視を行う方法です。代表的なものとしてTivoli® NetView®などがあり、この二つの機能を実装しています。

一般のネットワーク管理システムが、SNMPトラップとicmp echo/replyによるポーリングの双方を併用している理由は、それぞれ一方だけでは不十分だからです。

SNMPトラップには、次の欠点があります。

- ベースとなるプロトコルの性格上、信頼性に乏しい（ネットワーク上にて消失の可能性がある）。
- ネットワーク・アダプター障害の場合には、SNMPトラップを送信することが不可能。
- 監視対象システムのTCP/IPをつかさどるモジュールやオペレーティング・システムに異常が発生した場合には、SNMPトラップをSNMPマネジャーに送信できない。

一方、ポーリングだけに頼ると、ポーリング間隔時間の設定によっては、障害の検知が遅れることになります。

以上のような理由から、通常のネットワーク管理システムでは、両方の方法を併用しています。

リアルタイム制御システムでは、モデル・ケースの例で考えると、イーサネット・ネットワークが制御機器との通信バスになっています。このバスのトラフィックが多い場合、コンテンションが発生するので、必要最少限のネットワーク・トラフィックとすべきです。これは、スイッチング・ハブを使用した場合も同様であり、コンテンションが発生することを想定すべきです。なぜなら、対象制御機器とネットワーク監視マネジャーは別筐体（きょうたい）であるからです。では、ネットワーク監視のトラフィックの影響を除外すべく、ネットワーク監視専用のLANを設けてみた場合はどうでしょうか。

この場合、確かにネットワーク監視のicmp request/replyパケットによる業務データ送受信とのコンテンションは発生しなくなります。ここで重要なのは、監視パケットが増えれば、それを受信し応答処理をする側にも負荷が発生するということです。

結論をいえば、システムのヘルス・チェックという観点で、監視システムを設置することは有効であるといえます。しかし、それによってリアルタイム・システムの負荷が増えることは望ましいことではありません。よって、SNMPマネジャーによるヘルス・チェックを用いる場合には、あくまでも障害を運用担当者に通知するためのものと割り切ることを提案します。障害検知時のシステムの切り替えには別途、方法を検討することが望まれます。

ネットワーク障害やネットワーク・アダプターの障害を検知する方法として、HACMPは大変参考になる実装です。HACMPでは、ネットワーク経路の障害とシステム（筐体（きょうたい）レベル全体）障害との判別するために、ネットワーク経由の相互ポーリングと、Non-TCP/IPネットワーク（多くの場合、シリアル・ライン経由）でのハートビートといった二つの方法を併用します。これにより、システム・レベルでの障害なのか、ネットワーク経路の障害なのかを判別しています。

リアルタイム制御システムにおいても、本番機とスタンバイ・システムが相互にシステム状態を監視し合うという方法論は適用できると考えられます。ただし、実装する場合には慎重な検討が求められます。

というのも、まずHACMPのような製品を利用した場合、ハートビートの時間間隔が問題なのです。通常、このハートビート間隔は可変ですが、リアルタイム制御の要件を満たす間隔まで値を小さくできるかが問題となります。また、ハートビート・プロトコルに限らずネットワーク経由で通信を行う場合には、リトライ／タイムアウトなどの仕組みで、そのプロトコルの信頼性を高めることが一般的ですが、リトライの間隔と回数によって障害検知までの最大時間が決定されます。

これは、極めて短い時間でハートビートが必要で、厳しい条件となってくることを意味しています。また、ハートビートの処理が増えれば、それに伴うシステム負荷が増加するのは、SNMPによる監視と同様です。

HACMPの場合、本番機からリソースを引き継ぐときに、諸々の処理を行います。例えば、IPインターフェースの引き継ぎのために、IPアドレスの付け替え、MACアドレスの変更、ミドルウェア、アプリケーションの起動などです。製品を利用して実装する場合には、仕様の確認をすることは言うまでもないことですが、こうした処理が本当に必要なのかどうかの検討が必要です。また、逆の発想で、こうした処理が発生しないような方式を検討することが現実解でしょう。相互のシステムは、引き継ぎリソースはなしで、アプリケーションのみ、状況の変化を認識し動作するというデザインが最も効率的です。

以上のようなことから、障害検知後のスイッチ・オーバー処理のために独立したハートビートやポーリングという方式は、

リアルタイム制御では冗長な方式と言わざるを得ません。

これに代わる一つの方法として、本番機と制御機器の通信をスタンバイ機がモニターする方式が考えられます。すなわち、スタンバイ機はネットワーク・アナライザー機器のようにネットワーク上のパケットをすべてチェックします。予定された時間以内に、被制御機器とリアルタイム制御機間で通信が発生しなければ、無条件にスタンバイ機が業務を引き継ぎます。ポーリングやハートビートに比べてシステムのオーバーヘッドはなく、またリトライ処理の心配がないため、リカバリー時間の制約は緩和されます。

