システムソフトウェアとオペレーティング・システム 86－7
(2001. 3. 5)

Real-Time Linux を利用したμITRON シミュレータの開発

山本厳淳　　木本芳男　　田中克範
FJB Web テクノロジー株式会社
〒112-0004 東京都文京区後楽 1-7-27
g-yamamoto@fjb.co.jp y-kimoto@fjb.co.jp ka-tanaka@fjb.co.jp

あらまし

我々は、Linux に対してリアルタイム拡張を行ったカーネルである RTLinux 上に、μITRON シミュレータを実装した。本シミュレータはデバッガ等のさまざまな GNU ツールを使用可能で、安価なμITRON アプリケーション開発環境であると共に、リアルタイム性を備えたμITRON 実行環境である。本論文ではμITRON と RTLinux の設計の違いを中心に、開発の概要について述べる。

キーワード　　リアルタイム、μITRON、RTLinux、Linux、GNU ツール、開発環境

Development of μITRON simulator using Real-Time Linux

Genjun Yamamoto　　Yoshio kimoto　　katsunori Tanaka
FJB WebTechnology LTD.
7-27 Koraku 1-Chome, Bunkyo-Ku, Tokyo 112-0004
g-yamamoto@fjb.co.jp y-kimoto@fjb.co.jp ka-tanaka@fjb.co.jp

Abstract

We have implemented μITRON simulator on the RTLinux which performes Real-Time extension to Linux. This simulator is a μITRON application development environment cheaply using various GNU tools, such as a debugger, and a μITRON environment with Real-Time capability. We describe the outline of development, focusing on the difference in design of μITRON and RTLinux.

key words　　real-time, μITRON, RTLinux, Linux, GNUtools, development environment

1 はじめに

従来、研究や開発の現場では機器制御や計測を目的としてPCが利用されてきた。こうした場では、できるだけ安くかつ簡単にシステムを構築したいという要求から、PC上で動作するオープンソースソフトウェアのリアルタイムOSが利用されてきた。

いくつか存在するそうしたリアルタイムOSの中で、プログラム開発やデバッグ、GUIインターフェースの開発などに豊富なフリーソフトウェアのツールを利用することが可能なものとして、RTLinux、RTAIやART-Linuxなどをあげることができる。

今回、μITRON仕様OSのために書かれたプログラムをこうした環境で実行したいという要望があったのだが、既存のリアルタイムOSの中には、上で述べたようなツール群に加えて、μITRON仕様のシステムコールを使えるものは存在しなかった。そこで、既存のリアルタイムOS上にμITRONのシミュレータ環境を実装することにした。基盤となるリアルタイムOSとしては、入手可能なドキュメントが豊富だったことなどから、RTLinuxを採用した。

2 仕様およびソフトウェアのバージョン

本シミュレータではμITRON仕様3.02で定義されるシステムコールのサブセットを実装した。実装したシステムコールは次の通りである。

タスク管理 ext_tsk, rot_rdq, get_tid, sta_tsk, ter_tsk, chg_pri, rel_wai, ref_tsk

タスク付属同期 slp_tsk, wup_tsk, sus_tsk, rsm_tsk, tslp_tsk

同期・通信 sig_sem, wai_sem, preq_sem, snd_msg, rcv_msg, prcv_msg, set_flg, clr_flg, wai_flg, pol_flg

拡張同期・通信 psnd_mbf, rcv_mbf, prcv_mbf

割込み管理 loc_cpu, unl_cpu, ret_int, dis_int, ena_int

時間管理 set_tim, get_tim, dly_tsk

また、本シミュレータで使用したソフトウェアのバージョンは次の通りである。

- RTLinux 3.0-pre8（Linuxカーネル バージョン 2.2.17）
- Linuxディストリビューション Turbo Linux WorkStation 6.0

以降、特に断らない限りそれぞれのバージョンはこれらを指す。

3 RTLinux

RTLinuxはLinuxカーネルとハードウェアの間で独自のスケジューリング方式でリアルタイムタスクを制御する。ここでLinuxカーネルは最低優先度のタスクであるかのように実行される。リアルタイムタスクはカーネルにダイナミックリンクされるモジュールとしてinsmodコマンドやrmmodコマンドにより制御される。

