システムＬＳＩ設計技術　105－8
（２００２．３．５）

# μITRON4.0仕様リアルタイムOSにおけるミューテックスの実装

樋口 正雄　宮内 新　石川 知雄　　高田 広章
武蔵工業大学大学院　　豊橋技術科学大学 情報工学系

**概要**　μITRON4.0仕様では新たに「ミューテックス」機能に関する規定が追加された．これは排他制御に伴う「優先度逆転現象」の防止に有効とされる優先度継承・上限プロトコルをサポートする排他制御機構である．同仕様では原義の優先度継承・上限プロトコルに忠実な「厳密な優先度制御規則」に加え「簡略化された優先度制御規則」が定義され，どちらを採用するかは実装依存としている．このような仕様の有効性及び妥当性の検証，制御規則の選択基準の提供を目的として，実カーネル上に両者の制御規則のミューテックスを実装し，優先度制御規則による性能の差異を定量的に評価した．結果，タスクによるミューテックスのロックがネストするような環境においては，簡略化規則の採用によってオーバーヘッド増加の抑制が可能であった．

# Implementation of Mutex on μITRON4.0-specification Real-time Kernel

Masao Higuchi　Arata Miyauchi　Tomo Ishikawa　　Hiroaki Takada
Musashi Institute of Technology　　Toyohashi University of Technology

**Abstract**　μITRON4.0-specification, established in 1999, includes the definition of mutex function, added for the hard real-time systems. Mutex supports priority inherita nce protocols and priority ceiling protocols, approach to solve the problem of (uncontrolled) priority inversion accompanying mutual exclusion. μITRON4.0-specification defines two versions of priority control rule : strict rule and simplified rule. In this paper, we implemented the mutex function followed in both priority control rules on μITRON4.0-specification kernel and evaluated the dif ferences of performance. As the result, in the case that task locks multiple mutexes, the simplified priority control rule enables to suppress the overhead of mutex operation.

## 1　はじめに

機器組込み用ソフトウェアシステムの開発において，生産性や保守性の向上を目的としてリアルタイムオペレーティングシステムを用いることが一般的となっている．その適用範囲も，当初は工場の生産ラインの制御，交換機などといった大規模なシステムに限られていたが，近年ではOA機器や家電といった民生機器にも利用されるようになり，現在では小型情報通信端末，輸送機器制御，医療機器制御などといった分野にまで広がりを見せている．それに伴い，リアルタイムOSに対する要求も次第に変化してきており，わが国における機器組込み用リアルタイムオペレーティングシステム(RTOS)のデファクトスタンダードとなっているITRONも数回に渡って改訂が行われてきた．最新のμITRON4.0仕様(1999年)では，ITRON仕様OSのハードリアルタイムシステムへの適用への要求を取り込んだ結果として，ミューテックスと呼ばれる機能が追加されている．このμITRON4.0仕様に基づいたカーネルの実装により仕様の評価を行うと同時に，研究・教育用プラットフォームの提供を目的とした「TOPPERS/JSPカーネル」が豊橋技術科学大学組込みリアルタイム研究室を中心としたTOPPERSプロジェクト[1]により開発された．

本稿では，μITRON4.0仕様の評価及び商用カーネル開発者のためのリファレンス実装として提供することを目的として，ミューテックス機能をTOPPERS/JSPカーネル上に実装し性能評価を行ったので報告する．

[1]Toyohashi OPen Platform for Embedded Real-time Systems ... 組込みリアルタイムシステム構築の基盤となるフリーソフトウェアの開発を行うプロジェクト

## 2 μITRON4.0のミューテックス

### 2.1 優先度逆転現象

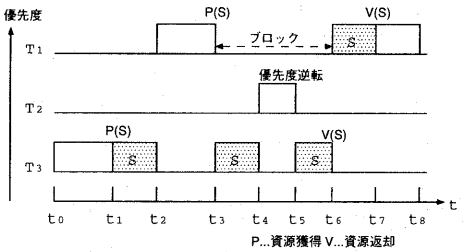


図 1: 優先度逆転現象

図1において，優先度の最低のタスク$T_3$が，クリティカルな資源S(セマフォによって排他制御されている) を占有している．この状態で優先度の最高のタスク$T_1$は資源Sを要求しても獲得できず，低優先度のタスク$T_3$に実行を待たされる (ブロックされる)．その間に中間の優先度のタスク$T_2$が起動すると，$T_1$よりも低優先度であるにもかかわらず先に実行されてしまう．このような現象を「(上限の無い) 優先度逆転」という．

ここでのタスク$T_1$の実行時間は$T_2$の実行時間に依存することとなり，実行時間の予測可能性 (Predictability) に影響を及ぼす．

このような現象に対する有効な手法として，優先度継承 (priority inheritance) プロトコル，優先度上限 (priority ceiling) プロトコルが提唱されている [2].

