

インテル® Pentium® 4 プロセッサおよび
インテル® Xeon™ プロセッサにおける
スピン・ループの使用

バージョン 2.1

2001 年 5 月

資料番号: 248674J-002

【輸出規制に関する告知と注意事項】

本資料に掲載されている製品のうち、外国為替および外国為替管理法に定める戦略物資等または役務に該当するものについては、輸出または再輸出する場合、同法に基づく日本政府の輸出許可が必要です。また、米国産品である当社製品は日本からの輸出または再輸出に際し、原則として米国政府の事前許可が必要です。

【資料内容に関する注意事項】

- ・本ドキュメントの内容を予告なしに変更することがあります。
 - ・インテルでは、この資料に掲載された内容について、市販製品に使用した場合の保証あるいは特別な目的に合うことの保証等は、いかなる場合についてもいたしかねます。また、このドキュメント内の誤りについても責任を負いかねる場合があります。
 - ・インテルでは、インテル製品の内部回路以外の使用にて責任を負いません。また、外部回路の特許についても関知いたしません。
 - ・本書の情報はインテル製品を使用できるようにする目的でのみ記載されています。
- インテルは、製品について「取引条件」で提示されている場合を除き、インテル製品の販売や使用に関して、いかなる特許または著作権の侵害をも含み、あらゆる責任を負わないものとします。
- ・いかなる形および方法によっても、インテルの文書による許可なく、この資料の一部またはすべてを複製することは禁じられています。

本資料の内容についてのお問い合わせは、下記までご連絡下さい。

インテル株式会社 資料センタ

〒305-8603 筑波学園郵便局 私書箱115号

Fax: 0120-47-8832

*一般にブランド名または商品名は各社の商標または登録商標です。

Copyright © Intel Corporation 1999, 2000

目次

1	はじめに	5
2	問題の説明と解決方法.....	5
2.1	スピン・ウェイト・ループに関する問題.....	5
2.1.1	PAUSE 命令使用のメカニズム	7
2.2	データの配置とキャッシュ・ライン・サイズ.....	7
3	問題の箇所の特定	9
4	まとめ	10
5	コード例	11
5.1	スピン・ウェイト・ロックの例	11
5.2	別のスピン・ウェイトの例	11

改訂履歴

改訂	改訂履歴	日付
2.1	コード名から製品名への改訂	2001年5月
2.0	インテル® Pentium® 4 プロセッサの推奨用語に関する改訂	2000年7月
1.0	初版	2000年2月

参考資料

このアプリケーション・ノートでは次の資料を参考にした。次の資料には、ここで取り上げた事項を理解するための有用な情報が含まれている。

インテル、『インテル® C/C++ コンパイラ・ユーザズ・ガイド』、資料番号 741901J、1999年

1 はじめに

マルチ・スレッドを使用する並列プログラムを正しく機能させるには、何らかの同期操作が必要である。通常の同期操作は、共用の同期変数と、それをチェックする「スピン・ウェイト」ループで行う。インテル® Pentium® 4 プロセッサとインテル® Xeon™ プロセッサ以降、インテル® IA-32 アーキテクチャは、スピン・ループに関するパフォーマンス問題に対処するための命令を新たに提供している。

このアプリケーション・ノートでは、高速プロセッサのマルチ・スレッド計算に関する重要な最適化問題、すなわちスピン・ループと共用データ管理について説明する。特に、スピン・ウェイト・ループには新しい PAUSE 命令を使用する方法と、共用データと非共用データを異なる 128 バイト・キャッシュ・ラインに振り分ける方法について説明する。この新しい PAUSE 命令は従来の IA-32 アーキテクチャすべてに対して上位互換性を持つので、できる限り早急に、スピン・ウェイト・ループで PAUSE 命令を使用するよう強くお勧めする。このアプリケーション・ノートでは、推奨する変更方法とその理由を説明する。

2 問題の説明と解決方法

マルチ・スレッド・システムを使用する場合は、同期に関するパフォーマンスを向上させるために 2 つの問題に取り組む必要がある。1 つは、共用同期変数をチェックするスピン・ウェイト・ループのプログラミング方法で、もう 1 つは同期変数のアライメント(メモリ上の実際の位置)である。実際には、アライメント問題は、プログラム内のすべての共用データのアライメントと、共用データと非共用データの共存方法を含めて一般化できる。これらの問題に対処すると、アプリケーション・パフォーマンスの大幅な低下と、不必要なシステム・リソース消費を避けられる。以下の各セクションに示すガイドラインに従って、パフォーマンスの向上とパワー消費量の削減を実現できる。

