第 4 回 OS の多重化

今回はまず前回の宿題を解説しておきましょう。
組込みの世界ではよくマルチタスクとよばれるプログラムの並行性と、それに伴う排他制御があり、この話をしたいと思っていたのですが、今回もそれ以外の量が増えてしまったため次回に回します。

前回の宿題とは、割り込みハンドラの構造で重要なのは、割り込みが発生した時に実行されていたのがタスクの場合、スケジューラに飛び、それ以外の場合はスケジューラを経由せず割り込まれた場所に直接戻ることです。これはほとんどの RTOS で共通の仕組みです。この理由はなんでしょうか？ということでした。念のため前回示した図を再度載せておきます。

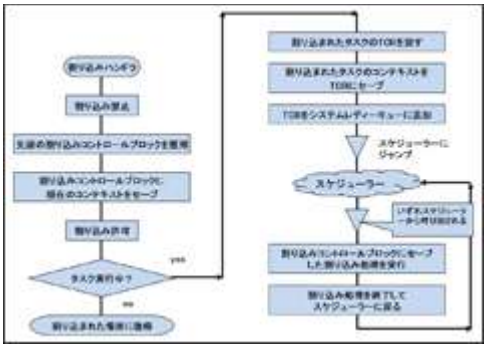

図 1：Mtos における割り込みハンドラの概要

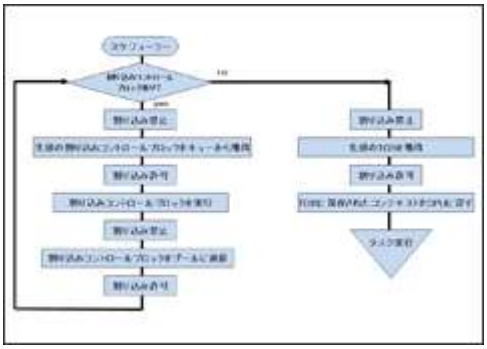
図 2：Mtos におけるスケジューラ(ディスパッチャー）

まずタスクの実行中以外とは、どのようなプログラム(命令)が実行されている時でしょうか？
一つ目は RTOS 自身の実行中です。RTOS もあくまでも 1 つのプログラムで、なおかつ RTOS は外部環境の変化に迅速に応答できるように、できる限り多くの場所で外部割り込みを許可するように作られているのです

から、当然 RTOS 実行中も様々な場所で外部割込みによりその処理を一時停止されます。その他にはなにがあるでしょうか？
外部割込みに関して RTOS 使用時は通常多重割り込みを許可しているため、他の割り込み処理中に割り込みが入ってくる場合もあります、これが二つ目の場合です。
つまりタスク実行中に入ってきた割り込みに対応する割り込み処理は、最後にスケジューラに飛び、それ以外の RTOS 自身が実行中の場合や、（別の）割り込み処理の実行中に入ってきた割り込みに対応する割り込み処理からは、割り込まれた場所に直接復帰する、つまり関数やサブルーチンの終了時と同じ動きをするということです。
ソフトウェアのつくりとしてはこの方が多重化に伴う排他制御の問題もなく、単純で当然品質も良くなります。また多重化並行化は緊急度の高い処理には有効ですが、割り込まれた方の処理のほうは当然遅くなります。つまり直接戻るように構成しないと RTOS や割り込み処理が遅くなり実行量は減ります。

スケジューラに飛ぶのは実行中のタスクよりも優先度の高いタスクが実行可能になった可能性があるからです。しかし RTOS 自身はタスクよりは優先度は高いですし、割り込みにいたっては RTOS よりもさらに優先度が高いと考えられます。

以上のような理由で、タスク以外のプログラム実行中に発生した割り込みに対する割り込み処理終了時は、直接割り込まれた場所に復帰するように設計しているわけです。

今回はコラムが増えてしまい本文は前回の問題の解説しかできませんでした。次回はマルチコア対応 Linux など、マルチコア化に伴って重要性が増してきたソフトウェアによる排他制御の話を中心に進めたいと思います。