リアルタイムタスクから直接デバイスを管理する場合にはデバイスドライバを新たに書き起こす必要がある。Linuxカーネルは常にリアルタイムタスクより後回しにされる。X Window Systemなどのインターフェースやリアルタイムタスクで管理しないデバイスはLinuxカーネルによって管理されるが、リアルタイムタスクの実行状況との関係には注意を要する。さらに、RTLinuxではリアルタイムタスクはカーネル空間で実行されるため、メモリ保護の対象とならない。リアルタイムタスクのバグによりシステムがクラッシュする可能性がある。

4 シミュレータの実装

μITRONシミュレータとして、RTLinuxを基盤にμITRON仕様に規定されているシステムコールのサブセットを実装した。実装に当たってはμITRON仕様に相当する機能がRTLinuxに存在する場合にはこれを利用し、存在しない場合には必要な機能追加をおこなった。以下、実装の詳細について述べる。

4.1 タスク状態

μITRONのタスク状態遷移図を図1に、RTLinuxのタスク状態遷移図を図2に示す。μITRON仕様では広義の待ち状態として待ち状態、二重待ち状態、強制待ち状態を区別している。これに対し、RTLinuxでは広義の待ち状態に対応するサスペンド状態は特に細分化されていない。また、μITRON仕様の休止状態に対応するタスク状態はRTLinuxには存在しない。シミュレータでは、μITRON仕様のタスク状態のうち、実行状態、実行可能状態、(狭義の)待ち状態、二重待ち状態、強制待ち状態、休止状態を実装した。

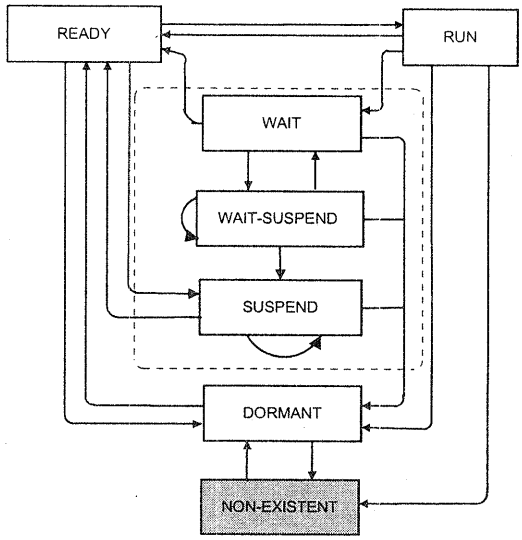

図1: μITRONにおけるタスク遷移図。点線内は広義の待ち状態であり、3つの状態に細分化されている。

4.1.1 待ち状態の実装

待ち状態、二重待ち状態、強制待ち状態を区別するため、タスクコントロールブロック(TCB)に相当するRTLinuxの構造体にμITRONタスク状態を管理するメンバーを追加した。タスクを広義の待ち状態に遷移させるシステムコールの実装では、タスクをサスペンド状態に遷移させるのと合わせてこのメンバーを更新し、待ち状態を管理した。この追加メンバーは次に述べる休止状態を管理するためにも用いた。

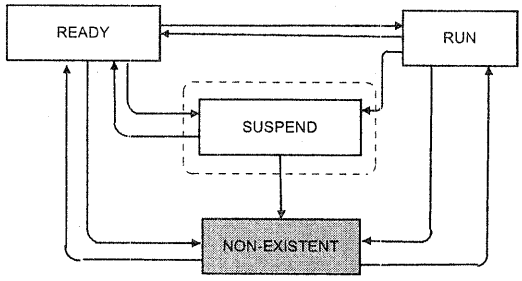

図2: RTLinuxにおけるタスク遷移図。待ち状態はSUSPEND状態しかなく、休止状態もない。

4.1.2 休止状態の実装

μITRON仕様の休止状態をRTLinuxのサスペンド状態でシミュレートすることとした。休止状態に関わるタスクの生成、起動と終了のシミュレートの

実装に関しては注意が必要であった。

タスクの起動　μITRONではタスクの生成と起動にはそれぞれ異なるシステムコールを用いる。対して、RTLinuxではタスクの生成と起動が同一のシステムコールでおこなわれる。さらに、このシステムコールは最低優先度のLinuxコンテキストからしか発行できないという制限がある。つまりRTLinuxでは、リアルタイムタスクから新しいタスクの生成や起動をおこなうことはできない。そこで、リアルタイムタスクの起動をおこなうinsmodコマンドの発行の時点をμITRON環境の起動時と定義し、この時点ですべてのタスクを生成することとした。すなわち、本シミュレータは、μITRONシステムとしては、システム起動時にすべてのタスクが生成されるインプリメントとなる。加えて、タスク起動システムコールの引数であるタスク起動コードについてはスタックの内容を書きかえることで対応した。

