Oracle® Solaris Studio 12.3: スレッドアナライザユーザーズガイド

Part No: E26458
2011 年 12 月

ORACLE

120220@25097

目次

はじめに

『スレッドアナライザユーザーズガイド』は、スレッドアナライザの紹介と2つの詳細なチュートリアルで構成されています。一つはデータの競合の検出を扱い、もう一方はデッドロックの検出を扱います。また、スレッドアナライザで認識されるAPIと役立つヒントが、付録に収録されています。

サポートされるプラットフォーム

このOracle Solaris Studioリリースは、Oracle Solarisオペレーティングシステムを実行するSPARCファミリーのプロセッサアーキテクチャーを使用するプラットフォームと、Oracle Solarisまたは特定のLinuxシステムを実行するx86ファミリーのプロセッサアーキテクチャーを使用するプラットフォームをサポートします。

このドキュメントでは、次の用語を使用してx86プラットフォームの違いを示しています。

- 「x86」は、64ビットおよび32ビットのx86互換製品を指します。
- 「x64」は、特定の64ビットx86互換CPUを指します。
- 「32ビットx86」は、x86ベースシステムで特定の32ビット情報を指します。

Linuxシステムに固有の情報は、サポートされているLinux x86プラットフォームだけに関連し、Oracle Solarisシステムに固有の情報は、SPARCおよびx86システムでサポートされているOracle Solarisプラットフォームだけに関連します。

サポートされているハードウェアプラットフォームおよびオペレーティングシステムリリースの完全なリストについては、Oracle Solaris Studio 12.3 リリースノートを参照してください。

Oracle Solaris Studioドキュメント

Oracle Solaris Studioソフトウェアの完全なドキュメントは次のようにして見つけることができます。

- 製品のドキュメントは、リリースノート、リファレンスマニュアル、ユーザーガイド、チュートリアルも含め、Oracle Solaris Studio Documentation Web サイトにあります。

- コードアナライザ、パフォーマンスアナライザ、スレッドアナライザ、dbxtool、DLight、および IDE のオンラインヘルプには、これらのツール内の「ヘルプ」メニューだけでなく、F1 キー、および多くのウィンドウやダイアログボックスにある「ヘルプ」ボタンを使用してアクセスできます。
- コマンド行ツールのマニュアルページでは、ツールのコマンドオプションが説明されています。

関連するサードパーティの Web サイトリファレンス

このドキュメントには、詳細な関連情報を提供するサードパーティの URL が記載されています。

注 – このドキュメントで紹介するサードパーティ Web サイトが使用可能かどうかについては、Oracle は責任を負いません。このようなサイトやリソース上、またはこれらを経由して利用できるコンテンツ、広告、製品、またはその他の資料についても、Oracle は保証しておらず、法的責任を負いません。また、このようなサイトやリソースから直接あるいは経由することで利用できるコンテンツ、商品、サービスの使用または依存が直接のあるいは関連する要因となり実際に発生した、あるいは発生するとされる損害や損失についても、Oracle は一切の法的責任を負いません。

開発者向けのリソース

Oracle Solaris Studio を使用する開発者のための次のリソースを見つけるには、Oracle Technical Network Web サイトにアクセスしてください。

- リソースは頻繁に更新されます。
- ソフトウェアの最近のリリースに関連する完全なドキュメントへのリンク
- サポートレベルに関する情報
- ユーザーディスカッションフォーラム

Oracle サポートへのアクセス

Oracle のお客様は、My Oracle Support にアクセスして電子サポートを受けることができます。詳細は、`http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` にアクセス、または、聴覚に障害がある方は、`http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` にアクセスしてください。

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P–1 表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	system% **su** password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
『』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	sun% **grep '^#define ** **XV_VERSION_STRING'**

Oracle Solaris OS に含まれるシェルで使用する、UNIX のデフォルトのシステムプロンプトとスーパーユーザープロンプトを次に示します。コマンド例に示されるデフォルトのシステムプロンプトは、Oracle Solaris のリリースによって異なります。

- C シェル

  machine_name% **command y|n** [*filename*]

- C シェルのスーパーユーザー

  machine_name# **command y|n** [*filename*]

- Bash シェル、Korn シェル、および Bourne シェル

  $ **command y|n** [*filename*]

- Bash シェル、Korn シェル、および Bourne シェルのスーパーユーザー

```
# command y|n [filename]
```

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: `Shift` キーを押します)。ただし、キーボードによっては `Enter` キーが `Return` キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、`Ctrl-D` は `Control` キーを押したまま `D` キーを押すことを意味します。

◆ ◆ ◆ 第 1 章

1 スレッドアナライザとその機能について

スレッドアナライザは、マルチスレッドプログラムの実行の分析に使用できる Oracle Solaris Studio ツールです。スレッドアナライザは、POSIX スレッド API、Solaris スレッド API、OpenMP 指令、またはこれらの組み合わせを使用して作成されたコード内でデータの競合やデッドロックなどのマルチスレッドプログラミングのエラーを検出できます。

この章では、次の内容について説明します。

1.1 スレッドアナライザの開始

スレッドアナライザはスレッド分析実験を検査するために設計された、パフォーマンスアナライザの特殊なビューです。パフォーマンスアナライザをこの特殊なビューで開始するには別のコマンド `tha` が使用され、この方法で開始されるツールはスレッドアナライザとして知られています。

スレッドアナライザは、このマニュアルで説明されているように、これらのタイプのデータを検査するために特別に生成できる実験でのデータの競合およびデッドロックを表示できます。

1.1.1 データの競合とは

スレッドアナライザは、マルチスレッドプロセスの実行中に生じたデータの競合を検出します。データの競合は、次のすべての条件に当てはまるときに生じます。

- 単一プロセス内の 2 つ以上のスレッドが、同じメモリー位置に同時にアクセスする
- 少なくとも 1 つが書き込みのためのアクセスである
- どのスレッドも、そのメモリーへのアクセスを制御するための排他的ロックを使用していない

この3つの条件が揃うとアクセス順序が定まらないため、実行するたびにその時の順序によって計算結果が異なる可能性があります。データの競合には、害のないもの (メモリーアクセスをビジーウェイトに使用するときなど) もありますが、データの競合の多くはプログラムのバグによるものです。

スレッドアナライザは、POSIX スレッド API、Solaris スレッド API、OpenMP、またはこれらの組み合わせを使用して作成されたマルチスレッドプログラムで動作します。

1.1.2 デッドロックとは

デッドロックとは、2 つ以上のスレッドが互いを待機しているために処理がまったく進まない状況を指します。デッドロックの原因は多数あります。スレッドアナライザは、相互排他ロックの不適切な使用によって生じたデッドロックを検出します。この種のデッドロックは、マルチスレッドアプリケーションでよく生じます。

2 つ以上のスレッドから成るプロセスは、次の条件がすべて揃うとデッドロックを生じることがあります。

- すでにロックを保持しているスレッドが新しいロックを要求する
- 新しいロックの要求が同時に行われる
- チェーン内の次のスレッドで保持されているロックを各スレッドが待機するという巡回チェーンを、2 つ以上のスレッドが形成する

デッドロック状況の簡単な例を次に示します。

- スレッド 1 はロック A を保持し、ロック B を要求する
- スレッド 2 はロック B を保持し、ロック A を要求する

デッドロックには潜在的デッドロックと実デッドロックの 2 種類があります。潜在的デッドロックは、所定の実行で必ず起きるわけではありませんが、スレッドのスケジュールや、スレッドによって要求されたロックのタイミングに依存したプログラムの実行で起きる可能性があります。実デッドロックは、プログラムの実行中に発生するものです。実デッドロックでは、関係するスレッドの実行は滞りますが、プロセス全体の実行は滞ることもあれば、そうでないこともあります。

1.2 スレッドアナライザ使用モデル

次の手順は、スレッドアナライザでマルチスレッドプログラムの問題を解決するプロセスを示しています。

1. データの競合の検出を行う場合、プログラムを計測します。
2. データの競合の検出またはデッドロック検出の実験を作成します。
3. 実験結果を検討し、スレッドアナライザで明らかになったマルチスレッドプログラミングの競合が、正当なバグまたは影響のない現象であるかどうかを判断します。
4. 本物のバグを修正し、入力データ、スレッド数、ループスケジュール、さらにはハードウェアなど、さまざまな要素を変更しつつ追加実験 (前述の手順 2) を作成します。これを繰り返すことで、決定論的ではない問題の突き止めに役立ちます。

