5M－3

C++言語用リアルタイムOSにおける優先度継承の実現

東原修†　中本幸一‡　門田浩‡
†NECマイコンテクノロジー　‡NECマイコンソフト開発環境研究所

1　はじめに

近年，オブジェクト指向プログラミングは広い範囲に渡って認知，普及してきており，それらの組み込みソフトウェアへの適用要求も今後増加すると考えられる．本稿では，オブジェクト指向およびC++言語の持つ優位性を持ち，優先度継承機構を実装したリアルタイムOSについて述べる．

2　C++システムコールAPIの提供

2.1　目的

本OSは，C++の言語仕様に基づく形式でシステムコールAPIを提供しており，OSの制御対象としてのリソースをクラスとして提供する事により，それらとプログラム，データとしてのリソースを統一的に扱う事を目的としている．

2.2　C++での実装上のメリット

本OSは，OSの制御対象のリソースのクラスとして，スレッド，セマフォ，割り込みハンドラ，例外ハンドラを提供している．スレッドをクラスとして提供する方法として，[1]における方法がある．それはスレッド作成用クラスを継承させたスレッドのユーザ定義クラスをスレッドとして直接扱わず，スレッド実行用のクラスを経由して扱う．しかし，この方法はプログラミング上，ユーザ定義のクラスをスレッドとして個別に扱う事ができない．
そこで，本OSでは以下の実装方法をとっている．

本OSにおける実装方法

OSの提供するスレッドクラスとユーザ定義クラスを継承したスレッドの実体としてのクラスをテンプレートを用いて作成し，スレッドとして直接扱う．プログラミング上，スレッドはユーザ定義毎の個別クラスとして扱う事も，OSの提供するスレッドクラスとして統一的に扱う事もできる．

2.2.1　スタック上でのリソース確保

クラスとしてOSの制御対象であるリソースを提供する事で，それらのスタック上での確保を可能とした．この場合，実行中のスレッドが確保されているスタックが解放される際の問題が考えられ，それは実行中のスレッドをdeleteオペレータによって削除する際に生じる問題と同様である．実行中のスレッドの削除を試みた場合のOSの振舞いについては，以下の2つが考えられる．

1. スレッドを強制終了させ削除する
2. スレッドが実行を完了するまで同期をとる

本OSでは，1の方法はスレッドの振舞いの予測を難しくするとの観点から2の方法を選択した．この方法は同時にスレッドが確保されているスタックが解放される際の問題を解決している．
例) スレッドA，Bがある．Aが自身のスタック上にBを確保した場合，Bの実行中にスタックが解放されようとすると，Bの実行が完了するまでAはペンディングされ，Bの実行完了後に削除，スタックの解放が行なわれる．
これによりUNIX等におけるfork，waitのようなJOIN操作をOSレベルで提供している事になる．

スタック上には割り込み，例外ハンドラクラスも確保でき，これらはスタック解放時に登録を解除されるため，スレッドやメソッドレベルでの固有の例外ハンドラの登録という処理を容易にしている．

"Implementing Priority Inheritance Mechanism in C++ Realtime OS ",
Osamu HIGASHIHARA, Yukikazu NAKAMOTO, Hiroshi MONDEN
†NEC Microcomputer Technology, Ltd.
‡NEC Corporation Microcomputer Software Engineering Laboratory

3 優先度の継承

OSの制御対象である共有データへのアクセスは排他されるべきであるが,排他がネストすると [2] における優先度の逆転現象が生じる場合がある. 優先度の逆転現象は, スレッドの実行時間等の予測の可能性を低くするという問題を持っている.
本OSは, 以下の問題を解決するため, 優先度継承機構を実現している.
・共有データの排他時に起こる優先度逆転
・スレッド削除時の同期によって起こる優先度逆転
優先度継承機構としては以下のものを検討した.

A: ロック中の共有データをロックしようとしたスレッドの優先度をロックしているスレッドへ継承させる ([3])

B: 共有データそのものへ優先度を付与してアクセスするスレッドへ優先度を継承させる

Aの機構は, 任意のスレッドが共有データをロック可能である. Bの機構は, Aに対してオーバーヘッドが小さくて済むが, 共有データの優先度より高い優先度を持つスレッドは, その共有データをロックできないケースが生じうる. 本OSでは, 上記Aの方法を実現し評価を行なった.
実装にあたり, 共有データのリリース時の処理として以下の2つが考えられる.

X: スレッドが複数の共有データをロックする場合, リリース時に優先度を計算する ([3] における Spec A)

Y: 上記aにおいて, リリース時の優先度の計算を簡略化する, 例えばリリース時には常にオリジナルの優先度へ復帰させる ([3] における Spec B) 等

Xの実装方法は, あるスレッドが複数の共有データをロックしている場合でも, 適正な優先度の継承が行なわれ, 優先度の逆転現象は起こりえない. Yの実装方法としての [3] における Spec B では, あるスレッドが複数の共有データをロックしている場合には, 優先度の逆転現象が起こりうる.
本OSでは, 上記Xの実装方法を採用している.

Xの方法においては, 優先度継承のため, ロック要求等によってペンディングしているスレッドおよび, 要求元の共有データをロックしているスレッドのネストを検索する. 更に本OSでは, この機能を利用して, ネストが自身に帰結してしまうために起こる, デッドロックの検出も可能とした.

4 性能評価

表1., 表2. に本OSの開発環境, 処理速度測定結果を記す.

表1. 開発環境

ターゲット	Forks 社製 FSB8745GX
CPU	A80486DX2−66/33MHz 内部 66MHz
コンパイラ	Watcom 社 Watcom C/C++ 32

表2. 処理速度測定結果

with thread dispatch	
lock	18 μ sec
lock (with a priority inheritance)	23 μ sec
lock (with two priority inheritance)	27 μ sec
unlock	23 μ sec
wait semaphore	15 μ sec
signal semaphore	20 μ sec

優先度の継承の際のオーバーヘッドは1継承あたり5 μ sec前後となっている. その結果, 本OSの実装した優先度継承機構はロックがネストした場合に処理が重くなることが予想されるため, 現状ではコンパクトなリアルタイムシステムにおいてはB.の機構がより実用的であると考える.

5 あとがき

本OSは, OSの制御対象をクラスオブジェクトとして実現し, プログラミングにおいて, それらを統一的に扱う事を目的とした. これは, スレッドのカプセル化, OSの制御対象のリソースのスタック上への確保等を可能とし, プログラミング時におけるプログラマーの負担を軽減できるものと考える.

参考文献

[1] DDC-I: "1st Object Exec User's Guide",1993.

[2] Sha,L. et al: "Priority inheritance protocol:An approach to realtime synchronization",IEEE Tran on Computers,Vol.39,No.9,1990.

[3] Takada,H., et al: "Experimental Implementations of Priority Inheritance semaphore on ITRON-specification Kernel", Proc of 11th TRON Project Symposium,1994.