2.1 スピン・ウェイト・ループに関する問題

同期変数の値をチェックするスピン・ウェイト・ループは、一般に次のようになる。

```
wait_loop: cmp    eax, sync_var
           jne    wait_loop
```

同期変数はメモリ位置 `sync_var` にあり、ループを一度反復するごとにその値が読み込まれる。別のスレッドが目的の値をここに書き込むまで、このメモリ位置が繰り返しチェックされる。別のスレッドはこのメモリ・アドレス空間を共用しているが、その実行は独立している。

最近のマイクロプロセッサでは、このようなループが複数の同時読み込み要求を発生させ、それらの要求が順不同で実行される場合が多い。ロード実行中に別のプロセッサからの書き込みを検出すると、プロセッサはメモリ要求違反が起こらないのを保証しなければならない。そのため、プロセッサ大きなペナルティを受けることになる。言い換えると、上記のようなスピン・ウェイト・ループは、常にループを抜ける際に大きなペナルティを受ける。このペナルティは、インテル® Pentium® Pro プロセッサ、インテル® Pentium® II プロセッサ、インテル® Pentium III プロセッサでも発生するが、Pentium 4 プロセッサに比べると少ない。Pentium 4 プロセッサでは、このループを抜ける時のパフォーマンス・ペナルティが 25 倍も大きい。

Pentium 4 プロセッサ、Xeon プロセッサ、および今後のプロセッサでは、次のようにループに PAUSE 命令を挿入すれば、このペナルティを回避できる。

```
wait_loop: pause
           cmp    eax, sync_var
           jne    wait_loop
```

PAUSE 命令を入れると、ループにわずかな遅延が入り、メモリ・システム・バスのほぼ最高速度でメモリ要求が出されるようになる。その速さは、別のプロセッサが sync_var 値を変更できる最高速度とほぼ同じである。これ以上頻繁にメモリ要求を出す意味はない。PAUSE 命令は、スピン・ウェイトのパフォーマンスを大きく向上させるために使用する。PAUSE 命令を挿入すると、システム・リソースの使用量が少なくなり、スピン・ウェイト中のパワー消費量が大きく減少するという副次効果がある。

この方法は反対の効果を持つように感じるかもしれないので、もう一度強調する。PAUSE 命令によりスピン・ウェイト・ループが遅くなり、その結果、全般的パフォーマンスが向上する。スピン・ループ自体は高速に実行する必要がない。そのうちに発生するイベントをチェックするだけだからである。同期変数をあまりに頻繁にチェックすると、不必要にシステム・リソースを消費し、プロセッサがループを抜けるのにかなり大きなペナルティを受けることになる。

すでに同期が確立しているときに PAUSE 命令を実行するのを避けるには、スピン・ウェイト・ループを次のようにコーディングすればよい。

```
           cmp    eax, sync_var
           je     next_inst
wait_loop: pause
           cmp    eax, sync_var
           jne    wait_loop
next_inst: ...
```

このようなコードは、分岐予測に若干影響があるのに注意すること。最近のマイクロプロセッサは、分岐が発生するかどうか判定するより高速に命令を実行できる。そのため、ほとんどの最近のマイクロプロセッサは条件分岐命令(JE、JNE など)の結果がわかる前に、結果を予測して実行を進める。すなわち、予測が正しかったかがわかるまでは、必要かどうか不確かなまま実行していることになる。プロセッサの予測が誤っていた場合は、予測で実行した命令がすべてキャンセルされる。スピン・ウェイト・ループは別のスレッドが使用している共用データをプロテクトするケースが多いので、この共用データが別のスレッドの使用中に予測でアクセスされる可能性がある。ただし、予測によるアクセスで共用データの変更やその値に応じた処理の実行が、決して行われないのを強調する必要がある。

スピン・ウェイト・ループに関する問題を回避する方法は、他にもいくつかある。1つは、オペレーティングシステム(OS)の標準タイミグ・サービスを使用して、ロック変数を定期的にチェックする方法である。もう1つは、OS 自体でよく採用されている方法である。OS は、待ち状態にあるスレッドに強制的に HALT 命令を実行させ、リソースをいっさい消費しないようにできる。同期が確立したかもしれないと OS が判断したら、スレッドを「起こし」て処理を再開させる。どちらの方法でも、PAUSE 命令は必要ない。スレッドが長時間待つと予測できるときは、OS 呼び出しにより HALT 命令を実行する方がよい。