タスクの終了　μITRON仕様では、一度起動して実行可能状態になったタスクが終了すると、TCB等のタスク状態を表すものが初期化され、休止状態に移る。タスク終了時の休止状態をシミュレートするため、タスク生成システムコールにスタックの状態やTCB等、タスクの初期状態の退避、保存処理を追加した。また、タスクを起動するシステムコールについて、対象がタスク終了後の休止状態にあるタスクの場合にTCB情報の復帰処理を追加した。

4.2　スケジューリング

μITRONとRTLinuxはスケジューリングの際に重要な役割を果たすレディキュー（RTLinuxでは単なるリスト）の考え方が大きく異なる。まず、μITRONでは、実行可能状態のタスクからなる優先度毎のレディキューの考え方によってタスクが管理される。これに対しRTLinuxでは、タスク状態、優先度に関わらずすべてのタスクがつながった単方向のリンクリストによってタスクが管理される (図3、図4)。

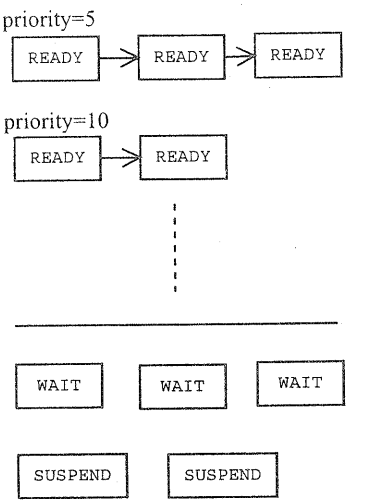

図 3: μITRONにおけるレディキューの模式図。実行可能状態のタスクにより優先度毎のレディキューが形成される。

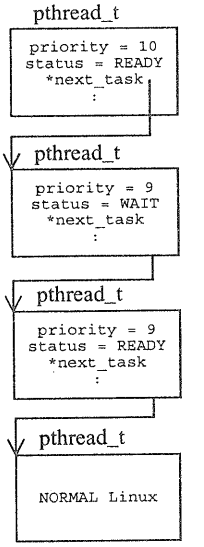

図 4: RTLinuxにおけるリンクリストの模式図。優先度、タスク状態が混在している。Linuxコンテキストは1タスクとして扱われる。

スケジューリングに当たっては、実行可能状態のタスクのうち最も優先度の高いタスクが複数ある場合、μITRON仕様とRTLinuxはともにレディキューないしリンクリストの先頭に近いものが次に実行されるタスクとして選択される。ここで、μITRONではFCFS（First Come First Service）方式によるスケジューリングをおこなうため、タスクが実行可能状態に移るときにレディキューの最後尾につながる。つまり、待ち状態から実行可能状態へ移る場合などタスクの状態が遷移するときにレディキューの中での順番が変わる。RTLinuxでは、タスクの生成時に形成されたリンクリストはタスクの終了（本シミュレータではrmmodによるμITRONシステム終了時に相当）まで固定され、タスク状態が変わってもリンクリスト上のタスクの位置は変わらない。すなわち、RTLinuxとμITRONは異なるスケジューリング方式を取る。

このため、本シミュレータではFCFS方式によるスケジューリングを実現する必要があった。実装に当たっては、将来のRTLinuxのバージョンアップへの対応などを考慮した結果、優先度毎のレディキューの考え方を取り入れるなどのRTLinuxのスケジューラの大規模な改造は避け、タスクを実行可能状態に移すシステムコール内で対象タスクをリンクリストの最後尾に移すこととした。

RTLinuxのリンクリストはタスク状態、タスクの優先度の異なるタスクが混在しているため、スケジューリングに当たって、リスト中の全タスクを検索する必要がある。このため、特にタスク数の増加時にスケジューリングのオーバーヘッドが増加するということが実装、性能テストを通じて分かった。本シミュレータではRTLinuxのリンクリストをそのまま利用したため、このオーバーヘッドは同様に発生することになる。

4.3 同期通信機能の実装

セマフォ、イベントフラグ、メールボックス、メッセージバッファといった同期通信関連のシステムコールは、RTLinuxで同期通信APIを提供しているIPCモジュールをベースに実装を行った。

