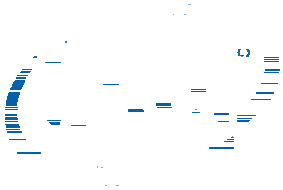

インテル® スレッドチェッカー 3.1 Linux* 版

入門ガイド

インテル® スレッドチェッカーは、競合状態、デッドロック、ストールなど、スレッド化問題を特定します。解析中にエラーが発生しない場合でも、潜在的なエラーを特定することができます。スレッドチェッカーを使用して、特定の診断結果をフィルターしたり、クリティカル・ソースの場所を識別したり、並列ソフトウェアをより強固にするためのヒントを得ることができます。

概要

本ガイドでは、マルチスレッド・コード例を用いて、インテル® スレッドチェッカーでスレッド化問題を特定し、対処する方法について示します。本ガイドは、スレッドチェッカーを使用してコードを解析、修正することを目的としています。

目次

著作権と商標について

改訂履歴

資料番号	リビジョン番号	説明	改訂日付
313445JA	001	初版	2006 年 5 月
313445 JA	002	マイナー変更	2006 年 9 月

1. サンプルコードのビルド

サンプルコード primes は、1 ～ 10,000 の範囲の素数を割り出し、総数を算出して記録します。POSIX* スレッド API を使用して、マルチスレッドで実行しますが、スレッドが同時に同じメモリー・ロケーションにアクセスするため、潜在的な競合状態を引き起こします。その結果、不正確な結果を生成することがあります。

サンプルコードをビルドするには

1. 次のコマンドを使用して、インテル® スレッドチェッカー環境を設定します。
2. `> source <path_to_tcheck_bin_directory>/tcheckvars.csh`
   または
   `> . <path_to_tcheck_bin_directory>/tcheckvars.sh`
3. 次のいずれかの方法で、primes 実行ファイルをビルドします。

   GNU* C/C++ コンパイラーでビルドするには

   a. primes ディレクトリーをコピーしたワークスペースに移動します。

   b. 次のコマンドを入力します。

      `> make`

      このコマンドは、デバッグ情報を有効にし最適化を無効にする `-g -o0` スイッチを使用して、`primes.gcc` と `primesFixed.gcc` という実行ファイルをビルドします。これらの設定によって、スレッドチェッカーはバイナリー・インストルメンテーション を実行できるようになり、実行ファイルをインストルメントして、POSIX スレッド API の呼び出しおよびメモリーアクセスを監視できます。

   インテル® C++ コンパイラー Linux 版でビルドするには

   a. シェルコマンドのプロンプトで、インテル® コンパイラー環境を設定します。

      次のように入力します。

      `> source <path_to_compiler_bin_directory>/iccvars.csh`
      または
      `> . <path_to_compiler_bin_directory>/iccvars.sh.`

   b. インテル® スレッドチェッカーがデフォルト (`/opt/intel/itt`) 以外の場所にインストールされている場合、変数 `ITT_BASE` が `tcheck` ディレクトリーを参照するように Makefile を編集してください。

c. primes ディレクトリーに移動して、次のコマンドを入力します。

```
> make icc
```

コンパイラーは、最も多くの情報を収集できるように、デバッグを有効にし、最適化を無効にする -g -o0 スイッチを使用して、.icc と .icc.tc という拡張子の実行ファイルをビルドします。拡張子が .icc.tc の実行ファイルでは、コンパイル時のソース・インストルメンテーションを可能にする -tcheck スイッチも使用されます。このスイッチにより、スレッドチェッカーは解析中により多くの情報を提供することができます。

メモ: 拡張子が .icc の実行ファイルは、-tcheck スイッチを使用してコンパイルされていないため、バイナリー・インストルメンテーションのみ利用できます。

注意: 別のウィンドウでコードを実行したり、新たにログイン後コードを実行する前に、再度 “source iccvars.*” コマンドを実行する必要があります。

primes 実行ファイルを何度か実行すると、次のような出力が表示されます。

```
> ./primes.gcc
Determining primes from 1 - 10000
Found 1228 primes

> ./primes.gcc
Determining primes from 1 - 10000
Found 1229 primes

> ./primes.gcc
Determining primes from 1 - 10000
Found 1229 primes

> ./primes.gcc
Determining primes from 1 - 10000
Found 1227 primes
```