第 4 回おわり

コラム① 汎用 OS と割り込み禁止

UNIX や Linux、Windows などのいわゆる汎用 OS（個人向け汎用 OS と呼ぶべきかもしれませんが）の場合は、アプリケーションプログラムから OS に処理が切り替わった時、言い換えると OS の実行が開始されると、同時に原則として割り込みを禁止します。
これは OS 自身を作りやすくするためです。汎用 OS は RTOS と異なり MMU(TLB)を利用してプロセスごとに独立したアドレス空間を与えています。これはつまりプロセスが切り替わる時には、RTOS で実行しているタスク実行環境の切り替え以外に、プロセスごとに存在する論理（仮想）アドレスを物理アドレスに変換するためのページテーブルの切り替えも必要ということです。また汎用 OS はハードディスクとのやり取りや人間とのやり取りなど、処理に時間のかかる I/O とのやり取りも RTOS に比べるとはるかに多く複雑です。
まだデバイスドライバの説明はしていないので、難しいかもしれませんが、例えば OS がドライバ（デバイス）に対してデータをあるアドレスに書き込むことを指示していたとします。その後デバイスは CPU とは独立して実行を続け、そのデバイスからの割り込みによって先ほど指示されたアドレスにデータを書き込もうとしても、すでにそのアドレスのメモリは別のプロセスが別の用途に使っているかもしれません。
このように汎用 OS は RTOS とは別の種類の解決しなければならないことがあります。ここに OS 自身の多重化も考慮するとなると複雑になり過ぎます。OS 実行中に割り込みを禁止しておけば OS 自身の排他制御に関しては気にしなくてもよくなります。
そもそも汎用 OS の目的はスループットの向上など RTOS の目的とは異なります。リアルタイム性の向上と

スループットの向上は実現手段が異なります。極端な言い方をすると、一つのことをやりだした場合は、他を気にせず集中してそれを最後までやり遂げたほうが作業量は増えそうですよね。それが汎用 OS で OS の実行中は割り込みを禁止する理由です。
なお、以上のような理由で、組込みシステムに対応するための Linux や Windows などのリアルタイム化にはカーネル内部における割り込みを許可している部分を増やすという内容も含まれています。

コラム② 割り込みと関数

割り込み処理ルーチン(ISR)や割り込みハンドラ（以後一括して ISR）は関数と似ている点もありますが、異なっている点も多くあります。
まず関数の場合は、その関数を呼び出している命令（プログラム）が必ずあります。ソースプログラムの構造などをチェックする静的解析ツールにかけると呼び出し元がない関数は、デッドコード（実行されることがない命令）として警告を受けるくらいです。
しかし ISR は現実に実行されます。ではどこから呼び出されるのでしょうか？
どこまで厳密に言うかは別として、いわば ISR は CPU ハードウェア自身から呼び出されていると考えてもよいでしょう。
つまり呼び出し元はほかのソフトウェアではなく、ハードウェアということです。具体的には外部割込みによりソフトウェアが呼ぶ出されるわけです。外部割込みが外部の変化を伝えるものなので、外部変化がソフトウェアを呼び出すわけです。これは現実物理世界が変化をソフトウェアに伝える仕組みそのものです。
外部割込みが発生すると FSM（Finite State Machine - 有限状態機械）である CPU は自分自身の状態を変化させます。FSM の話に踏み込むと長くなるので、話はまた別の機会に譲りますが、割り込みは発生すると CPU は自分自身の状態を変化させます。
CPU の状態としては種類はありますが、ソフトウェアからよく見える状態としては一般的には PSW（プロセッサ・ステータス・ワード、個々の CPU で SR（ステータス・レジスタ）や FLG（フラグレジスタ）など呼び方は異なります）と呼ばれるレジスタに見られる割り込み関連の状態などもわかりやすいでしょう。割り込みを禁止/許可の制御を行う割り込みマスク（すべての外部割り込みを一括して禁止／許可できる機能とその時の CPU の状態を示す）や割り込みに優先度を持たせている CPU の場合は実行されている割り込みレベルを示したり、レベルによる割り込みの禁止／許可できる機能などが CPU にはあり PSW からそれを知ることができます。