### 2.2 優先度継承プロトコル・優先度上限プロトコル

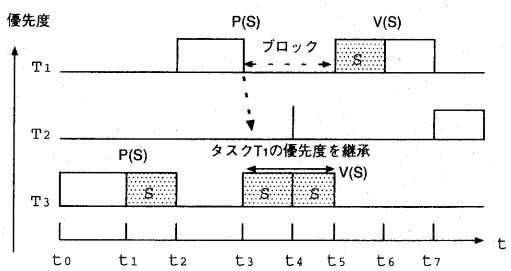


図 2: 優先度継承プロトコル

図1のような場合に優先度継承プロトコルを適用すると，図2のようになる．タスク$T_1$がクリティカルな資源Sを獲得しようとして待ち状態になる ($T_3$にブロックされる) 場合，資源Sを獲得しているタスク$T_3$に一時的に$T_1$と同じ優先度を割り振る．結果として，タスク$T_2$が$T_1$より先に実行されるという現象の回避が可能となる．

しかしながら，優先度継承プロトコルによって，優先度逆転現象が回避された環境においても，デッドロックが発生した場合にはタスクの予測可能性に影響を及ぼす．このような状況にも対応可能な手法が優先度上限プロトコル[2]である．

タスク$T_1$：セマフォ$S_1$→セマフォ$S_2$　タスク$T_2$：セマフォ$S_2$→セマフォ$S_1$

という順にロックを行うタスクが存在する環境を想定する．これらのタスクの同時実行はデッドロックを招く可能性がある．

図3では，このタスクセットに優先度上限プロトコルを適用し，デッドロックを回避する例を示している．クリティカルな資源には予め上限優先度 (その資源を獲得する可能性のあるタスク中の最高の優先度) を定義する．(この例においてはセマフォ$S_1$・$S_2$とも上限優先度はタスク$T_1$の優先度値となる．) あるタスクが資源を獲得する際には，一時的にそのタスクの優先度を上限優先度まで上昇させる．

このような取り決めを行うと，複数の資源を複数のタスクが共有する際，クリティカルセクションがオーバーラップしないため，デッドロックが回避される．また，低優先度のタスクによって実行がブロックされる回数が1回に限られるため，実行時間の予測可能性が保証される．

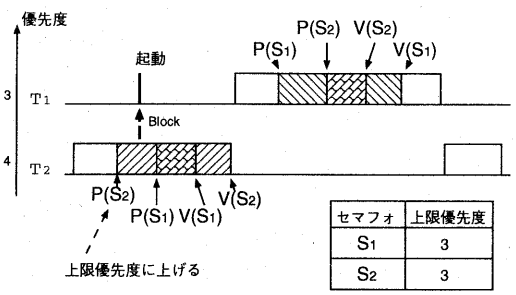


図 3: 優先度上限プロトコル

[2] μ ITRON4.0 仕様で定義される優先度上限プロトコルは広い意味でのものであり，初めに提案されたものとは若干異なるものである．

### 2.3 ミューテックスの導入

応答時間に対する厳しい時間制約をもつハードリアルタイムシステムへのITRON仕様OSの適用を考えた際，優先度継承・上限プロトコルのサポートによる優先度逆転の回避が求められる．この様な背景から，従来から存在した排他制御機構であるセマフォに加え，優先度継承・上限プロトコルをサポートする排他制御機構の必要性が生じ，ミューテックス(Mutex)の規定が加えられた．

また，携帯電話や小型情報機器等といった製品のソフトウェアシステムにおいては，高度なGUIやネットワーク機能をJavaで開発するケースが増加している．ハードウェアに対する抽象度が高いJavaプログラムからは，タスクレベル，ハードウェアレベルでの振舞が捉えにくいため，優先度逆転が生じないように考慮したプログラミングが困難になる．このような環境下では，排他制御機構としてミューテックスを予防的に利用する事が効果的であると考えられる．