この出力結果をよく見ると、同じプログラムであるにもかかわらず、実行結果が異なっていることがわかります。

1 ～ 10,000 の素数の正しい総数は 1,229 です。ここでは、スレッド化による不整合を比較的簡単に見つけることができましたが、大規模なプログラムでは、スレッド化による不整合を見つけることはより困難になります。

スレッドチェッカーは、すべての実行で発生しないものも含め、スレッド化による不整合を特定します。

2. データ収集

セクション 1 で行ったように、次のコマンドを使用して、インテル® スレッドチェッカーのコマンドライン・ツール環境を設定します。

```
> source <path_to_tcheck_bin_directory>/tcheckvars.csh
```

または

```
> . <path_to_tcheck_bin_directory>/tcheckvars.sh
```

次のコマンドを入力して、コマンドライン・ツールを起動します。

```
> tcheck_cl ./primes.gcc
```

次のような出力が表示されます。

```
Intel® Thread Checker 3.1 command line instrumentation driver
Copyright (c) 2006 Intel Corporation. All rights reserved.
     Building project
     Instrumenting
      25% primes.gcc       ( All Functions ):..
      75% libc-2.3.2.so    ( Minimal ):....
     100% libpthread-0.60.so ( Minimal ):..

     Running: <path>/primes/primes.gcc

     Determining primes from 1 - 10000
     Found 1229 primes

     Application finished
     ....
```

コマンドライン・ツールは、実行ファイル `primes.gcc` および関連モジュールのバイナリー・インストルメンテーションを実行します。インストルメンテーションの予測時間とインストルメントされるモジュールが表示されます。大規模なアプリケーションでは、処理に時間がかかります。インストルメント・レベルも表示されます。

`primes.gcc` ではデバッグ情報が有効なため、スレッドチェッカーは最も高いインストルメント・レベル **All Functions (すべての関数)** を使用します。

コンパイル済みライブラリーにはデバッグ情報がないため、スレッドチェッカーはライブラリー関数の呼び出しだけを記録するインストルメント・レベル **Minimal (最小)** を使用します。

インテル® スレッドチェッカーは、一時的にデータを格納するためのディレクトリーを作成します。デフォルトでは、ディレクトリーの名前は `/tmp/<login_name>_tc_cl_cache` です。次のように、短い形式または長い形式のコマンドライン・スイッチを使用して別のディレクトリーを指定することもできます。

```
> tcheck_cl -d /home/sample_data primes.gcc
```

または

```
> tcheck_cl --cache_dir /home/sample_data primes.gcc
```

3. 結果の解析およびコードの修正

実行後、インテル® スレッドチェッカーはインストルメンテーション・データを収集し解析して、利用可能なシンボル情報と関連付けます。診断結果は次のように表示されます。

ID	Short Description	Severity Name	Count	Context [Best]	Description	1st Access [Best]	2nd Access [Best]
1	Write -> Read data-race	Error	941	primes.c:27	Memory read at primes.c: 41 conflicts with a prior memory write at primes.c:42 (flow dependence)	primes.c:42	primes.c:41
2	Write -> Read data-race	Error	941	primes.c:27	Memory read at primes.c: 42 conflicts with a prior memory write at primes.c:42 (flow dependence)	primes.c:42	primes.c:42
3	Write -> Write data-race	Error	941	primes.c:27	Memory write at primes.c:42 conflicts with a prior memory write at primes.c:42 (output dependence)	primes.c:42	primes.c:42
4	Write -> Write data-race	Error	1	primes.c:27	Memory write at primes.c:41 conflicts with a prior memory write at primes.c:41 (output dependence)	primes.c:41	primes.c:41
5	Thread termination	Information	1	Whole Program 1	Thread termination at primes.c:61 - includes stack allocation of 10489856 and use of 2332 bytes	primes.c:61	primes.c:61

これで、診断結果を解析し、アプリケーションにおけるスレッド化による不整合を修正する準備が完了しました。

上の表で **ID 1** というのが最初の診断データです。ここでは、**Write -> Read data-race (書き込み -> 読み取りの競合状態) エラー**が発見されました。**Severity Name (重要度)** は、診断クラスを示します。ここでは、**Error (エラー)** となっています。**Count (カウント)** は、実行中にこのイベントが発生した回数を示します。実際のカウントは、スレッドのスケジューリングによって実行ごとに異なります。

Context[Best] (コンテキスト [最高]) は、診断のコンテキストを示します。**Best (最高)** では、スレッドチェッカーは最も多くの情報を表示します。デバッグ情報が有効なため、ソースファイル **primes.c** の **27 行目**の関数というように、コンテキストが表示されます。