5.2.1. 通信プロトコルの選定

通信は、相手システムとプロトコル、フォーマットを共有した上で成り立つものなので、一方的に自システムの都合だけで決定できるものではありません。そのため、最初に以下のような検討項目が発生します。

- 制御・被制御機・共通でサポートしているネットワーク・アダプターは何か。
- 各システムでは、通信プロトコルとして何がサポートされているか。
- サポートされている通信プロトコルでは、どういった仕様のAPIが提供されているか。あるいは、どのレベルまでユーザーが制御可能か。
- 製品の動向と寿命。

現在は、PCでも通信プロトコルとしてTCP/IPのプロトコル・スタックが実装されており、APIとしてソケット・インターフェースが利用できます。リアルタイム制御システムの場合は、機器のメモリー搭載量、オペレーティング・システムの制約などによって、より下位プロトコル・レベルで制御される傾向にありました。また、実際、Point-to-Point通信を行う場合が多く、TCP/IPのような高度なプロトコルの必要性も低かったかもしれません。

オペーレーティング・システム上での処理を考慮する場合、明らかにTCP/IPのような構造化されたプロトコル・スタックとして実装されるプロトコルを利用する場合の方が、パス・レングスが長くなります。直接、下位プロトコル・レベルでコーディングした方が、IPやTCP/UDPのプロトコル・スタックの処理を経由しないため、パス・レングスが短くなり処理も早くなります。

このような理由から、リアルタイム・タスクのパス・レングスが長くなると推測される場合には、TCP/IPやSNAなどのプロトコルの採用が必須かどうか判断が必要です。AIXの場合は、前述しているようにGDLCというサービスが提供されているため、下位プロトコル・レベルでの制御を開発することは容易です。

一方、時代の流れに沿って、被制御機器もTCP/IP、ソケット・インターフェースを実装している場合には、開発生産性から採用したくなるのが当然でしょう。このような場合は、TCP/IPを利用した場合のパス・レングスを計測する必要があります。また、通常、arpパケットもブロードキャストされるので、そのような影響も考慮して検証を行う必要があります。

通信プロトコルを決定する上で忘れてならない点は、コネクション・オリエンテッド・プロトコルを利用するのか、あるいは、コネクションレス・プロトコルを利用するかの判断です。コネクション・オリエンテッド・プロトコル、例えばTCPですが、そのプロトコル・メカニズムの中に信頼性確保のためにリトライなどの処理が含まれています。一般には、リアルタイム処理で必要となる制約時間よりは非常に大きい時間単位ですから、よほどのことがない限り原則としてはコネクションレス・プロトコルを使用するべきです。

UDPは信頼性のないプロトコルです。信頼性が不足する部分はアプリケーションで補う必要があります。

フレームの信頼性は、CRCでチェックされます。問題は、フレームの欠落と同一フレームの複数配信です。こうした事象は、ゲートウェイにおけるルーティング処理で発生しやすいとされています。TCP/IPを採用する場合には、ゲートウェイまたはルーターを利用しないネットワーク構成を設計すべきです。その上で、フレームの欠落／重複発生頻度を実測すべきです。アプリケーション側では、シーケンス番号によって重複フレームの有無を判断するなどのロジックが必要です。パケットの欠落に関して受信パケットについては、Ackフレームを返信するよう設計しなければなりません。

5.3. ソフトウェアの側面

システム管理として、ソフトウェア、特にリアルタイム・タスク（機能）のヘルス・チェックをどう考え実装するかを議論します。

一般に、UNIXというオペレーティング・システムには、プロセスあるいはスレッドの起動（生成）や停止（消滅）を第三者が知る仕組みがありません。一般的には、TivoliのDistributed Monitorなどのプロセス・モニター・プログラムは、psコマンドからモニター対象のプロセス名／コマンド名をサーチし、ヘルス・チェックとしています。リアルタイム・システムを考えたとき、この方法はまったくナンセンスです。

まず、この方法を実行するために、psコマンドやgrepなどのパターン・マッチング・ユーティリティーのプロセスなど、幾つかのプロセスが生成し消滅し、非効率です。また、この方法は一種のポーリング方式であり、検知のリアルタイム性にも問題があります。

一つの方法としては、AIXのカーネル・サービスが備える「Process State-Change Notification」という機能を利用することです。このルーチンを作成することにより、プロセス、スレッ

ドの生成や消滅などの状態を知る仕組みが作成可能です。MVSなど、メインフレームでは当然の機能であり、エンタープライズ・クラスのサーバーとして、AIXでも標準の機能となることを期待したいものです。