### 2.4 優先度制御規則

排他制御における優先度継承・上限プロトコルの実現に伴い，オーバーヘッドの増加が予想される．例を挙げれば，ITRONとJavaのハイブリッドアプローチであるJTRON環境においては，Javaスレッドにおける排他制御(sychronized文)のネストにより，これと対応するITRONのミューテックス機能のオーバーヘッドの著しい増加が懸念される．$\mu$ ITRON4.0仕様では，本来の優先度継承・上限プロトコルの定義に従った「厳密な優先度制御規則」に加え，上昇させた優先度をもとに戻すタイミングを限定することによりオーバーヘッドの低減を狙った「簡略化した優先度制御規則」という2種類の優先度制御規則を規定し，どちらを採用するかは実装依存としている．

この2種類の制御規則のミューテックスを実装し性能を比較した上，仕様の有効性を検証し，ミューテックス実装の際の選択基準を提供することを主な目的として性能評価を行った．

## 3 ミューテックス機能の実装

### 3.1 開発環境

CPUとして日立製SH7709A(SH-3 クロック周波数133MHz)を搭載した評価用ボード“Solusion Engine”を使用した．ベースとなるカーネルには前述のTOPPERS/JSPカーネルを採用し，Linux-PC環境上でGCCにてクロスコンパイルし，GDBにてプログラムを転送・リモートデバッグを行う．

### 3.2 データ構造

新たにTOPPERS/JSPカーネル上に実装したミューテックス管理ブロック(MTXCB)の構造を示す．設計に当たり，既存の同期・通信機能で共通に用いられているデータ構造を可能な限り継承することとした．これは，既存のセマフォとの比較を行うことを念頭においたためである．

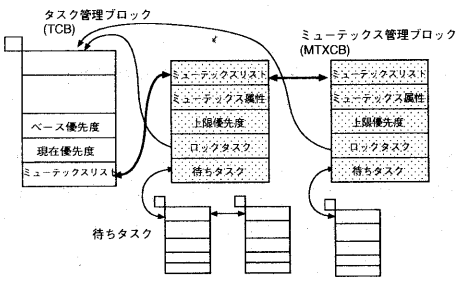


図 4: ミューテックス管理ブロックとタスク管理ブロック

**ミューテックス管理ブロック** ロックを獲得しているタスクをミューテックス側から管理する点がセマフォと大きく異なる．

ミューテックス属性にTA_INHERITを指定すると優先度継承プロトコルを，TA_CEILINGを指定すると優先度上限プロトコルをサポートする．優先度上限プロトコルを適用する際，各ミューテックスごとに上限優先度(ceiling priority)を定義しておく必要がある．

**タスク管理ブロック内のフィールド** タスク優先度をベース優先度(ユーザーアプリケーションで与えられる優先度)と現在優先度(優先度継承で変化する実効的な優先度)とに区別して管理するよう改良を行った．また，タスク側からもロックしているミューテックスを管理する必要があるため，双方向リンクの「ミューテックスリスト」の追加も行った．

### 3.3 厳密な優先度制御規則の実装

#### 3.3.1 ミューテックス操作

**ミューテックスのロック (loc_mtx サービスコール)** ミューテックスのロック操作 (リストへの挿入) に加え，プロトコルの定義に従い必要に応じてミューテックスをロックしているタスクの現在優先度を上昇させる．

**ミューテックスの解放 (unl_mtx サービスコール)** ミューテックスの解放 (リストからの切り離し) に加え，タスクがロックしている残りの TA_INHERIT ミューテックスの待ちタスクと TA_CEILING ミューテックスを探索し，現在優先度を下げるべきか判断する．

#### 3.3.2 ミューテックス導入に伴う既存機能への処理追加

TA_INHERIT 属性のミューテックスのロックを待つタスクに対しベース優先度変更 (chg_pri サービスコール)，待ち状態強制解除 (rel_wai サービスコール)，タイムアウト処理を行うと，ミューテックスをロックしているタスクの現在優先度の変更が必要になる可能性があり，ミューテックスの解放のときと同様の探索操作が必要となる．