2.1.1 PAUSE 命令使用のメカニズム

PAUSE 命令は、Pentium 4 プロセッサで初めて導入された。この命令は厳密には、現在実行中のコードがスピン・ウェイト・ループであることをハードウェアに知らせるヒントである。このヒントを受けたときのハードウェアの動作は、プロセッサによって異なる。Pentium 4 プロセッサの場合、この命令は前述のように働き、その結果、全般的パフォーマンスが向上する。ただし、既存の IA-32 プロセッサでは、PAUSE 命令が NOP(no operation)と解釈されるので、プログラムにはいっさい影響しない。Pentium 4 プロセッサより前のすべてのインテル® アーキテクチャで、PAUSE 命令が NOP と解釈されるのが確認されている。また、インテル x86 ファミリ以外のプロセッサでも、テスト時に使用可能なものについては、NOP として動作することが確認された。

そこで、すべてのスピン・ウェイト・ループにすぐに PAUSE 命令を挿入するのを強くお勧めする。PAUSE 命令を使用すると、既存プラットフォーム上ではプログラムに影響はないが、Pentium 4 プロセッサ・プラットフォームではパフォーマンスが向上する。

コードに PAUSE 命令を挿入するには、次の 3 つの方法がある。

- セクション 5 の例に示すように、ニーモニックの PAUSE を使用する。
- インテル® C/C++ コンパイラで提供される `_mm_pause` イントリンシックを使用する。イントリンシックの詳細については、『インテル C/C++ コンパイラ・ユーザズ・ガイド』を参照のこと。
- PAUSE 命令を挿入する他の方法がなければ、この命令バイトを直接コーディングする。そのための便利なマクロを次に示す。

```
#define _MM_PAUSE    {__asm{__emit 0xf3};__asm {__emit 0x90}}
```

2.2 データの配置とキャッシュ・ライン・サイズ

インテル IA-32 アーキテクチャはコヒーレント・キャッシュを持つので、共用データへのアクセス方法は、全般的パフォーマンスに大きく影響する。この問題は複数スレッドから操作するデータに限られた問題だと思われるかもしれないが、そうではない。アクセスされた共用データはキャッシュに置かれるので、それと同じキャッシュ・ライン上のデータはすべて共用される。これは、2つのスレッドが偶然同じキャッシュ・ライン上にある2つの非共用データにアクセスするときに起こる「疑似共用」問題につながる可能性がある。システムの共用はキャッシュ・ライン単位で行われるので、このようなアクセスは実際に共用データ・アクセスに見える。このセクションでは、キャッシュ・ライン・サイズとデータ配置が、マルチ・スレッド・アプリケーションのパフォーマンスにどのように影響するかを説明する。

プロセッサが共用データを変更する際は、そのデータを含むキャッシュ・ラインに対して排他的「所有権」を取得する必要がある。プロセッサはそのために、このキャッシュ・ラインのコピーを持つ他のすべてのプロセッサが自分のキャッシュ・ラインを無効にするバス・トランザクションを発行する。これは、「所有権読み込み」トランザクション、または「無効」トランザクションである。他のプロセッサはこのどちらかのトランザクションを検出すると、このキャッシュ・ラインの自分のコピーを無効にする。その結果、これらのプロセッサがそのキャッシュ・ライン上のどれかのデータにアクセスするときは、システム・バスを使用してそのキャッシュ・ラインをもう一度読み込む必要がある。こうして、キャッシュ・ラインの最新コピー

を持つプロセッサがメモリを更新し、要求を出したプロセッサは更新済みデータのコピーを取得する。このようにして、どのプロセッサも常に最新データにアクセスできる。古いコピーが使用されるのはあり得ない。

複数プロセッサが共用データに書き込むときは、キャッシュ・ラインの「所有権」をプロセッサ間で受け渡す必要がある。このとき、プログラマが共用データの配置に気を付けないと、過度のバス・アクティビティが発生する可能性がある。このアプリケーション・ノートでは、2つのカテゴリの共用データに絞って説明する。1つは基幹的セクションを保護する同期変数で、もう1つはその基幹的セクションでアクセスするデータである。ただし、このセクションで取り上げる問題は一般にすべての共用データに共通である。