なお各機能におけるオブジェクトの作成、初期化は、μITRON環境の起動時に実行されるように実装を行った。

4.3.1 セマフォ

先のIPCモジュールで提供されているセマフォ関連のAPIである、rt_sem_*をベースに実装を行った。μITRONとの機能的相違点は、μITRONの場合、セマフォ待ちから解除され実行可能状態に移るタスクは待ちタスクリストの先頭タスクであるが、RTLinuxでは待ちタスクリスト中の最も高いプライオリティを持つタスクであるという点にある。この点を修正するために、IPCモジュール内の待ちタスクを検索する部分に変更を加えた。

μITRONでは優先度ベースのタスク検索は拡張機能に属しており、FIFOベースでの検索と選択できる必要がある。IPCモジュールでの待ちタスク検索は、FIFOベースの検索と優先度ベースの検索を選択できるようにはなっておらず、そのため待ちタスクを全て検索して最も優先度の高いタスクを探していたのを、最も先に待ちタスクリストにはいったものを実行可能状態にするように変更した。

4.3.2 イベントフラグ

IPCモジュールではイベントフラグに相当するAPIは用意されておらず、一から実装を行った。

4.3.3 メールボックス

IPCモジュールで提供されているメッセージキュー関連のAPIである、rt_mq_*をベースに実装を行った。IPCモジュールにおけるメッセージキュー機能において提供されているのは、固定長メッセージ(メッセージ本体)をリングバッファにのせる機能である。

ITRONでのメールボックスはメッセージの先頭アドレスをタスク間でやりとりするための機能であ

るので、メッセージへのポインタを固定長メッセージとみなし、IPCモジュールでのメッセージキューにのせて、タスク間でポインタのやりとりを行うように実装を行った。

またメッセージキューに対する待ちタスクリストの検索も、セマフォに対するそれと同じ形式であったので同様の変更を加えた。

4.3.4 メッセージバッファ

IPCモジュールで提供されているメッセージキュー関連のAPIである、rt_mq_*をベースに実装を行った。IPCモジュールで提供されているメッセージキューは上述のように固定長を扱うものである。メッセージバッファは可変長メッセージを扱うものであるので、メッセージバッファの作成化を行う際に、そのメッセージバッファで扱うメッセージの最大長を指定した上で、疑似的に可変長メッセージを扱うようにした。

具体的には実際にメッセージキューにのせるメッセージに対して、メッセージサイズを情報として追加してからメッセージキューにのせるように変更を行った。メッセージを受け取るときには、各メッセージに付加したメッセージサイズを元にメッセージの受渡しを行う。

4.4 割込みハンドラ

μITRON仕様では、割込みハンドラ中でタスク状態やタスクの優先度を変更するシステムコールが発行された場合、割込みハンドラが終了し、タスクに実行が移る時点でスケジューリングがおこなわれる。また、割込みがネストした場合、スケジューリングがおこなわれるのは、ネストして実行されている割込みハンドラがすべて終了し、タスク部に実行が移る時点となる（遅延ディスパッチの原則）。これに対し、RTLinuxでは割込みハンドラの終了時にスケジューリングはおこなわれない。このため、割込みハンドラが終了すると割込みの発生時に実行されていたタスクに実行が移る。本シミュレータでは、1. 割込みのネストをカウント、2. 割込みハンドラ本体を実行、3. スケジューリングを実行、という一連の処理を実行するラッパーを実装し、これを割込みハンドラとしてRTLinuxに登録することとした。ラッパーの終了時に割込みネストのカウントを判定し、ラッパーの終了後に実行されるのがタスクである場合にスケジューリングをおこなうようにした。また、タスク状態を変更する各システムコールでも割込みネストのカウントを判定し、割込みハンドラの実行中である場合にはタスク状態を管理するTCB情報の変更のみおこない、スケジューリングはおこなわないこととした。なお、本シミュレータで実装したシステムコールのうち、μITRON仕様で明示的に禁止されていないものは、割込みハンドラ本体の中から発行することが可能である。

5 評価

以上のような実装を行ったものに対して、その性能をはかるために2種類の計測を行った。両計測とも計測を行ったマシンは、タスクを立ち上げるのに十分なメモリを積んでおり、CPUはPentium 166MHzのものを使った。