### 3.4 優先度制御処理の最適化

前述のように，unl_mtx,chg_pri,rel_wai サービスコール処理では優先度値の探索が必要となるが，実際には探索を行わずとも優先度値が自明である場合が存在する．実装の第 2 段階として以上のような場合を考慮し，不要な探索処理の削減を行ったものを性能評価の対象とする．

### 3.5 簡略化した優先度制御規則の実装

ミューテックスのロックに伴い一時的に上昇させた優先度を下げるタイミングは以下のとおりである．

- ミューテックスの解放
- タスクのミューテックス待ち状態の強制解除
- TA_INHERIT 属性ミューテックス待ちのタスクの優先度変更 (降下)

「簡略化した優先度制御規則」では，優先度を下げるタイミングをタスクが全てのミューテックスを解放した時に限定しているため，探索に要していたオーバーヘッドの削減が見込まれる．

この制御規則の本質に関わる限りは，タスク側からロックしているミューテックスを管理する必要は無い．しかしながら，実際にはエラーチェック，タスク終了時のミューテックスの自動解放の実現のために，ミューテックスリストが必要であった．このため，データ構造に関しては厳密な制御規則と同一になり，ミューテックスリスト維持操作の省略が不可能であった．

## 4 評価

### 4.1 ミューテックス処理サービスコールの応答性能

ミューテックス処理サービスコールの応答性能を，以下のようなテストプログラムにより測定した．

1. 対象ミューテックスが他のタスクによってロックされていない場合の loc_mtx サービスコールの応答時間．
2. 同上．ただし，対象ミューテックスが TA_CEILING 属性であり，タスクの優先度を上限優先度に上げる操作も含む．
3. 対象ミューテックスが既に他のタスクによってロックされていて，待ち状態に入る場合の loc_mtx サービスコールの応答時間．タスクディスパッチ操作も含む．
4. 同上．ただし，対象ミューテックスが TA_INHERIT 属性であり，先にロックしているタスクが待ちタスクの優先度を継承する操作も含む．
5. 同上．対象ミューテックスを先にロックしているタスクが別のミューテックス待ち状態であった場合，4. の優先度継承に加え，そのミューテックスをロックしているタスクも優先度を継承する．(遷移的な優先度継承)
6. unl_mtx サービスコールの応答時間．ロック待ちタスクは存在しない．
7. unl_mtx サービスコールの応答時間．ロック待ちタスクの待ち解除操作を含むが，タスクディスパッチは生じない．
8. unl_mtx サービスコールの応答時間．ロック待ちタスクの待ち解除操作とタスクディスパッチも

含む.

9. 同上. 他にもロックしているミューテックスがある場合.

測定は前述のSolutionEngine(SH3, 133MHz), キャッシュOFFで行った. 表1に各テストプログラムにおけるサービスコールの応答時間を, 対応するセマフォ処理の応答時間とともに示す.

表 1: ミューテックス処理サービスコール応答時間 (キャッシュOFF 単位: $\mu$ s)

| プログラム | セマフォ | 厳密な規則 | 簡略化規則 |
|---|---|---|---|
| 1 | 12.2 | 18.7 | 18.7 |
| 2 | - | 36.0 | 36.0 |
| 3 | 58.5 | 62.9 | 62.9 |
| 4 | - | 76.8 | 76.3 |
| 5 | - | 89.7 | 89.7 |
| 6 | 14.6 | 30.2 | 28.3 |
| 7 | 30.4 | 49.9 | 48.0 |
| 8 | 45.3 | 89.8 | 61.9 |
| 9 | - | 102.2 | 40.8 |

ロックするミューテックスが単数の場合には, 両者の制御規則との間に応答性能の本質的な違いが見られなかった. また, ミューテックスロック解除 (unl_mtx) の応答時間をセマフォ資源の返却 (sig_sem) と比較すると約2倍のオーバーヘッドを要している. しかし, 複数のミューテックスをロックする場合 (プログラム9) には, 優先度探索変更操作の削減による応答性能の改善が見られ, セマフォとほぼ同等の応答性能を示している. この効果を確認するため, 複数のTA_CEILING属性ミューテックスをロックしている際のunl_mtxサービスコール処理1回 (ミューテックス1個のアンロック) を測定した (図5). このように, 厳密な優先度制御規則においてはタスクがロックしているミューテックス数に比例して, 応答時間が増加するということがわかる. 一方, 簡略化した優先度規則においては, ミューテックス数に依存せずに一定のオーバーヘッドで処理が可能である.