最も望ましいのは、同期変数が存在するキャッシュ・ラインは、同期変数が更新されたときだけ、システム・メモリ・バスを通して送信されるケースである。最悪なのは、多くの偽の送信が発生し、貴重なシステム・バス帯域幅を無駄に消費するケースである。他に方法があるのに、同じキャッシュ・ラインに複数の同期変数が存在すると、何らかの最悪のケースが発生する可能性がある。このセクションに示すいくつかの簡単な規則に従うだけで、最悪のケースを回避し、最良のパフォーマンスを得られる。

最も望ましいのは、各同期変数をそれぞれ別のキャッシュ・ラインに配置することである。そのキャッシュ・ラインに他のデータを置かなければ、さらに良い。その理由を明らかにするために、次のような、4つのプロセッサが基幹セクションを保護するロックを取り合うような典型的な4プロセッサ・システムを考えてみる。プロセッサAは、同期変数を使用して基幹セクションを保護する。プロセッサAが同期変数を変更するときは、まずシステム・バスを通してキャッシュ・ラインの所有権を取得し、プロセッサB、C、Dがそのキャッシュ・ラインの自分のコピーを無効にする。その結果、プロセッサB、C、Dはそのキャッシュ・ラインを再読み込みし、自分たちが基幹セクションにアクセスできないのを知って、スピン・ウェイト・ループを開始する。B、C、Dは同期変数のキャッシュ・コピーを持っており、プロセッサAが基幹セクションを解放するまではシステム・バス上にトランザクションは要求されない。プロセッサAは同期変数を変更すると、この同期変数が入っているキャッシュ・ラインをシステム・バスに書き込み、プロセッサB、C、Dがそのキャッシュ・ラインの自分のコピーを無効にする必要がある(2度目)。その結果、プロセッサB、C、Dはそのキャッシュ・ラインを再読み込みし、それぞれが書き込みを試みる。どれか1つのプロセッサが書き込み競争に「勝利」し、残りのプロセッサは再びそのキャッシュ・ラインを無効にして再読み込みする。書き込み競争に「負けた」プロセッサは、それぞれのスピン・ウェイト・ループに戻る。

そこで、基幹セクションのデータを同期変数と同じキャッシュ・ラインに置く影響を考えてみる。基幹セクション内では、プロセッサAは、データを変更するために同期変数への排他的アクセスを取得する。この間、プロセッサB、C、Dは常に同期変数を読み込むが、その同期変数は基幹セクションのデータと同じキャッシュ・ラインにあるので、プロセッサAはデータを変更した直後に排他的アクセスを失う。言い換えると、プロセッサAはデータへの排他的アクセスが必要なので、同期変数への排他的アクセスを取得する必要がある。プロセッサB、C、Dは同期変数への共用アクセスが必要なので、両方のデータを持つキャッシュ・ラインの状態は頻繁に変化する。この状態を「疑似共用」と呼び、システム・バスに多くの無駄なトランザクションが発生して、全体的なシーケンス・パフォーマンスに深刻な影響を与える。この状態は、同期変数のあるキャッシュ・ラインに置き、基幹セクションのデータを別のキャッシュ・ラインに置いて回避できる。そうすると、プロセッサAが2番目のキャッシュ・ラインの排他的アクセスを保持したまま、最初のキャッシュ・ラインをプロセッサ間で共用できる。プロセ

ッサ A が基幹セクションを解放するまで、どちらのキャッシュ・ラインの所有権も変化しない。

複数の同期変数を同じキャッシュ・ラインに置くと、無駄なバス・トランザクションの多発という同様のシナリオが発生する。複数の同期変数に共通なキャッシュ・ラインを一連のプロセッサが所有しようとするので、所有権が変わるたびに複数のバス・トランザクションが発生する。このような状況を避けるように十分注意する必要がある。

これらのシナリオは、常にキャッシュ・ラインとの関係を意識してすべての共用データを配置しなければならない理由を説明している。特に、頻繁にアクセスする同期変数は他の共用データと別のキャッシュ・ライン上に配置する必要がある。データの配置を制御するには、ターゲット・プラットフォーム・アーキテクチャのキャッシュ・ライン・サイズを知る必要がある。Pentium III プロセッサではキャッシュ・ライン・サイズが 32 バイトだが、Pentium 4 プロセッサでは 128 バイトである(これが 64 バイトの 2 つのセクタに分かれている)。したがって、Pentium 4 プロセッサならば 1 つの 128 バイト・キャッシュ・ラインに同期変数だけを置くのが最適だが、それが無理ならば次善の方法として 1 つの 64 バイト・キャッシュ・ライン上に同期変数だけを置くとよい。