コラム③ 割り込み処理ルーチンと割り込みハンドラ

外部割込み発生時の通常の動作は“必要最低限の動作終了後は速やかにスケジューラ（ディスパッチャ）に飛ぶ”です。割り込みが発生したということは外部の状況に何らかの変化があった可能性があるわけで、その変化を待っていたタスクがいた場合は、そのタスクを再開させなければならない場合もあるはずです。そのためにスケジューラに飛ぶのですが、割り込み発生に RTOS のスケジューラが実行される分だけアプリケーションに関係のないソフトウェア（RTOS のことです）が実行されることになります。アプリケーションに割り当てる CPU 時間が足りなくなる場合もあります。デバイスドライバの割り込みは基本的には毎回スケジューラを呼び出すべきですが、毎回スケジューラが実行されると時間が足りなくなるような場合は、複数回分の割り込みに対応する処理をまとめて行うようにシステムを構成する場合もあります。そのような場合スケジューラに飛ばずに直接割り込まれた部分に戻ったほうが都合がよい場合もあります。
これを実現するには、自分で関数の最後に機械語の命令の割り込みからの復帰命令をおいてもよいのですが、関数の先頭で実施されるレジスタの保存命令に対するレジスタの復元命令もきちんとしてやるなどの処理は必

要になります。そこで VxWorks では intConnect と intVecSet という二種類の API(システムコール)が用意されています。
intConnect は ISR に必要な定型処理は自動生成して、その途中でユーザーの作った関数を呼び出してくれるため、ユーザーはアルゴリズム的には（内容は）ISR だということはもちろん意識して作成しなければなりませんが、形式的手続き的には全く普通の関数を作るのと変わりなく作成することができます。一方 intVecSet はハードウェア的に割り込みベクタテーブルのある CPU の場合は単に割り込みベクタテーブルに指定の関数のアドレスを書き込むだけです。そのため登録する関数は内容だけでなく、形もちゃんと ISR であることを意識してその形式で作成する必要があります。
さらに VxWorks5.5.x では RISC 系でハードウェア的に割り込みベクタテーブルを持たない CPU の場合、VxWorks はソフトウェア的に割り込みベクタテーブルを提供することによりタスクのアプリケーションコードの互換性や作りやすさに配慮していました。ただしその場合は intVecSet を使用してもスケジューラの呼び出しを行っていました。その時は割り込み処理から直接呼ばれた部分に復帰させたい場合は、呼び出し部分と割り込み処理の戻り部分を自分で作成する必要はあります。
RTOS の中には、スケジューラに戻る割り込み処理を割り込み処理ルーチン(ISR)、直接呼ばれた部分に戻る割り込み処理を割り込みハンドラと区別しているものもありますが、VxWorks などほとんどの RTOS は特にその区別はありません。

コラム④ タスクと ISR

並列処理の実行単位であるタスクと割り込み処理の並列性、排他制御に関する特徴を次にまとめました。

	RTOS 使用	**RTOS 未使用***
並行性の実現手段	タスク	ISR
排他制御の手段	セマフォが代表だが他にも RTOS の提供する多くの手段がある	割り込み禁止および許可
優先度	ソフトウェア(RTOS)で付加する 自由につけることが出来る 一般的には 256(255)レベル レベル数は RTOS における重要なデータ構造の一つであるシステム READY キューの構造に影響を与える	外部デバイス(PIC など)の追加も含めて、ハードウェアで付加する場合が多い ハードウェアに優先度がない場合はソフトウェアで優先度をつける場合もある ハードウェアによりレベルは異なるが 8 レベル程度の物が多い
排他制御の（悪）影響	関連するタスクのみに限られる 関連しない優先度の高いタスクには影響を与えない 優先度逆転などによるデッドロックに注意	システム全体、全 ISR に及ぶ 優先度の高い ISR も停止させてしまう
待ち状態時の情報の保存場	TCB RTOS はアドレスを知っている	割り込みスタック RTOS 使用時も ISR に関しては RTOS の管

所		理外、当然アドレスも RTOS は知らない
特徴	多項目に及ぶが使い方を誤ってもシステム全体に影響を及ぼすことはまれ ただし優先度の高いタスクで使用する場合は要注意 またデッドロックにも要注意	比較的単純に使えるが、使い方を誤るとシステム全体の性能に影響を及ぼすため危険 使い方を誤ると最悪システムダウンを招く

* RTOS 使用時の割り込みも通常同じである