図6は, タスクがロックしている複数のミューテックスを全て解放するまでの所要時間である.

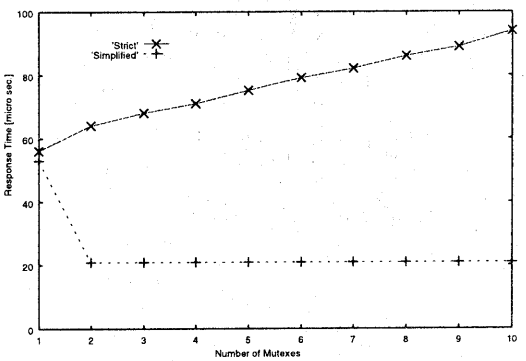


図 5: ロック中ミューテックス数と unl_mtx サービスコール処理時間 (以下, Strict…厳密な制御規則 Simplified…簡略化規則)

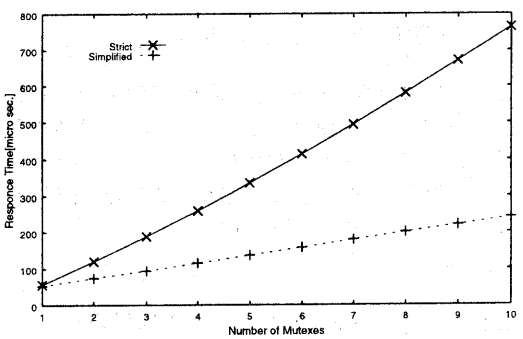


図 6: ロック中ミューテックス数とアンロック所要時間 (図5のデータの総和から算出)

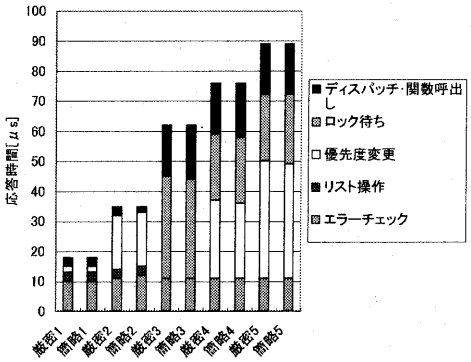


図 7: loc_mtx サービスコール処理の内訳

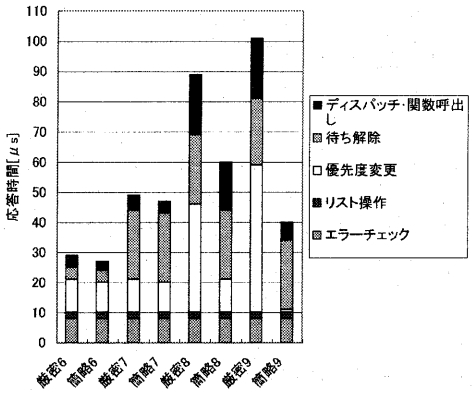


図 8: unl_mtx サービスコール処理の内訳

## 4.2 サービスコール処理を構成する内部処理のオーバーヘッド切り分け

ミューテックスのロック，解放に含まれる処理 (エラーチェック，リスト操作，タスク待ち操作，優先度制御 (変更) など) を分割して測定を行った (図 7,8). ミューテックス処理オーバーヘッド増加の最大要因は優先度制御操作であることがわかる．その一方で，ミューテックスのロック (loc_mtx) で規定されるエラーチェックに 10 $\mu$ s 要しており，プログラム 1 においては，サービスコール処理の半分以上をエラーチェックに費していることになる．

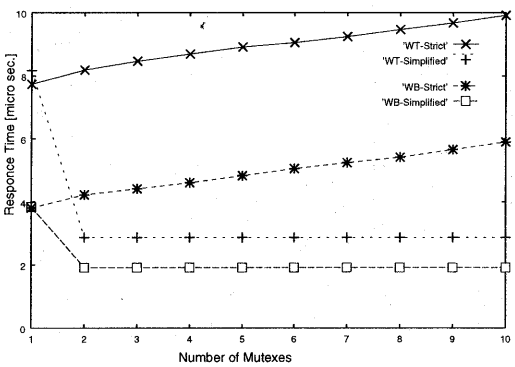


図 9: ロック中ミューテックス数と unl_mtx サービスコール処理時間 (キャッシュ有効)