5.1 周期タスクについて

一定負荷をかけた状態で、周期タスクを複数動作させた際の、タスクの立上りの遅れを計測した。各タスクの周期は、dly_tskを連続して実行することで発生させた。負荷タスク、周期タスクは以下の条件で動作させた。

- 負荷タスクは$50ms$の周期で動作させ、各周期で1から100まで計数させた。これを100タスク実行した場合、200タスク実行した場合、1000タスク実行した場合のそれぞれについて計測を行った。
- それぞれ異なる周期を持つ4種類のタスクを

負荷タスク数	評価タスク	性能評価 $[ms]$
100	taskA	0.27
	taskB	0.30
	taskC	0.50
	taskD	0.50
200	taskA	0.50
	taskB	0.35
	taskC	0.45
	taskD	0.50
1000	taskA	1.2
	taskB	2.1
	taskC	1.6
	taskD	1.5

表 1: 200 タスク程度であれば $500\mu s$ 程度の遅延に収まる。1000 タスクになると $1ms$ をこえた遅延が起こる。

同時に実行した。周期ごとに DIO のビットを ON/OFF する動作を行わせた。各タスクの遅延はそれぞれ、$10ms$(taskA)、$100ms$(taskB)、$1000ms$(taskC)、$10000ms$(taskD) とした。なおカッコ中は各タスクに便宜上つけた名前である。

これらのタスクを起動させ、周期タスクが DIO を 5 回 ON もしくは OFF するのに要した時間を計測し、5 で割ったものから各周期を引くことで、周期タスクの立ち上がりの遅れを計測した。計測結果は表 1 の通りである。これらの結果は、我々が要求していた仕様を満たすものであった。

5.2 ディスパッチ性能

実時間タスクのスイッチ性能の計測を行った。計測方法は以下の通りである。

1. 計測するタスク数だけ休止状態のタスクを生成する。
2. 最初に起動されたタスクが DIO の ON を行う。
3. 次のタスクを起動し、終了する。
4. 以降、起動したタスクは、次のタスクを起動しては終了する。
5. 最後のタスクが DIO を OFF する。
6. DIO の ON/OFF 間の時間を計測した結果を最初に生成したタスク数で割り、1 タスク当たりのディスパッチにかかる時間とした。

タスク数	時間 $[ms]$/タスク数
100	0.19
200	0.36
500	1.46

表 2: タスク数が 200 程度であれば μs のオーダになる。200 タスク時の結果と 500 タスク時の結果の違いの大きさは、今後の課題となる。

全てのタスクは sta_tsk によって次のタスクを起動した。タスク数は 100 タスク、200 タスク、500 タスクのそれぞれについて計測を行った。結果は表 2 の通りである。この結果は我々の要求する仕様を満たすものであった。

6 むすび

RTLinux を利用した μITRON シミュレータ環境を実装したことについて説明した。実装したシステムコールは μITRON 仕様のサブセットではあるが、リアルタイム OS としての μITRON 仕様を特徴付けるタスク管理、割込み管理、同期通信などのシステムコールを実装できた。今回実装しなかったシステムコールの実装、およびタスク数増加時のリアルタイム性能の向上は今後の課題である。なお、本シミュレータは著作権の関係上、そのままでは第三者にソースコードを公開できない。今後、再度コードを書き起こすなどして、オープンソースソフトウェアとして公開する方向を模索したい。その際には、

開発、デバッグ環境の充実についても合わせて取り組みたい。

参考文献

[1] Michael Barabanov: "Getting started with Real-Time Linux", rtlinux-3.0-pre8, 2000

[2] RTLinux.org Home Page: "RTLinux Home Pages", http://www.rtlinux.org/, 2000

[3] ALESSANDRO RUBINI 著 山崎 康宏, 山崎 邦子 共訳: "LINUX デバイスドライバ", オライリー・ジャパン, 1998

[4] トロン協会 ITRON 部会: "ITRON Project Home Page", http://tron.um.u-tokyo.ac.jp/TRON/ITRON/home-j.html, 1999

[5] 社団法人トロン協会 ITRON 専門委員会 編集: "ITRON 標準ガイドブック 2", パーソナルメディア, 1994.

[6] 社団法人トロン協会 編集・発行: "μITRON3.0 標準ハンドブック改定新版", パーソナルメディア, 1998.

[7] "インターフェイス増刊 技術者のための UNIX 系 OS 入門", CQ 出版社, 7/1/00