前述の手順 1 から 3 については、以降の節で説明します。

1.2.1 データの競合を検出するための使用モデル

データの競合を検出するには、次の 3 つの手順を実行する必要があります。

1. データの競合の検出を有効にするコードを計測する
2. 計測したコードで実験を作成する
3. データの競合の実験結果を検討する

1.2.1.1 データの競合を検出するコードを計測する

アプリケーションでデータの競合の検出を可能にするには、実行時にメモリーアクセスを監視するコードをあらかじめ計測しておく必要があります。コードの計測方法としては、コンパイル中にアプリケーションのソースレベルで行う場合もあれば、バイナリに対し追加ツールを実行することによってアプリケーションのバイナリレベルで行う場合もあります。

ソースレベルの計測は、コンパイルに特別なオプションを指定して行います。また、使用する最適化レベルおよびその他のコンパイラオプションを指定できます。ソースレベルの計測は、コンパイラが一部の分析と計測を少ないメモリーアクセスで行うことができるため、実行時間がより短縮されます。

バイナリレベルの計測は、ソースコードが使用できない場合に役立ちます。ソースコードがある場合でもバイナリ計測を使用することがあります。ただしこの場合、アプリケーションが使用している共有ライブラリはコンパイルできません。`discover` ツールを使用したバイナリ計測では、バイナリだけでなく、開かれている共有ライブラリすべてを計測します。

ソースレベルの計測

ソースレベルで計測するには、特別なコンパイラオプションを付けてソースコードをコンパイルします。

`-xinstrument=datarace`

このコンパイラオプションを付けてコンパイラで生成されたコードが、データの競合の検出用に計測されます。

`-g` コンパイラオプションも、アプリケーションバイナリの構築時に使用する必要があります。このオプションを付けると、スレッドアナライザでデータの競合を報告するときにソースコードおよび行番号情報を表示するための追加データを生成できます。

バイナリレベルの計測

バイナリレベルで計測するには、`discover` ツールを使用する必要があります。バイナリが `a.out` という名前の場合、次のように実行することによって、計測済みのバイナリ `a.outi` を作成できます。

`discover -i datarace -o a.outi a.out`

`discover` ツールでは、開かれている共有ライブラリを、それがプログラム内で静的にリンクされているか、`dlopen()` によって動的に開かれているかにかかわらず、すべて自動的に計測します。デフォルトで、ライブラリの計測済みコピーは、ディレクトリ `$HOME/SUNW_Bit_Cache` に書き込まれます。

有効な `discover` コマンド行オプションの一部を次に示します。詳細は、`discover`(1) のマニュアルページを参照してください。

`-o` *file*　計測済みバイナリを、指定したファイル名で出力する

`-N` *lib*　指定したライブラリを計測しない

`-T`　どのライブラリも計測しない

`-D` *dir*　キャッシュディレクトリを *dir* に変更する

データの競合を検出するためにプログラムのバイナリコードを計測する場合、`discover` ツールでは、入力バイナリを次の条件を満たしてコンパイルする必要があります。

- オペレーティングシステムのバージョンが、少なくとも Oracle Solaris 10 Update 5 または Oracle Solaris 11 である。
- コンパイラは Oracle Solaris Studio 12 Update 1 以降のリリースのものである必要がある。
- コンパイラ最適化フラグ (`-xO1`、`-xO2`、`-xO3`、`-xO4`、`-xO5`) のいずれかを使用している。

- データの競合の報告時にスレッドアナライザがソースコードと行番号情報を表示できるように、`-g` コンパイラオプションも使用する必要がある。

また、バイナリがコンパイラオプション `-xbinopt=prepare` を付けてコンパイルされた場合は、SPARC ベースのシステムで実行中の、以前の Solaris バージョンでも `discover` ツールを使用できることがあります。このコンパイラオプションについては、`cc`(1)、`CC`(1)、または `f95`(1) のマニュアルページを参照してください。

1.2.1.2 計測済みアプリケーションで実験を作成する

データの競合の検出実験を作成するには、`-r race` フラグを付けて `collect` コマンドを使用して、アプリケーションを実行し、プロセスの実行中に実験データを収集します。`-r race` オプションを使用すると、競合を起こしたデータアクセスの対を収集データから知ることができます。

1.2.1.3 データの競合についての実験結果を検討する

データの競合を検出する実験を検討するには、`tha` コマンドを使用します。このコマンドにより、スレッドアナライザのグラフィカルユーザーインタフェースが起動します。`er_print` コマンド行インタフェースも使用できます。

1.2.2 デッドロックを検出するための使用モデル

デッドロックの検出には、次の 2 つの手順が必要です。

1. デッドロック検出実験の作成
2. デッドロック実験結果の検討

1.2.2.1 デッドロックを検出するための実験を作成する

デッドロック検出実験を作成するには、`-r deadlock` フラグを付けて `collect` コマンドを使用して、アプリケーションを実行し、プロセスの実行中に実験データを収集します。`-r deadlock` オプションを使用した場合、巡回チェーンを構成するロックの保持とロックの要求が収集データに含まれます。

1.2.2.2 デッドロックの実験結果を検討する

デッドロックを検出する実験を検討するには、`tha` コマンドを使用します。このコマンドにより、スレッドアナライザのグラフィカルユーザーインタフェースが起動します。`er_print` コマンド行インタフェースも使用できます。

1.2.3 データの競合およびデッドロックを検出するための使用モデル

データの競合とデッドロックを同時に検出するには、13 ページの「1.2.1 データの競合を検出するための使用モデル」で説明した 3 つの手順に従いデータの競合を検出し、`-r race,deadlock` フラグを付けた `collect` コマンドでアプリケーションを実行します。これで競合検出とデッドロック検出の両方のデータが実験結果に含まれます。

1.3 スレッドアナライザのインタフェース

スレッドアナライザは `tha` コマンドで起動することができます。

スレッドアナライザは、マルチスレッドプログラムの解析向けに設計されたパフォーマンスアナライザのインタフェースを採用しています。ただし、パフォーマンスアナライザの通常のタブの代わりに、「競合」、「デッドロック」、「デュアルソース」、「競合の詳細」、「デッドロックの詳細」というタブが表示されます。パフォーマンスアナライザを使用してマルチスレッドプログラムの実験結果を調べる場合、データの競合とデッドロックのためのタブとともに、「関数」、「逆アセンブリ」など従来からパフォーマンスアナライザにあるタブが表示されます。

◆ ◆ ◆ 第 2 章

2 データの競合チュートリアル

この章は、スレッドアナライザを使用してデータの競合を検出し修正する方法を学ぶ詳細なチュートリアルです。

このチュートリアルは、次の節から構成されています。

2.1 データの競合チュートリアルのソースファイル

このチュートリアルでは、データの競合を含んだ 2 つのプログラムを使用します。

- 最初のプログラムは素数を見つけます。このプログラムは C 言語で作成され、OpenMP 指令で並列化されています。ソースファイルは `prime_omp.c` と呼ばれます。
- 2 番目のプログラムも素数を見つけるもので、やはり C 言語で作成されます。ただし、これは OpenMP 指令でなく POSIX スレッドで並列化されます。ソースファイルは `prime_pthr.c` と呼ばれます。

2.1.1 データの競合チュートリアルのソースファイルの入手

このチュートリアルで使用するソースファイルは、Oracle Solaris Studio 開発者ポータルのサンプルのダウンロードエリア (`http://www.oracle.com/`

technetwork/server-storage/solarisstudio/downloads/solaris-studio-samples-1408618.html)からダウンロードできます。

サンプルファイルをダウンロードして展開したあと、SolarisStudioSampleApplications/ThreadAnalyzer ディレクトリからサンプルを見つけることができます。サンプルは、prime_omp および prime_pthr サブディレクトリにあります。各サンプルディレクトリには、手順に関する DEMO ファイルと Makefile ファイルが 1 つずつ含まれていますが、このチュートリアルではそれらの手順に従わず、Makefile も使用しません。代わりに、コマンドを個別に実行していきます。

このチュートリアルに従うには、サンプルディレクトリから prime_omp.c と prime_pthr.c ファイルを別のディレクトリにコピーするか、独自のファイルを作成し、次のコードリストからコードをコピーしてください。