6. アーキテクチャーの確立

システム・レベルとして図3のような方式を採ることが、一つの結論です。

アプリケーションの要件上、可能であるなら、リアルタイム処理を行うプロセス／スレッドと、リアルタイム性が必要でない処理（例えば、エンドユーザー・インターフェースの処理プロセス／スレッド）は分割し、相互にキュー（FIFO）などで、通信するようなデザインを採ると、制約時間内で処理を行うという要求仕様を満足させることに有利だといえます。これはアーキテクチャー・パターンとしては、レイヤー・パターンの応用です。

カーネル・プロセスとして実装した場合には、Cライブラリーの利用について制約があるため見極めておくことが必要です。一般に、リアルタイム性が厳しく問われるシステムであるなら、最初からカーネル・プロセスとして実装することを考えることです。また、可能性のあるすべての割り込みについて、実行時間遅延値をテスト結果を基に導出し、リアルタイム処理中に割り込みが発生した場合に、制約時間を満足できるか判定することが必要です。

例えば、時刻同期タイマー機器からのメッセージが定期的に入ると同時に、イーサネット経由で被制御機器とのネットワーク入出力が発生します。また、場合によっては、ポインティング・デバイス（マウスなど）を動かす場合もあり得ますし、制御データの交換のために、フロッピー・ディスクでI/Oが発生するかもしれないので、こういったことを想定の上で設計を行う必要があります。

今、一つのアーキテクチャー案として、以下のようなコンポーネント配置が考えられます。

- カーネル拡張として部品の組み込み：タイマー・ルーチン／メモリー・ロッキング・ルーチン／I/Oインターフェース、ネットワーク処理など、ソフトウェア部品をカーネル拡張として用意します。
- カーネル・プロセス：リアルタイム・プロセスを実装、カーネル・サービス／カーネル拡張を利用し業務ロジックを担当。
- ユーザー・インターフェース：ユーザー・モード・プロセスとして実装／カーネル・プロセスとユーザー・インターフェース間は、カーネル・サービスが提供するキューなどを利用。

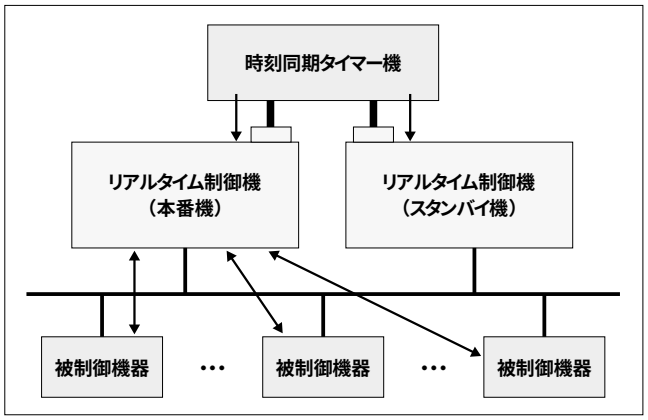

図3．アプリケーション・プロセス／スレッドの構造

7. おわりに

当初はLinuxに着目し、このOSでリアルタイム制御システムを構築できないかと期待していましたが、最も身近なAIXの実力と、真の意味でのスケーラビリティーをあらためて認識しました。

ナレッジ・マネジメントの観点としては、こういったビジネスの主流ではないながらも、コンピューター技術者として、システムの根幹であるオペレーシティング・システムを、いつもとは違う角度から見直し、オペレーティング・システムで提供される機能を再認識し、技術資料としてまとめることも重要であると考えました。

本論文がUNIX系SE職種の方々の参考になることを期待しています。

[参考文献]
[1] DANIEL P.BOVET, MARCO CESATI著、岡島 順治郎・田宮 まや・三浦 広志 訳、高橋 浩和・早川 仁 監訳『詳解LINUXカーネル』オライリー・ジャパン
[2] Alessandro Rubini著、山崎 康宏・山崎 邦子 訳『LINUXデバイスドライバ』オライリー・ジャパン
[3] David A. Solomon, Mark E. Russionvich著、多摩ソフトウェア訳『アーキテクチャー徹底解説 Microsoft Windows 2000上巻』日経BPソフトプレス
[4] 船木 陸議「RTLinux 2.2系の導入と詳細」『Interface 2001年6月号』CQ出版社
[5] 森友 一朗・薬師 輝久・馬場 秀忠『RTLinuxリアルタイム処理 プログラミングハンドブック』秀和システム
[6] 『AIX Version 4.3.Kernel Extensions and Device Support Programming Concepts』日本アイ・ビー・エム
[7] DIPTO CHAKRAVARTY, CASEY CANNON著、アスキー書籍編集部 訳『PowerPC概論』アスキー
[8] 『The Single UNIX Specification Version 2』The Open Group

（ページ数および表記上の観点から、著者の了解を得て編集部にて手を入れてあります）