Description (説明) は、より詳細な診断結果を示します。ここでは、**primes.c** の **42 行目**で 1 つのスレッドが変数に書き込み (**1st Access[Best] (1 回目のアクセス [最高])**) を行い、**41 行目**で別のスレッドが保護されていない同じ変数から読み取り (**2nd Access[Best] (2 回目のアクセス [最高])**) を行っているために、データ依存エラー (フロー依存) が発生しています。

ソース・インストルメント済みの `primes.icc.tc` を使用してスレッドチェッカーを実行すると、最初の診断結果が次のように表示されます。

ID	Short Description	Severity Name	Count	Context [Best]	Description	1st Access [Best]	2nd Access [Best]
1	Write -> Read data -race	Error	737	primes.c:27	Memory read of primeCount at primes.c:41 conflicts with a prior memory write of primeCount at primes.c:42 (flow dependence)	primes.c:42	primes.c:41

この診断結果には、前出の診断結果と異なる点が 2 つあります。まず、スケジューリングが異なるために、**Count (カウント)** 値が異なっています。また、**Description (説明)** では、問題となっているグローバル変数 **`primeCount`** についても表示されています。この追加情報は、ソース・インストルメンテーションを行った場合のみ表示されます。バイナリー・インストルメンテーションでは、グローバル・オブジェクト名は表示されません。

実際に、問題のソース行を見てみましょう。

```
38    while ( (number % factor) != 0 ) factor += 2;
39        if ( factor == number )
40        {
41            primes[ primeCount ] = number;
42            primeCount++;
43        }
```

グローバル変数 `primeCount` とグローバル配列 `primes[]` は保護されておらず、4 つの作業スレッドすべてによってアクセスされます。同期オブジェクトを追加して変数の使用をシリアル化することにより、予測不可能な同時変更から共有変数を保護することができます。

コードを修正し問題を排除するには

1. グローバル・ミューテックスを追加して、メインプログラムで初期化します。
2. 次のように、変数の読み取り/書き込みを行う前に、各スレッドにミューテックスを取得させます。
3. primes ディレクトリーをワークスペースにコピーします。デフォルトでは、このサンプルコードは `/opt/intel/itt/tcheck/samples/primes` にインストールされます。

```
pthread_mutex_t cs;
...

void * findPrimes ( void * arg )
{
    ....

    while ( (number % factor) != 0 ) factor += 2;
        if ( factor == number )
        {
            pthread_mutex_lock(&cs);
            primes[ primeCount ] = number;
            primeCount++;
            pthread_mutex_unlock(&cs);
        }
}
```

サンプルコード `primesFixed.c` には修正後のコードが含まれています。実行ファイル `primesFixed.gcc` は、すぐにインテル® スレッドチェッカーで解析することができます。インテル® C/C++ コンパイラー Linux 版でビルドされている場合、`primesFixed.icc` ではバイナリー・インストルメンテーションを使用した解析を行うことができます。また、`primesFixed.icc.tc` は、コンパイラーにより既にインストルメント済みです。

4. 次のステップ

インテル® スレッドチェッカーを最大限に活用するために、次のリソースも参考にしてください。

- **オンラインヘルプ**: オンラインヘルプには、スレッドチェッカーで利用可能なコマンドライン・オプションの一覧が含まれています。オンラインヘルプにアクセスするには、次のコマンドを実行してください。

  ```
  > tcheck_cl --help
  ```

- **診断**: `tcheck/doc` ディレクトリーにある `DiagnosticsGuide.pdf` では、診断の詳細、原因、および解決方法を提供しています。
- **サンプル**: 追加のコード例を利用することができます。本ガイドで取り上げていないほかのスレッド化問題を特定し、対処する方法について理解するのに役立ててください。サンプルは、`tcheck/samples` ディレクトリーにあります。
- **リリースノート**: `tcheck/doc` にある製品のリリースノート (`Release_Notes.txt`) には、製品のリリースノートには、製品の簡単な概要、動作環境、テクニカルサポート、および既知の制限事項に関する更新情報が含まれています。
- **インテル® スレッド・プロファイラー**: インテル® スレッドチェッカーでコードを解析した後に、**インテル® スレッド・プロファイラー**を使用してパフォーマンスを向上させることができます。
  スレッド・プロファイラーおよびその他のインテル® ソフトウェア開発製品に関する情報は、http://www.intel.co.jp/jp/software/products/ を参照してください。