2.1.2 prime_omp.c のソースコード

prime_omp.c のソースコードは次に示すとおりです。

```
 1  /*
 2   * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights Reserved.
 3   * @(#)prime_omp.c 1.3 (Oracle) 10/03/26
 4   */
 5
 6  #include <stdio.h>
 7  #include <math.h>
 8  #include <omp.h>
 9
10  #define THREADS 4
11  #define N 10000
12
13  int primes[N];
14  int pflag[N];
15
16  int is_prime(int v)
17  {
18   int i;
19   int bound = floor(sqrt(v)) + 1;
20
21   for (i = 2; i < bound; i++) {
22          /* no need to check against known composites */
23          if (!pflag[i])
24              continue;
25          if (v % i == 0) {
26              pflag[v] = 0;
27              return 0;
28          }
29      }
30      return (v > 1);
31  }
32
33  int main(int argn, char **argv)
```

```
34  {
35      int i;
36      int total = 0;
37
38  #ifdef _OPENMP
39      omp_set_dynamic(0);
40      omp_set_num_threads(THREADS);
41  #endif
42
43      for (i = 0; i < N; i++) {
44          pflag[i] = 1;
45      }
46
47      #pragma omp parallel for
48      for (i = 2; i < N; i++) {
49          if ( is_prime(i) ) {
50              primes[total] = i;
51              total++;
52          }
53      }
54
55      printf("Number of prime numbers between 2 and %d: %d\n",
56             N, total);
57
58      return 0;
59  }
```

2.1.3 prime_pthr.c のソースコード

prime_pthr.c のソースコードは次に示すとおりです。

```
 1  /*
 2   * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights Reserved.
 3   * @(#)prime_pthr.c 1.4 (Oracle) 10/03/26
 4   */
 5
 6  #include <stdio.h>
 7  #include <math.h>
 8  #include <pthread.h>
 9
10  #define THREADS 4
11  #define N 10000
12
13  int primes[N];
14  int pflag[N];
15  int total = 0;
16
17  int is_prime(int v)
18  {
19      int i;
20      int bound = floor(sqrt(v)) + 1;
21
22      for (i = 2; i < bound; i++) {
23          /* no need to check against known composites */
24          if (!pflag[i])
25              continue;
```

```
26          if (v % i == 0) {
27              pflag[v] = 0;
28              return 0;
29          }
30      }
31      return (v > 1);
32  }
33
34  void *work(void *arg)
35  {
36      int start;
37      int end;
38      int i;
39
40      start = (N/THREADS) * (*(int *)arg);
41      end = start + N/THREADS;
42      for (i = start; i < end; i++) {
43          if ( is_prime(i) ) {
44              primes[total] = i;
45              total++;
46          }
47      }
48      return NULL;
49  }
50
51  int main(int argn, char **argv)
52  {
53      int i;
54      pthread_t tids[THREADS-1];
55
56      for (i = 0; i < N; i++) {
57          pflag[i] = 1;
58      }
59
60      for (i = 0; i < THREADS-1; i++) {
61          pthread_create(&tids[i], NULL, work, (void *)&i);
62      }
63
64      i = THREADS-1;
65      work((void *)&i);
66
67      for (i = 0; i < THREADS-1; i++) {
68          pthread_join(tids[i], NULL);
69      }
70
71      printf("Number of prime numbers between 2 and %d: %d\n",
72             N, total);
73
74      return 0;
75  }
```

2.1.3.1 prime_omp.c および prime_pthr.c でのデータの競合の影響

コードに競合状態が含まれ、実行するごとに別々の計算結果が得られる場合、メモリーアクセスの順序は決まっていません。`prime_omp` および `prime_pthr` プログラムの正しい答えは 1229 です。

例をコンパイルして実行できるので、prime_omp または prime_pthr を実行するごとに、コード内のデータの競合によって誤ったまたは矛盾した結果が生じることがわかります。

次の例で、太字のコマンドを入力して、prime_omp プログラムをコンパイルし実行します。

```
% cc -xopenmp=noopt  -o prime_omp prime_omp.c -lm
%
% ./prime_omp
Number of prime numbers between 2 and 10000: 1229
% ./prime_omp
Number of prime numbers between 2 and 10000: 1228
% ./prime_omp
Number of prime numbers between 2 and 10000: 1180
```

次の例で、太字のコマンドを入力して、prime_pthr プログラムをコンパイルし実行します。

```
% cc -mt -o prime_pthr prime_pthr.c -lm
%
% ./prime_pthr
Number of prime numbers between 2 and 10000: 1140
% ./prime_pthr
Number of prime numbers between 2 and 10000: 1122
% ./prime_pthr
Number of prime numbers between 2 and 10000: 1141
```

各プログラムを 3 回実行した結果が矛盾していることに注意してください。矛盾した結果が表示されるまで、4 回以上プログラムを実行する必要がある場合もあります。

次に、データの競合が生じている位置を特定できるように、コードを計測し、実験を作成します。

2.2 スレッドアナライザを使用したデータの競合の検出方法

スレッドアナライザは、Oracle Solaris Studio パフォーマンスアナライザが使用するものと同じ "収集-分析" モデルに従います。

スレッドアナライザを使用するには、次の 3 つの手順を行います。

1. 22 ページの「2.2.1 コードを計測する」
2. 23 ページの「2.2.2 データの競合の検出実験を作成する」
3. 24 ページの「2.2.3 データの競合の検出実験を検証する」

2.2.1 コードを計測する

プログラムでデータの競合の検出を可能にするには、実行時にメモリーアクセスを監視するコードをあらかじめ計測しておく必要があります。この計測は、アプリケーションソースコードに行うことも、特定の Oracle コンパイラ最適化フラグでコンパイルされているアプリケーションバイナリに行うこともできます。このチュートリアルでは、プログラムを計測する両方のメソッドの使用方法を示します。

2.2.1.1 ソースコードを計測する

ソースコードを計測するには、特別なコンパイラオプション `-xinstrument=datarace` を使ってアプリケーションをコンパイルする必要があります。このオプションは、データの競合の検出用に生成したコードを計測するようにコンパイラに指示します。

`-xinstrument=datarace` コンパイラオプションを、プログラムのコンパイルに使用する既存のオプションセットに追加します。

注 – `-xinstrument=datarace` を使ってプログラムをコンパイルするときには、必ず `-g` オプションも指定してください。アナライザの全機能を有効にするための追加情報が生成されます。データの競合の検出用にプログラムをコンパイルするときには、高度な最適化を指定しないでください。`-xopenmp=noopt` を使って OpenMP プログラムをコンパイルしてください。高度な最適化を使用した場合、行番号や呼び出しスタックなど、報告された情報が間違っていることがあります。

このチュートリアル用にソースコードを計測するには、次のコマンドを使用できます。

```
% cc -xinstrument=datarace -g -xopenmp=noopt -o prime_omp_inst prime_omp.c -lm

% cc -xinstrument=datarace -g -o prime_pthr_inst prime_pthr.c -lm
```

例では、バイナリが計測済みバイナリであることがわかるように、出力ファイルの末尾に `_inst` と指定されていることに注意してください。これは必須ではありません。

2.2.1.2 バイナリコードを計測する

ソースコードの代わりにプログラムのバイナリコードを計測するには、`discover` ツールを使用する必要があり、このツールは、Oracle Solaris Studio に含まれ、`discover`(1) のマニュアルページと『Oracle Solaris Studio 12.3 Discover および Uncover ユーザーズガイド』で説明されています。

バイナリ計測の要件については、14 ページの「バイナリレベルの計測」を参照してください。

チュートリアルの例では、次のコマンドを入力して、最適化レベル 3 でコードをコンパイルし、discover で使用できるバイナリを作成します。

```
% cc -xopenmp=noopt -g -o prime_omp_opt prime_omp.c -lm

% cc -g -O3 -o prime_pthr_opt prime_pthr.c -lm
```

続いて、discover を、作成した prime_omp_opt および prime_pthr_opt 最適化済みバイナリで実行します。

```
% discover -i datarace -o prime_omp_disc prime_omp_opt

% discover -i datarace -o prime_pthr_disc prime_pthr_opt
```

これらのコマンドは計測済みバイナリ、prime_omp_disc および prime_pthr_disc を作成します。これらのバイナリを collect で使用して、スレッドアナライザで検証する実験を作成できます。