1 つのキャッシュ・ライン・サイズを満たすようにパディングした同期データ構造を作成するだけでは不十分である。そのデータ構造をキャッシュ境界にアライメントする必要がある。アライメントしなければ、他の同期データ構造と同じキャッシュ・ラインに入ることはないが、1 つのキャッシュ・ライン上にそのデータだけを置くのは保証されない。アライメントを強制するには、2 つの方法がある。ダイナミック・メモリには、次のようなコードを使用する。

```
struct syn_str { int s_variable; };
void *p          = malloc ( sizeof (struct syn_str) + 127 );
syn_str * align_p = (syn_str *) ( ((int) p) + 127 & -128 );
```

インテル C/C++ コンパイラを使用する場合は、次のコードも使用できる。

```
_declspec(align(128)) struct syn_str aligned_structure;
```

3 問題の箇所の特定

次の作業は、アプリケーション・コードからスピン・ループと同期変数を見つけ出すことである。アプリケーションは標準ライブラリを使用して、同期を確立する場合が多い。その際、PAUSE の使用と同期変数のキャッシュ・ラインとアライメントが適切に行われているかチェックする必要がある。同期呼び出しがアプリケーション・コード内に散財しており、1 つずつ探し出さなければならないときがある。その場合は、問題の箇所の洗い出すのに VTune™ パフォーマンス・アナライザを使用するとよい。VTune アナライザで問題の箇所を発見したら、同期に関する問題があるのか、他の問題があるのか、その箇所を重点的に調べられる。

問題の箇所を見つけやすくするには、作業量が多く、多くのプロセッサを使用し、アプリケーションの基幹セクションへの負荷が大きいテスト・ケースを作成する。次に、VTune アナライザのセットアップ・ウィザードを使用して、タイムベース・サンプリング(TBS)をセットアップする。関数別のホット・スポット・レポートを調べる。アプリケーション実行時間の多くの部分を占める関数をすべてダブルクリックし、ソース・レベルで表示する(またはアセンブリ・コードで表示する)。インターロック交換命令(XCHG)があれば、同期コードの可能性が高い。

あるいは、特に理由がないのにキャッシュ・アクティビティが高ければ、同期変数同士が微妙に影響し合っている可能性がある。

他にも、コンテキスト切り替えが多発(Windows NT⁺のパフォーマンス・モニタで調べられる)しており、CPU 使用率が高く、プロセッサ数に応じたスケラビリティが得られなければ、同期コードの存在が疑われる。この場合は、スピン・ウェイト・ループが原因のことが多い。また、コンテキスト切り替えが多発しているのに、CPU 使用率が低い場合も注意が必要である。CPU 使用率が低いのは、ウェイト・ループに入っている HALT 命令が原因であることが多い。

4 まとめ

スレッド間で同期を確立するには、スピン・ウェイト・ループを使用するが多い。スピン・ウェイト・ループでは、パフォーマンスを最大にし、パワー消費を最低にするために、PAUSE 命令を使用する。既存のインテル・アーキテクチャでは PAUSE 命令が無視されるだけなので、すべてのアセンブリ・コードに PAUSE 命令を追加しても問題はない。また、1つのキャッシュ・ラインに同期変数だけを配置して、システム・バス・トラフィックを最小にする。Pentium 4 プロセッサでは、キャッシュ・ライン・サイズを 128 バイトにして対処するのが望ましいが、無理な場合は次善策でキャッシュ・ライン・サイズを 64 バイトにして対処する。

5 コード例

一般的なスピン・ウェイト・ループの例を次に示す。

5.1 スピン・ウェイト・ロックの例

```
get_lock:  mov    eax, 1
           xchg  eax, A           ; Try to get lock
           cmp   eax, 0           ; Test if successful
           jne   spin_loop

critical_section:
           <critical section code>
           mov   A, 0           ; Release lock
           jmp   continue

spin_loop: pause                   ; Short delay
           cmp   0, A           ; Check if lock is free
           jne   spin_loop
           jmp   get_lock

continue:
```

5.2 別のスピン・ウェイトの例

```
// Come here if we didn't get the lock on the first try.
for (;;)
{
    for (int i=0; i < SPIN_COUNT; i++)
    {
        if ( (i & SPIN_MASK) == 0
            && m_dwLock == UNLOCKED
            && InterlockedExchange( &m_dwLock, LOCKED ) == UNLOCKED)
            return;
#ifdef _X86_
        _mm_pause();
#endif
    }
    SleepForSleepCount( cSleeps++ );
}
```