4P–4

# レジスタウィンドウを用いた 高速マルチスレッドアーキテクチャの検討

高瀬 亮, 日高 康雄, 小池 汎平, 田中 英彦
{takase, hidaka, koike, tanaka}@mtl.t.u-tokyo.ac.jp
東京大学 工学部*

## 1 はじめに

レジスタウィンドウは, プロシジャコール時の引数渡しのオーバーヘッドを削減して, 高速なプロシジャコールを実現するために考えられた機構である. しかしながら, レジスタウィンドウ上でマルチスレッド処理を行なおうとすると, スレッドのスイッチなどを行なう際のオーバーヘッドが大きくなり, 性能が低下してしまう.

本稿では, プロシジャコールを高速に行なうというレジスタウィンドウが目指した本来の目的を損なわないと同時に, スレッドハンドリングも高速, かつ低いハードウェアコストで行なうことを目指した, レジスタウィンドウ管理のアーキテクチャについて検討する.

## 2 レジスタウィンドウを用いたマルチスレッディング

レジスタウィンドウとマルチスレッディングの両者を融合し, レジスタウィンドウ上でマルチスレッドを実装する手法には,

1. 全ウィンドウを一つのスレッドが使用
   スレッドのスイッチの際に全レジスタウィンドウの退避・復元を行なうため, スイッチのオーバーヘッドが大きい
2. 各スレッドに固定的にウィンドウを割り付け
   レジスタウィンドウ上へのスレッドスイッチが高速である反面, プロシジャコールの高速化というレジスタウィンドウ本来の目的を失っている
3. T-windows [1] など, ハードウェア指向の方法
   スレッドのハンドリングが柔軟で, オーバヘッドも少ない反面, レジスタをヒープのように確保するため, 複雑なハードウェアが必要となる

などがあるが, スレッドハンドリングまたはプロシジャコールの高速性と, ハードウェアコストという両方の点から考えると, 一長一短である.

そこで, 従来のレジスタウィンドウの形態はそのままにして, この上に複数のスレッドを共存させることによ

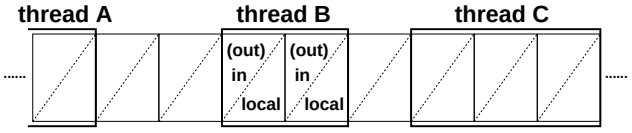


図 1: レジスタウィンドウ上のマルチスレッド

り, 効率の良いマルチスレッディングを実現することを目的とする (図 1).

## 3 管理アルゴリズム

### 3.1 スレッドハンドリング

マルチスレッディングでは, スレッドの生成, スレッドの終了, スレッドのスイッチ, という三つの処理に大別して考えることができる.

スレッドの生成では, スレッドをどのレジスタウィンドウに割り当てるかということが問題となる. 例えば, 既にレジスタウィンドウ上がスレッドで一杯である場合, 新たに生成するスレッドのために, スレッドを追い出す必要がある. このハンドリングは, FIFO, LRU などさまざまな手法が考えられるが, プロセッサ側からは, レジスタウィンドウの使用状況を表した管理レジスタにアクセスする手立てを用意しておくだけに留め, 実際のハンドリングはソフトウェアに任せる.

スレッドの終了時は, スレッドの管理情報から終了するスレッドの情報を削除する.

スレッドのスイッチは, 更にスイッチ先のスレッドがレジスタウィンドウ上にある場合と, スイッチ先がレジスタウィンドウから退避してしまっている場合に分類することができる.

前者は, レジスタウィンドウの管理レジスタから, 切り替え先のレジスタのレジスタウィンドウのトップにウィンドウを切り替えることで, 低コストで切り替えが行なえる. また, 後者の場合は, 新たにスレッドを生成することとほぼ同じ動作になる. 異なる点は, レジスタがメモリ上に退避してあるので, これを復元する必要があることだけである.

### 3.2 プロシジャのハンドリング

スレッドにおけるレジスタウィンドウのトップで, プロシジャコールが発生し, ウィンドウをずらすと, 他の

* *"A Study of Fast Multithreading Architecture using Register Windows"*
Ryo TAKASE, Yasuo HIDAKA, Hanpei KOIKE and Hidehiko TANAKA
Faculty of Engineering, the University of Tokyo
7–3–1 Hongo, Bunkyo-ku, Tokyo, 113, Japan

スレッドが使用しているウィンドウを破壊してしまうことが起こり得る.