2.2.2 データの競合の検出実験を作成する

-r race フラグを付けて collect コマンドを使用してプログラムを実行し、プロセスの実行中にデータの競合の検出実験を作成します。OpenMP プログラムの場合、使用されるスレッド数が 1 より大きいことを確認してください。チュートリアルの例では 4 つのスレッドが使用されます。

ソースコードを計測して作成したバイナリから実験を作成するには、次のスレッドを使用します。

```
% collect -r race -o prime_omp_inst.er prime_omp_inst

% collect -r race -o prime_pthr_inst.er prime_pthr_inst
```

discover ツールを使用して作成したバイナリから実験を作成するには、次のスレッドを使用します。

```
% collect -r race -o prime_omp_disc.er prime_omp_disc

% collect -r race -o prime_pthr_disc.er prime_pthr_disc
```

データの競合を検出する可能性を高めるには、-r race フラグ付きで collect を使用して、複数のデータの競合の検出実験を作成することをお勧めします。実験ごとに異なるスレッド数と異なる入力データを使用してください。

たとえば prime_omp.c では、スレッド数は次の行で設定されます。

```
#define THREADS 4
```

この 4 を 1 より大きな他の整数 (たとえば 8) に変えると、スレッド数を変更できます。

prime_omp.c の次の行は、2 ~ 3000 の素数を検出するようにプログラムを制限します。

```
#define N 3000
```

N の値を変更して別の入力データを指定すると、プログラム作業量を増減できます。

2.2.3 データの競合の検出実験を検証する

スレッドアナライザ、パフォーマンスアナライザ、er_print ユーティリティーで、データの競合の検出実験を検証できます。スレッドアナライザおよびパフォーマンスアナライザはどちらも GUI インタフェースを表示します。スレッドアナライザはデフォルトの簡略セットのタブを表示しますが、それ以外はパフォーマンスアナライザと同じです。

2.2.3.1 スレッドアナライザを使用したデータの競合実験の表示

スレッドアナライザを開始するには、次のコマンドを入力します。

```
% tha
```

スレッドアナライザ GUI は、メニューバー、ツールバー、および各種表示用のタブを含む分割区画で構成されます。

左側の区画には、デフォルトで次の 3 つのタブが表示されます。

- 「競合」タブには、プログラムで検出されたデータの競合と、関連する呼び出しスタックトレースの一覧が表示されます。デフォルトでこのタブが選択されています。
- 「デュアルソース」タブには、選択したデータの競合の 2 つのアクセスに対応する 2 つのソースの位置が表示されます。データの競合アクセスが起きたソース行が強調表示されます。
- 「実験」タブには、実験でのロードオブジェクトが表示され、エラーおよび警告メッセージが一覧表示されます。

スレッドアナライザ画面の右側区画には、次の 2 つのタブが表示されます。

- 「概要」タブには、「競合」タブで選択したデータの競合アクセスに関する概要情報が表示されます。
- 「競合の詳細」タブには、「競合」タブで選択したデータの競合または呼び出しスタックトレースに関する詳細情報が表示されます。

2.2.3.2 er_print を使用したデータの競合実験の表示

er_print ユーティリティーは、コマンド行インタフェースを表示します。インタラクティブセッションで er_print ユーティリティーを使用して、セッション中にサブコマンドを指定します。コマンド行オプションを使用して、インタラクティブでない方法でもサブコマンドを指定できます。

次のサブコマンドは、er_print ユーティリティーで競合を調べるときに役立ちます。

- -races

  これは、実験で明らかになったデータの競合をすべて報告します。(er_print) プロンプトで races と指定するか、er_print コマンド行で -races と指定します。

- -rdetail *race_id*

  これにより、指定した *race_id* を持つデータの競合に関する詳細な情報が表示されます。(er_print) プロンプトで rdetail と指定するか、er_print コマンド行で -rdetail と指定します。指定した *race_id* が **all** の場合、すべてのデータの競合に関する詳細情報が表示されます。それ以外では、最初のデータの競合を表す **1** などの単一の競合番号を指定します。

- -header

  これは、実験に関する記述的情報を表示し、すべてのエラーまたは警告を報告します。(er_print) プロンプトで header と指定するか、コマンド行で -header と指定します。

詳細は、collect(1)、tha(1)、analyzer(1)、および er_print(1) のマニュアルページを参照してください。

2.3 実験結果について

この節では、er_print コマンド行とスレッドアナライザ GUI の両方を使用して、検出したデータの競合それぞれに関する次の情報を表示する方法について説明します。

- データの競合の一意の ID。
- データの競合に関連付けられた仮想アドレス、Vaddr。複数の仮想アドレスがある場合は、Multiple Addresses のラベルが括弧に囲まれて表示されます。
- 2 つの異なるスレッドによる仮想アドレス Vaddr へのメモリーアクセス。アクセスの種類 (読み取りまたは書き込み) のほか、関数、オフセット、およびアクセスが行われたソースコード内の行番号が表示されます。
- データの競合に関連付けられた呼び出しスタックトレースの総数。各トレースは、2 つのデータの競合アクセスが行われた時点で、スレッド呼び出しスタックの組を参照します。GUI を使用している場合、個々の呼び出しスタックトレース

を選択すると、2 つの呼び出しスタックが「競合の詳細」タブに表示されます。er_print ユーティリティーを使用している場合、rdetail コマンドによって 2 つの呼び出しスタックが表示されます。

2.3.1 prime_omp.c でのデータの競合

prime_omp.c でのデータの競合を調べるには、23 ページの「2.2.2 データの競合の検出実験を作成する」で作成したいずれかの実験を使用できます。

er_print で prime_omp_instr.er 実験のデータの競合情報を表示するには、次のコマンドを入力します。

```
% er_print prime_omp_inst.er
```

(er_print) プロンプトで **races** と入力すると、次のような出力が表示されます。

```
(er_print)races

Total Races:  2 Experiment:  prime_omp_inst.er

Race #1, Vaddr: 0x21ca8
      Access 1: Write, is_prime,
                       line 26 in "prime_omp.c"
      Access 2: Read,  is_prime,
                       line 23 in "prime_omp.c"
  Total Callstack Traces: 1

Race #2, Vaddr: (Multiple Addresses)
      Access 1: Write, main,
                       line 50 in "prime_omp.c"
      Access 2: Write, main,
                       line 50 in "prime_omp.c"
  Total Callstack Traces: 2
(er_print)
```

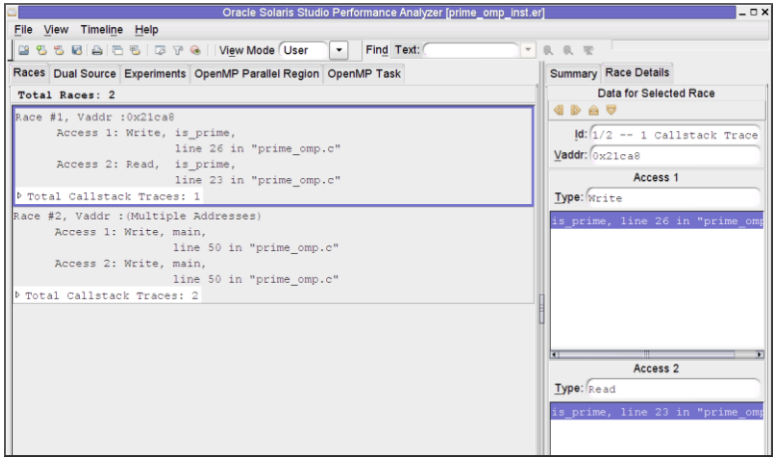
この特定のプログラム実行中に、2 つのデータの競合が生じました。

スレッドアナライザで prime_omp_inst.er 実験結果を開くには、次のコマンドを入力します。

```
% tha prime_omp_inst.er
```