## 4.3 CPU キャッシュの効果

続いて，SH-3 のキャッシュ機能を有効にした状態でミューテックス機能のサービスコールの応答性能を測定した (表 2，図 9). 複数のミューテックスをロックする環境でのアンロックの応答時間は，簡略化による効果が縮小している事が判明した．

## 4.4 既存サービスコールの応答性能

ミューテックスをロックするタスク及びロックを待つタスクの優先度がミューテックスの上限優先度より高く変更されると，優先度上限プロトコルの定義に反する．chg_pri サービスコール (タスクのベース優先度変更) では，処理の前にこのチェックを行うことが規定されているため，ロックしている TA_CEILING 属性ミューテックスの上限優先度の探索が必要となる．ミューテックス数が増えるに従い，このオーバーヘッドによるサービスコールの応答性能の悪化が予想される．この検証のため，複数の TA_CEILING 属性ミューテックスをロックしているタスクに対するベース優先度変更の応答性能を測定した (図 10)． 結果，ロックしているミューテックスが 1 つ増える毎に約 2[$\mu$ s] 応答時間が増加し，ミューテックス数が 10 の場合は応答時間が 2 倍に達するという結果となった．

表 2: キャッシュ方式とミューテックス処理サービスコール応答時間 (単位: $\mu$ s)(WT…ライトスルー方式 WB…ライトバック方式)

| | 厳密な規則 | | | 簡略化規則 | | |
|---|---|---|---|---|---|---|
| プログラム | 無効 | WT | WB | 無効 | WT | WB |
| 1 | 18.7 | 2.9 | 1.4 | 18.7 | 2.9 | 1.4 |
| 2 | 36.0 | 5.3 | 2.9 | 36.0 | 5.3 | 2.9 |
| 3 | 62.9 | 9.1 | 4.3 | 62.9 | 9.1 | 4.3 |
| 4 | 76.8 | 11.6 | 4.8 | 76.3 | 12.5 | 6.2 |
| 5 | 89.7 | 13.0 | 5.8 | 89.7 | 13.4 | 6.2 |
| 6 | 30.2 | 3.8 | 2.4 | 28.3 | 3.8 | 1.9 |
| 7 | 49.9 | 7.7 | 3.8 | 48.0 | 9.1 | 5.3 |
| 8 | 89.8 | 13.4 | 5.8 | 61.9 | 11.0 | 5.3 |
| 9 | 102.2 | 14.4 | 6.2 | 40.8 | 9.6 | 6.2 |

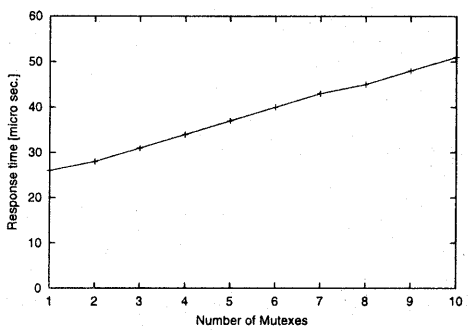


図 10: ロック中ミューテックス数と chg_pri サービスコール応答時間

# 5 考察

## 5.1 優先度制御規則の選択

### 5.1.1 簡略化規則が有効であると考えられるケース

アプリケーションのタスクによるミューテックスのロックがネストする場合，アンロック操作を行う時点でのロックしているミューテックス数に比例して応答時間が増大する．しかし，簡略化規則の導入により，アンロック所要時間を一定のオーダーに抑える事が可能である．

前出の JTRON 仕様 OS において，リアルタイム OS の上位階層の Java 実行環境の排他制御をミューテックスで実現するような状況では，オーバーヘッド増加の抑制に効果をもたらすといえる．

### 5.1.2 簡略化規則が問題となるケース

図 11 で示すように，簡略化規則の下ではロックした複数のミューテックスをアンロックするフェーズの間，優先度継承・上限プロトコルの定義に従えば段階的に下がるはずの現在優先度が，最高値に上昇したままの状態となる．本来であれば下がるべきタスク優先度が必要以上に高い状態であり，ブロックする必要の無いタスクの実行までもがブロックされてしまう．この現象によって生じ得る弊害を列挙する．

- タスクの実行順が変化する．
- ブロックされた高優先度タスクの応答時間はブロックしている低優先度のタスクの実行時間に依存する．
- クリティカルセクションに関した共有関係に無いタスクの実行もブロックされ得る．