この場合, 破壊されるウィンドウが属するスレッドを特定して, そのウィンドウをメモリ上に退避する動作が必要である. この動作は不可避であり, 退避にはオーバーヘッドがかかるが, スレッド生成の際に, ある程度の大きさのレジスタウィンドウを確保しておくことによって, 絶対的な退避回数を低減することが可能である.

この動作と同時に, ウィンドウが属するスレッドも変更しておく必要がある.

なお, プロシジャからリターンする際に, ウィンドウのボトムを検出した場合は, ウィンドウの移動は行なわない. 従って, スレッドがウィンドウのトップから破壊されることはない [2].

以上述べてきたように, レジスタウィンドウ上に複数のスレッドを実装することを考えると, ウィンドウ上のスレッドの干渉を, いかにうまく管理するかが鍵となる. この管理を行なうのが, プロセッサのレジスタウィンドウ管理レジスタであり, このアーキテクチャとその動作をうまく実装することに帰着できる.

## 4 ハードウェアアーキテクチャ

3 節で述べたようなレジスタウィンドウ管理を行なうために必要な, ハードウェアアーキテクチャについて, とくに管理レジスタに着目してみる.

まず, レジスタウィンドウ上に同時に複数のスレッドが存在することになるため, 隣接したスレッド間での干渉が起きないようにする必要がある. このため, レジスタウィンドウには WIM[1] (Window Invalid Mask) を設ける (図 2).

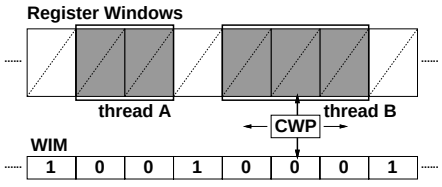


図 2: WIM と CWP

この WIM は, 各スレッドの境界を示しており, 両境界に挟まれたレジスタウィンドウが, そのスレッドが使用できるウィンドウになる. また, 常に現在のウィンドウ位置を示すポインタ CWP (Current Window Pointer) がある. この二つのレジスタにより, スレッドが, 使用可能なレジスタウィンドウを越えて使用してしまうことを避けることができる.

WIM は, レジスタウィンドウに存在する全スレッドに対して一つだけ存在するビットマップであるが, この管理レジスタだけだと, どのスレッドが別のスレッドによって破壊される (=退避させられる) ことになるのかが判断できない. 従って, これとは別にスレッド毎にウィンドウの使用範囲を示すテーブルを用意する (図 3).

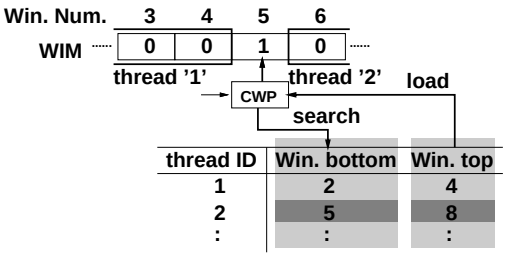


図 3: ウィンドウ使用範囲のテーブル

あるスレッドにおけるプロシジャのコールを考える. スレッドの予約済みウィンドウを越えて, 他のスレッドに干渉する事態になることがあるが, まず WIM によりスレッドの干渉の可能性がチェックされる. 干渉しない場合は問題ないが, 干渉する場合 (WIM の対応ビットが 1 の場合), テーブルのウィンドウボトム値をサーチする. ウィンドウが他のスレッドによって使用されていれば, 必ずこのテーブル内の 1 エントリにヒットし, 破壊されるスレッドがわかる.

また, ウィンドウトップの値は, スレッドスイッチの際に参照する. スレッドがスイッチする際, スイッチ先のウィンドウは必ずウィンドウトップであるので, この値を CWP にロードすることにより, スレッドをスイッチすることができる.

## 5 まとめ

レジスタウィンドウ上にマルチスレッドを実装する際の管理アルゴリズムについて述べた. また, このような管理を行なうのに適したハードウェアアーキテクチャについて述べた.

今後は, このようなアーキテクチャを, レジスタウィンドウを持った従来型プロセッサに付加することを考え, シミュレータを作成し, その評価をする予定である.

## 参考文献

[1] D. J. Quammen and D. R. Miller. Flexible register management for sequential programs. In *Proc. of 18th Annual Intl. Symp. on Comp. Arch.*, pp. 320–329, 1991.

[2] Y. Hidaka, H. Koike, and H. Tanaka. Multiple threads in cyclic register windows. In *Proc. of 20th Annual Intl. Symp. on Comp. Arch.*, pp. 131–142, May 1993.

[1] SPARC アーキテクチャにおける WIM とほぼ同義である.