次のスクリーンショットには、スレッドアナライザに表示された、prime_omp.c で検出された競合が示されています。

図 2–1　prime_omp.c で検出されたデータの競合



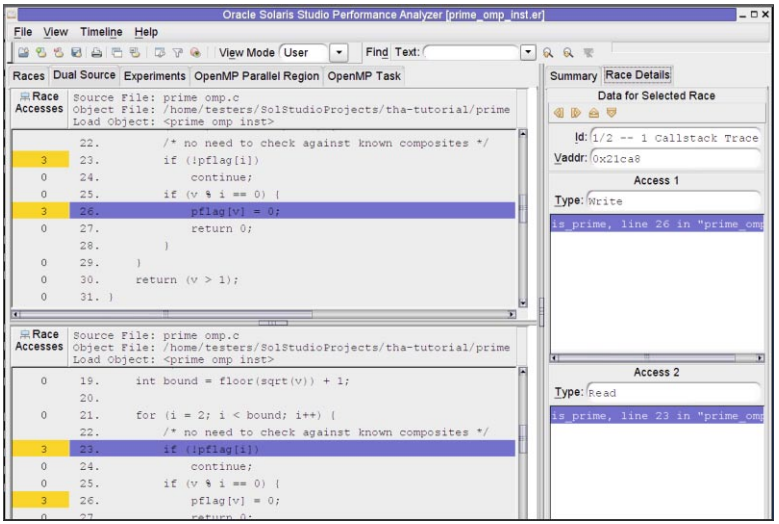
prime_omp.c には、次の 2 つのデータの競合が示されています。

- Race #1 は、行 26 での関数 is_prime における書き込みと、行 23 での同じ関数における読み取りとの競合を示しています。ソースコードを見ると、これらの行で、pflag[ ] 配列がアクセスされていることがわかります。スレッドアナライザで「デュアルソース」タブをクリックすると、両方の行番号でのソースコードとともに、コードの影響を受けた行での競合アクセス数を示すメトリックを簡単に確認できます。
- Race #2 は、main 関数の行 50 への 2 つの書き込み間の競合を示しています。「デュアルソース」タブをクリックすると、行 50 の primes [ ] 配列の値にアクセスする複数回の試行があることがわかります。

  Race #2 は、配列 primes[ ] の異なる要素で生じたデータの競合のグループを表します。これは、Multiple Addresses と指定された Vaddr で示されます。

スレッドアナライザの「デュアルソース」タブでは、データの競合に関連付けられた 2 つのソース位置を同時に確認できます。たとえば、「競合」タブで prime_pthr.c に Race #2 を選択し、続いて「デュアルソース」タブをクリックします。次のように表示されます。

図 2–2　prime_omp.c で検出されたデータの競合のソースコード



ヒント– 「デュアルソース」タブの左マージンに「競合アクセス」メトリックを表示するには、各ソースパネルのヘッダー上にマウスをドラッグする必要があります。

2.3.2 prime_pthr.c でのデータの競合

prime_pthr.c でのデータの競合を調べるには、23 ページの「2.2.2 データの競合の検出実験を作成する」で作成したいずれかの実験を使用できます。

er_print で prime_pthr_instr.er 実験のデータの競合情報を表示するには、次のコマンドを入力します。

```
% er_print prime_pthr_inst.er
```

(er_print) プロンプトで **races** と入力すると、次のような出力が表示されます。

```
(er_print) races

Total Races:  4 Experiment:  prime_pthr_inst.er

Race #1, Vaddr: (Multiple Addresses)
      Access 1: Write, is_prime + 0x00000270,
                       line 27 in "prime_pthr.c"
      Access 2: Write, is_prime + 0x00000270,
                       line 27 in "prime_pthr.c"
  Total Callstack Traces: 2

Race #2, Vaddr: 0xffbfe714
```

```
      Access 1: Write, main + 0x0000025C,
                       line 60 in "prime_pthr.c"
      Access 2: Read,  work + 0x00000070,
                       line 40 in "prime_pthr.c"
  Total Callstack Traces: 1

Race #3, Vaddr: (Multiple Addresses)
      Access 1: Write, work + 0x00000150,
                       line 44 in "prime_pthr.c"
      Access 2: Write, work + 0x00000150,
                       line 44 in "prime_pthr.c"
  Total Callstack Traces: 2

Race #4, Vaddr: 0x21a90
      Access 1: Write, work + 0x00000198,
                       line 45 in "prime_pthr.c"
      Access 2: Write, work + 0x00000198,
                       line 45 in "prime_pthr.c"
  Total Callstack Traces: 2
(er_print)
```

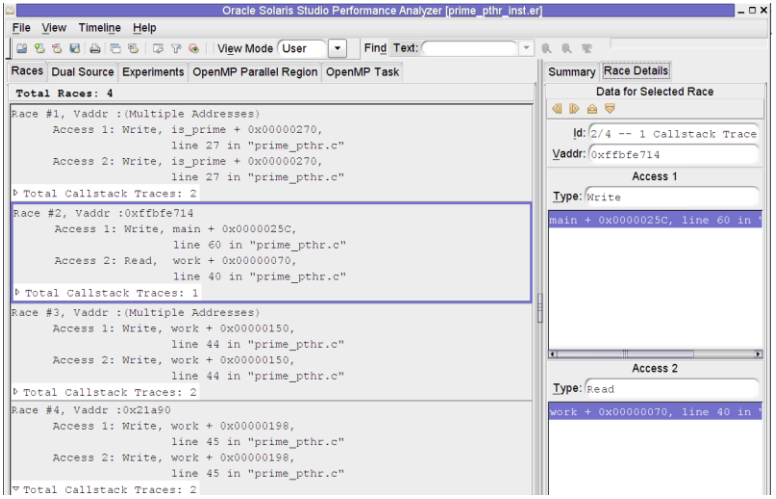
この特定のプログラム実行中に、4 つのデータの競合が生じました。

スレッドアナライザで `prime_pthr_inst.er` 実験結果を開くには、次のコマンドを入力します。

```
% tha prime_pthr_inst.er
```

次のスクリーンショットには、スレッドアナライザに表示された、`prime_pthr.c` で検出された競合が示されています。`er_print` で示された競合と同じであることに注意してください。

図 2–3　prime_pthr.c で検出されたデータの競合

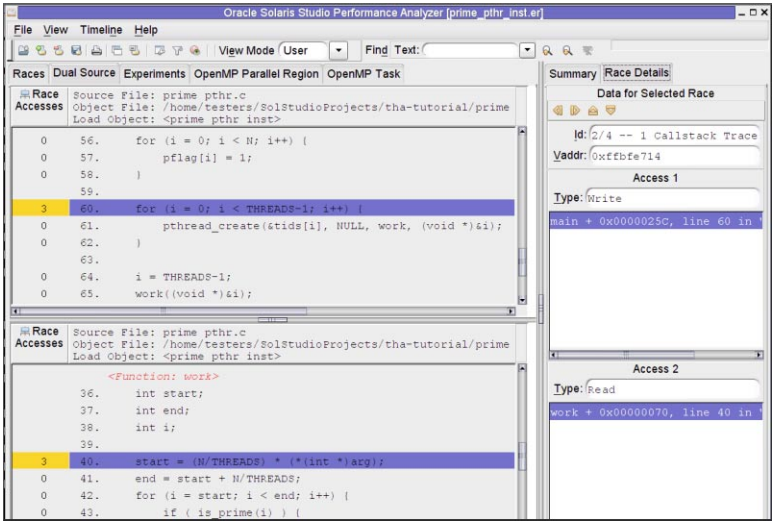


prime_pthr.c には、次の 4 つのデータの競合が示されています。

- Race #1 は、行 27 での関数 is_prime における pflag[ ] 配列への書き込みと、同じ行での pflag[ ] への別の書き込みとのデータの競合です。
- Race #2 は、行 60 での main() における i というメモリー位置への書き込みと、行 40 での (work() における *arg という) 同じメモリー位からの読み取りとのデータの競合です。
- Race #3 は、行 44 での primes[total] への書き込みと、同じ行での primes[total] への別の書き込みとのデータの競合です。
- Race #4 は、行 45 での total への書き込みと、同じ行での total への別の書き込みとのデータの競合です。