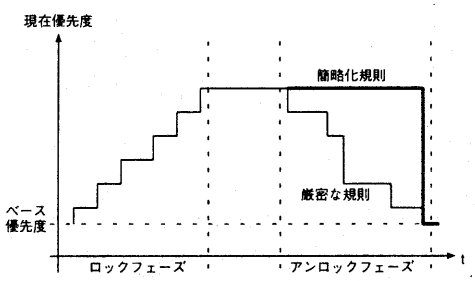


図 11: 優先度継承・上限プロトコル下でのクリティカルセクションにおけるタスクの現在優先度の変化

#### 5.1.3 簡略化規則が効果を発揮しないケース

ミューテックスのロックのネスト段数が一定の段数に限られる場合，CPU キャッシュの効果が現れやすい環境においては，両者の制御規則の性能差が縮小され得る．ネストが見られない場合には，両者の制御規則の間に本質的な性能差が見受けられない．また，簡略化規則は前節で述べたようなデメリットを併せ持ち，アプリケーションタスクの設計には，本来のプロトコルの定義以外で注意を必要とする．

以上の事から，ユーザーが直接システムコールを記述してミューテックス機能を利用する事が予想されるカーネルにおいては，Java 環境のように排他制御のネストが多段になる事が考えにくい事から，厳密な制御規則の採用が適切であると判断する．

### 5.2 サブセット化実装による性能向上の可能性

ITRON 仕様においては「弱い標準化」の考えに基づき，対象アプリケーションによっては機能をサブセット化して実装する事が許されている．性能向上に有効なサブセット化の手法を 2 つの側面から提案する．

#### 5.2.1 上限優先度違反エラーチェックの省略

ミューテックスをロックしているタスクに対するベース優先度の変更 (chg_pri サービスコール) のオーバーヘッドの増加は，ロックしているミューテックス数に比例する．よって，上限優先度違反のエラーチェックを省略した場合には，オーバーヘッドは一定とする事が可能である．ランタイムにおける上限優先度違反の検出が不可能となるが，本来であればアプリケーションタスクを設計する段階において考慮されるべきである．

#### 5.2.2 優先度継承・上限のいずれかのみのサポート

**優先度継承プロトコルに限定した場合**

優先度上限プロトコルの実現に必要な処理 (優先度制御の一部，上限優先度違反のエラーチェック)，ミューテックス管理 (初期化) ブロック内の上限優先度のフィールドが省略可能となる．

**優先度上限プロトコルに限定した場合**

優先度継承プロトコルの実現に必要な処理 (優先度制御の一部) が省略可能となる．また，タスクがロックしているミューテックスを管理する方法も簡素化が可能である．

## 6 おわりに

本研究では，TOPPERS/JSP カーネル上に 2 種類の制御規則のミューテックスを実装し，両者の制御規則の比較を中心とした性能評価を行った．この結果から商用カーネルの開発の際，実装するミューテックス機能の決定のための判断材料として有益な情報の提供が出来たと考える．また，上述の結果が ITRON 仕様の検証にフィードバックされ，次回の仕様改訂の際にはより完成度の高いミューテックスの仕様が規定される事が望まれる．

## 参考文献


[1] 坂村健 監修／高田広章 編:“$\mu$ ITRON4.0 仕様”，トロン協会 ITRON 部会 (1999)

[2] L.Sha, R.Rajkumar, and J.P.Leoczky : “Priority Inheritance Protocols : An Approach to Real-time Synchronization”, IEEE Trans. Computers, vol.39, pp.1774-1885, Sept. 1990.

[3] Hiroaki Takada, Ken Sakamura : “Experimental Implementations of Priority Inheritance Semaphore on ITRON-specification Kernel”,Proc. of 11th TRON Project Intl. Sumposium, pp.106-113, IEEE CS Press, Dec. 1994.

[4] 坂村健 監修／ JTRON 仕様 WG 編:“JTRON2.1 仕様”，トロン協会 ITRON 部会 (2000)

[5] 樋口正雄，高田広章，石川知雄：“$\mu$ ITRON4.0 仕様 OS におけるミューテックスの実装”，情報処理学会第 63 回全国大会講演論文集 (分冊 1 ) pp.61-62 (2001)