Race #2 を選択した後に「デュアルソース」タブをクリックした場合、次のスクリーンショットのように、2 つのソース位置が表示されます。

図 2–4　データの競合のソースコード詳細

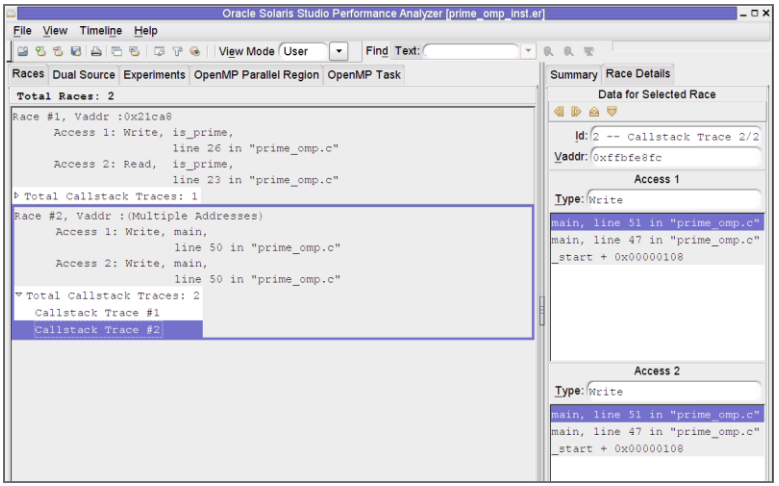


Race #2 の最初のアクセスは行 60 で行われ、上部のパネルに表示されます。2 番目のアクセスは行 40 で行われ、下部のパネルに表示されます。ソースコードの左側に「競合アクセス」メトリックが強調表示されます。このメトリックは、その行でデータの競合アクセスが報告された回数を示します。

2.3.3　データの競合の呼び出しスタックトレース

スレッドアナライザの「競合」タブで一覧表示されたデータの競合ごとに、1 つまたは複数の呼び出しスタックトレースが関連付けられています。呼び出しスタックは、データの競合を招く、コード内の実行パスを表示します。「呼び出しスタックトレース」をクリックすると、右側パネルの「競合の詳細」タブに、データの競合を招く関数呼び出しが表示されます。

図 2–5　prime_omp.c の呼び出しスタックトレースを示した「競合」タブ



2.4　データの競合の原因の診断

この節では、データの競合の原因を診断する基本的な方法について説明します。

2.4.1　データの競合が誤検知であるかどうかをチェックする

誤検知のデータの競合は、スレッドアナライザで報告されますが、実際には起こっていないデータの競合です。スレッドアナライザは、報告する誤検知の数を減らそうと試みます。ただし、ツールが正確なジョブを行えずに、誤検知のデータの競合を報告する場合があります。

誤検知のデータの競合は本当のデータの競合ではなく、したがってプログラムの動作に影響しないので、このデータの競合は無視できます。

誤検知のデータの競合の例については、36 ページの「2.5 誤検知」を参照してください。レポートから誤検知のデータの競合を削除する方法については、69 ページの「A.1 スレッドアナライザユーザー API」を参照してください。

2.4.2　データの競合が影響のないものであるかどうかを確認する

影響のないデータの競合は、存在していてもプログラムの正確さには影響しない意図的なデータの競合です。

一部のマルチスレッドアプリケーションでは、データの競合を引き起こすコードを意図的に使用します。設計によってデータの競合が存在するので、修正は必要ありません。ただし、場合によっては、このようなコードを正しく実行させるには非常に慎重を要します。これらのデータの競合については注意深く調べてください。

影響のない競合については、36 ページの「2.5 誤検知」を参照してください。

2.4.3 データの競合ではなくバグを修正する

スレッドアナライザは、プログラム内でデータの競合を見つけるときに役立ちますが、プログラム内のバグを自動的に見つけることも、見つかったデータの競合の修正方法を提示することもできません。データの競合は、バグによって生じることもあります。バグを見つけて修正することが重要です。単にデータの競合を取り除くだけでは正しいアプローチにはならず、以降のデバッグがさらに困難になる可能性があります。

2.4.3.1 prime_omp.c でのバグの修正

ここでは、prime_omp.c でのバグを修正する方法について説明します。完全なファイルのリストについては、「18 ページの「2.1.2 prime_omp.c のソースコード」」を参照してください。

配列 primes[ ] の要素でのデータの競合を削除するために、行 50 および 51 を critical セクションに移します。

```
47      #pragma omp parallel for
48      for (i = 2; i < N; i++) {
49          if ( is_prime(i) ) {
                #pragma omp critical

                {
50                 primes[total] = i;
51                 total++;
                }
52          }
53      }
```

また、次のように行 50 および 51 を 2 つの critical セクションに移すこともできますが、この変更ではプログラムを修正できません。

```
47      #pragma omp parallel for
48      for (i = 2; i < N; i++) {
49          if ( is_prime(i) ) {
                #pragma omp critical
                {
50                 primes[total] = i;
                }
                #pragma omp critical
                {
```

```
51                  total++;
                }
52          }
53      }
```

スレッドは、排他的ロックを使用して primes[ ] 配列へのアクセスを制御しているので、行 50 および 51 の critical セクションによってデータの競合が取り除かれます。ただし、プログラムはまだ正しくありません。2 つのスレッドは、同じ total 値を使用して primes[ ] の同じ要素を更新する可能性があり、primes[ ] の要素の中には、値がまったく割り当てられないものが生じる可能性があります。

行 23 での pflag[ ] からの読み取りと、行 26 での pflag[ ] への書き込みとの 2 番目のデータの競合は間違った結果を招かないので、実際には影響のない競合です。影響のないデータの競合の修正は必須ではありません。

2.4.3.2 prime_pthr.c でのバグの修正

ここでは、prime_pthr.c でのバグを修正する方法について説明します。完全なファイルのリストについては、19 ページの「2.1.3 prime_pthr.c のソースコード」を参照してください。

単一の相互排他ロックを使用して、行 44 での prime[ ] のデータの競合と、行 45 での total のデータの競合を取り除きます。

行 60 での i への書き込みと、行 40 での (*arg という) 同じメモリー位置からの読み取りとのデータの競合と、行 27 での pflag[ ] のデータの競合は、別々のスレッドによる変数 i への共有アクセスに問題があることを明らかにします。prime_pthr.c の初期スレッドは、行 60 ~ 62 でループの子スレッドを作成し、関数 work() で動作するようにこれらをディスパッチします。ループインデックス i は、アドレスで work() に渡されます。すべてのスレッドは i に対して同じメモリー位置にアクセスするので、各スレッドの i の値は一意のままではありませんが、初期スレッドがループインデックスを増やすたびに変化します。別々のスレッドが同じ i の値を使用するので、データの競合が起こります。問題を修正する 1 つの方法は、i をアドレスではなく値で work() に渡すことです。

次に、修正されたバージョンの prime_pthr.c を示します。

```
 1  /*
 2   * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights Reserved.
 3   * @(#)prime_pthr_fixed.c 1.3 (Oracle) 10/03/26
 4   */
 5
 6  #include <stdio.h>
 7  #include <math.h>
 8  #include <pthread.h>
 9
10  #define THREADS 4
11  #define N 10000
12
13  int primes[N];
```

```
14  int pflag[N];
15  int total = 0;
16  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
17
18  int is_prime(int v)
19  {
20      int i;
21      int bound = floor(sqrt(v)) + 1;
22
23      for (i = 2; i < bound; i++) {
24          /* no need to check against known composites */
25          if (!pflag[i])
26              continue;
27          if (v % i == 0) {
28              pflag[v] = 0;
29              return 0;
30          }
31      }
32      return (v > 1);
33  }
34
35  void *work(void *arg)
36  {
37      int start;
38      int end;
39      int i;
40
41      start = (N/THREADS) * ((int)arg) ;
42      end = start + N/THREADS;
43      for (i = start; i < end; i++) {
44          if ( is_prime(i) ) {
45              pthread_mutex_lock(&mutex);
46              primes[total] = i;
47              total++;
48              pthread_mutex_unlock(&mutex);
49          }
50      }
51      return NULL;
52  }
53
54  int main(int argn, char **argv)
55  {
56      int i;
57      pthread_t tids[THREADS-1];
58
59      for (i = 0; i < N; i++) {
60          pflag[i] = 1;
61      }
62
63      for (i = 0; i < THREADS-1; i++) {
64          pthread_create(&tids[i], NULL, work, (void *)i);
65      }
66
67      i = THREADS-1;
68      work((void *)i);
69
70      for (i = 0; i < THREADS-1; i++) {
71          pthread_join(tids[i], NULL);
72      }
```

```
73
74      printf("Number of prime numbers between 2 and %d: %d\n",
75             N, total);
76
77      return 0;
78  }
```

2.5 誤検知

スレッドアナライザは、実際にはプログラム内で生じていないデータの競合を報告する場合があります。これらは誤検知と呼ばれます。ほとんどの場合、誤検知は、ユーザー定義の同期によって、またはさまざまなスレッドでリサイクルされるメモリーによって引き起こされます。詳しくは、36 ページの「2.5.1 ユーザー定義の同期」および37 ページの「2.5.2 さまざまなスレッドでリサイクルされるメモリー」を参照してください。

2.5.1 ユーザー定義の同期

スレッドアナライザは、OpenMP、POSIX スレッド、および Solaris スレッドで提供されるほとんどの標準同期 API と構文を認識できます。ただし、ツールはユーザー定義の同期を認識できず、このような同期がコードに含まれる場合、誤検知のデータの競合を報告することがあります。

注 – このような誤検知のデータの競合を報告しないようにするために、スレッドアナライザには、ユーザー定義の同期が実行されたときにツールに通知するために使用できる一連の API が用意されています。詳しくは、69 ページの「A.1 スレッドアナライザユーザー API」 を参照してください。

なぜ API を使用する必要があるかを説明するため、次のように考えてみましょう。スレッドアナライザは、CAS 命令を使用したロックの実装、ビジー待機を使用した送信および待機操作などを認識できません。次に、プログラムが POSIX スレッド状態変数の一般的な使用法を採用した、誤検知のクラスの一般的な例を示します。

```
/* Initially ready_flag is 0 */

/* Thread 1: Producer */
100   data = ...
101   pthread_mutex_lock (&mutex);
102   ready_flag = 1;
103   pthread_cond_signal (&cond);
104   pthread_mutex_unlock (&mutex);
...
/* Thread 2: Consumer */
```

```
200   pthread_mutex_lock (&mutex);
201   while (!ready_flag) {
202       pthread_cond_wait (&cond, &mutex);
203   }
204   pthread_mutex_unlock (&mutex);
205   ... = data;
```

pthread_cond_wait() 呼び出しは、通常、プログラムエラーと疑わしいウェイクアップから保護するために述語をテストするループ内で行われます。述語のテストおよび設定は、多くの場合、相互排他ロックによって保護されます。前述のコードでは、スレッド 1 は行 100 で変数 data の値を生成し、行 102 で ready_flag の値を 1 に設定してデータが生成されていることを示します。続いて、 pthread_cond_signal() を呼び出して、消費者スレッドであるスレッド 2 を呼び起こします。スレッド 2 はループ内の述語 (*!ready_flag*) をテストします。フラグが設定されていることを検出すると、行 205 でデータを消費します。

行 102 での ready_flag の書き込みと行 201 での ready_flag の読み取りは、同じ相互排他ロックで保護されています。したがって、2 つのアクセス間にデータの競合はなく、ツールは正しく認識します。

行 100 での data の書き込みと、行 205 での data の読み取りは、相互排他ロックによって保護されません。ただし、プログラムロジックでは、フラグ変数 ready_flag のために行 205 での読み取りは常に、行 100 での書き込み後に行われます。この結果、データへのこれら 2 つのアクセス間にデータの競合は生じません。ただし、pthread_cond_wait() の呼び出し (行 202) が実際には実行時に呼び出されない場合、ツールは、2 つのアクセス間でデータの競合があると報告します。行 201 が実行される前に行 102 が実行された場合は、行 201 が実行されると、ループエントリテストは失敗し、行 202 はスキップされます。ツールは pthread_cond_signal() 呼び出しおよび pthread_cond_wait() 呼び出しを監視し、それらを組み合わせて同期を派生できます。行 202 で pthread_cond_wait() が呼び出されない場合、行 100 での書き込みが常に行 205 の読み取り前に実行されることがツールにはわかりません。したがって、これらが同時に実行されていると見なし、これらの間でデータの競合が生じていると報告します。

libtha(3C) のマニュアルページと69 ページの「A.1 スレッドアナライザユーザー API」 では、API を使用して、このような誤検知のデータの競合を報告しないようにする方法について説明しています。

2.5.2 さまざまなスレッドでリサイクルされるメモリー

一部のメモリー管理ルーチンは、あるスレッドが別のスレッドで使用できるように解放したメモリーをリサイクルします。スレッドアナライザは、別々のスレッドが使用する同じメモリー位置の寿命が重複していないことを認識できない場合があります。これが起きたときに、ツールは誤検知のデータの競合を報告することがあります。次の例は、この種の誤検知を示しています。

```
/*----------*/                    /*----------*/
/* Thread 1 */                    /* Thread 2 */
/*----------*/                    /*----------*/
 ptr1 = mymalloc(sizeof(data_t));
 ptr1->data = ...
 ...
 myfree(ptr1);

                                   ptr2 = mymalloc(sizeof(data_t));
                                   ptr2->data = ...
                                   ...
                                   myfree(ptr2);
```

スレッド 1 とスレッド 2 は同時に実行します。各スレッドは、プライベートメモリーに使用されるメモリーのチャンクを割り当てます。ルーチン `mymalloc()` は、`myfree()` の以前の呼び出しによって解放されたメモリーを提供できます。スレッド 2 は、スレッド 1 が `myfree()` を呼び出す前に、`mymalloc()` を呼び出します。この場合、`ptr1` と `ptr2` は別々の値を取り、2 つのスレッド間でデータの競合は生じません。ただし、スレッド 1 が `myfree ()` を呼び出した後にスレッド 2 が `mymalloc()` を呼び出した場合、`ptr1` と `ptr2` が同じ値を取ることがあります。スレッド 1 はこのメモリーにアクセスできなくなるので、データの競合は生じません。ただし、`mymalloc()` がメモリーをリサイクルしていることがわかっていない場合、ツールは、`ptr1` データの書き込みと `ptr2` データの書き込みとのデータの競合を報告します。この種の誤検知は、多くの場合、C++ アプリケーションで、C++ 実行時ライブラリがメモリーを一時変数用にリサイクルするときに起こりますまたしばしば、独自のメモリー管理ルーチンを実装したユーザーアプリケーションでも起こります。現在、スレッドアナライザは、標準の `malloc()`、`calloc()`、および `realloc()` インタフェースで実行されたメモリー割り当ておよび解放操作を認識できます。

2.6 影響のないデータの競合

マルチスレッドアプリケーションの中には、パフォーマンスを高めるためにデータの競合を意図的に許可する場合があります。影響のないデータの競合は、存在していてもプログラムの正確さには影響しない意図的なデータの競合です。次の例は、影響のないデータの競合を示します。

注 – 影響のないデータの競合以外でも、大きなクラスのアプリケーションでは、正しく設計するのが困難なロックフリーおよびウェイトフリーアルゴリズムを使用しているので、データの競合を許可します。スレッドアナライザは、これらのアプリケーションでのデータの競合の位置を特定する場合に役立ちます。

2.6.1 素数検索用のプログラム

`prime_omp.c` 内のスレッドは、関数 `is_prime()` を実行することによって、整数が素数かどうかをチェックします。

```
16  int is_prime(int v)
17  {
18      int i;
19      int bound = floor(sqrt(v)) + 1;
20
21      for (i = 2; i < bound; i++) {
22          /* no need to check against known composites */
23          if (!pflag[i])
24              continue;
25          if (v % i == 0) {
26              pflag[v] = 0;
27              return 0;
28          }
29      }
30      return (v > 1);
31  }
```

スレッドアナライザは、行 26 での pflag[ ] への書き込みと行 23 での pflag[ ] の読み取りとの間にデータの競合があることを報告します。ただし、このデータの競合は、最終的な結果の正確さには影響しないので影響のないものです。23 行で、スレッドは、所与の i の値で、pflag[i] が 0 に等しいかどうかをチェックします。pflag[i] が 0 に等しい場合、i が既知の合成数である (つまり、 i は素数でないとわかっている) ことを意味します。この結果、v が i で割り切れるかどうかをチェックする必要がなくなります。v がいずれかの素数で割り切れるかどうかだけをチェックすればよくなります。したがって、pflag[i] が 0 に等しい場合、スレッドは i の次の値に進みます。pflag[i] が 0 に等しくなく、v が i で割り切れる場合、スレッドは 0 を pflag[v] に割り当てて、v が素数でないことを示します。

正確さの観点からは、複数のスレッドが同じ pflag[ ] 要素をチェックし、同時にそれに書き込むかどうかは重要ではありません。pflag[ ] 要素の初期値は 1 です。スレッドはこの要素を更新するときに、0 の値を割り当てます。つまり、スレッドはその要素に対してメモリーの同じバイトの同じビットに 0 を格納します。現在のアーキテクチャーでは、このような格納は不可分であると想定することが安全です。つまり、その要素がスレッドによって読み取られるときに、読み取られる値は 1 か 0 のどちらかになります。スレッドは、0 の値を割り当てる前に所定の pflag[ ] 要素をチェックする場合 (行 23)、行 25 ~ 28 を実行します。その間に別のスレッドがその同じ pflag[ ] 要素に 0 を割り当てた場合も (行 26)、最終結果は変化しません。これは、基本的に、最初のスレッドが不必要に行 25 ~ 28 を実行したが、最終結果は同じであったことを意味します。

2.6.2 配列値の型を検証するプログラム

スレッドのグループが check_bad_array() を同時に呼び出して、配列 data_array の要素が「間違っている」かどうかをチェックします。各スレッドは配列の異なるセクションをチェックします。スレッドは、要素が間違っていることを検出した場合、グローバル共有変数 is_bad の値を true に設定します。

```
20  volatile int is_bad = 0;
 ...

100  /*
101   * Each thread checks its assigned portion of data_array, and sets
102   * the global flag is_bad to 1 once it finds a bad data element.
103   */
104  void check_bad_array(volatile data_t *data_array, unsigned int thread_id)
105  {
106     int i;
107     for (i=my_start(thread_id); i<my_end(thread_id); i++) {
108          if (is_bad)
109              return;
110          else {
111              if (is_bad_element(data_array[i])) {
112                  is_bad = 1;
113                  return;
114              }
115          }
116     }
117  }
```

行 108 での is_bad の読み取りと行 112 での is_bad への書き込みとの間にデータの競合があります。ただし、データの競合は最終結果の正確さに影響しません。

is_bad の初期値は 0 です。スレッドは is_bad を更新するときに、値 1 を割り当てます。つまり、スレッドは、is_bad に対してメモリーの同じバイトの同じビットに 1 を格納します。現在のアーキテクチャーでは、このような格納は不可分であると想定することが安全です。したがって、is_bad がスレッドで読み取られるときに、読み取られる値は 0 か 1 のどちらかになります。スレッドは、値 1 が割り当てられる前に is_bad をチェックする場合 (行 108)、for ループの実行を継続します。その間に別のスレッドが値 1 を is_bad に割り当てても (行 112)、最終結果は変化しません。スレッドが for ループを必要以上長時間実行したというだけのことです。

2.6.3 二重検査されたロックを使用したプログラム

シングルトンは、特定の種類のオブジェクトが、プログラム全体で 1 つしか存在しないようにします。二重検査されたロックは、マルチスレッドアプリケーションでシングルトンを初期化する一般的で効率的な方法です。次のコードは、この実装方法を示します。

```
100 class Singleton {
101     public:
102     static Singleton* instance();
103     ...
104     private:
105     static Singleton* ptr_instance;
106 };
107
108 Singleton* Singleton::ptr_instance = 0;
...
```

```
200 Singleton* Singleton::instance() {
201     Singleton *tmp;
202     if (ptr_instance == 0) {
203         Lock();
204         if (ptr_instance == 0) {
205             tmp = new Singleton;
206
207             /* Make sure that all writes used to construct new
208                Singleton have been completed. */
209             memory_barrier();
210
211             /* Update ptr_instance to point to new Singleton. */
212             ptr_instance = tmp;
213
214         }
215         Unlock();
216     }
217     return ptr_instance;
```

行 202 の ptr_instance の読み取りは、ロックによって意図的に保護されていません。このため、マルチスレッド環境で Singleton がすでにインスタンス化されているかどうかを判別するチェックが効率的になります。変数 ptr_instance の行 202 での読み取りと行 212 での書き込みとの間でデータの競合があるが、プログラムは正しく動作します。ただし、データの競合を許可する正しいプログラムを作成すると、余分な注意が必要になります。たとえば、上記の二重検査されたロックのコードで、行 209 での memory_barrier() の呼び出しは、Singleton を構築するためのすべての書き込みが完了するまで、ptr_instance がスレッドによって非 NULL として認識されないようにするために使用されます。

◆ ◆ ◆ 第 3 章

3 デッドロックのチュートリアル

このチュートリアルでは、スレッドアナライザを使用して、マルチスレッドプログラム内の潜在的デッドロックおよび実デッドロックを検出する方法について説明します。

チュートリアルでは次のトピックを扱います。

3.1 デッドロックについて

デッドロックという用語は、2 つ以上のスレッドが互いを待機しているために処理がどこへも進まない状況を意味します。デッドロックの原因は、間違ったプログラムロジックや (ロックやバリアーなどの) 同期の不適切な使用など数多くあります。このチュートリアルでは、相互排他ロックの不適切な使用によって生じたデッドロックに焦点を当てます。この種のデッドロックは、マルチスレッドアプリケーションでよく生じます。

2 つ以上のスレッドを含むプロセスが次の 3 つの条件に当てはまるときに、デッドロックが発生する可能性があります。

- すでにロックを保持しているスレッドが新しいロックを要求する
- 新しいロックの要求が同時に行われる
- チェーン内の次のスレッドで保持されているロックを各スレッドが待機するという巡回チェーンを、2 つ以上のスレッドが形成する

デッドロック状況の簡単な例を次に示します。

- スレッド 1 はロック A を保持し、ロック B を要求する
- スレッド 2 はロック B を保持し、ロック A を要求する

デッドロックには、潜在的デッドロックと実デッドロックの 2 つの種類があり、次のような違いがあります。

- 潜在的デッドロックは、所定の実行で必ず起きるわけではありませんが、スレッドのスケジュールや、スレッドによって要求されたロックのタイミングに依存したプログラムの実行で起きる可能性があります。
- 実デッドロックは、プログラムの実行中に発生するものです。実デッドロックでは、関係するスレッドの実行は滞りますが、プロセス全体の実行は滞ることもあれば、そうでないこともあります。

3.2 デッドロックチュートリアルのソースファイルの入手

このチュートリアルで使用するソースファイルは、Oracle Solaris Studio 開発者ポータルのサンプルのダウンロードエリア (http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/solaris-studio-samples-1408618.html)からダウンロードできます。

サンプルファイルをダウンロードして展開したあと、SolarisStudioSampleApplications/ThreadAnalyzer ディレクトリからサンプルを見つけることができます。サンプルは din_philo サブディレクトリにあります。din_philo ディレクトリには、手順に関する DEMO ファイルと Makefile ファイルが 1 つずつありますが、このチュートリアルではそれらの手順に従わず、Makefile も使用しません。代わりに、コマンドを個別に実行していきます。

このチュートリアルに従うには、SolarisStudioSampleApplications/ThreadAnalyzer/din_philo ディレクトリから din_philo.c ファイルを別のディレクトリにコピーするか、独自のファイルを作成し、次のコードリストからコードをコピーしてください。

食事する哲学者の問題をシミュレートする din_philo.c サンプルプログラムは、POSIX スレッドを使用する C プログラムです。このプログラムでは、潜在的デッドロックと実デッドロックの両方が示されます。

3.2.1 din_philo.c のソースコードリスト

din_philo.c のソースコードは次に示すとおりです。

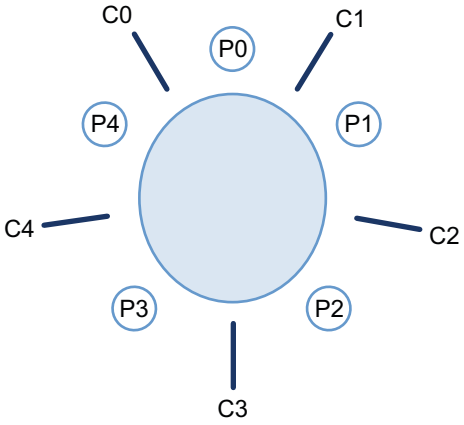
```
 1  /*
 2   * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights Reserved.
 3   * @(#)din_philo.c 1.4 (Oracle) 10/03/26
 4   */
 5
 6  #include <pthread.h>
 7  #include <stdio.h>
 8  #include <unistd.h>
 9  #include <stdlib.h>
10  #include <errno.h>
11  #include <assert.h>
12
13  #define PHILOS 5
14  #define DELAY 5000
15  #define FOOD 100
16
17  void *philosopher (void *id);
18  void grab_chopstick (int,
19                       int,
20                       char *);
21  void down_chopsticks (int,
22                        int);
23  int food_on_table ();
24
25  pthread_mutex_t chopstick[PHILOS];
26  pthread_t philo[PHILOS];
27  pthread_mutex_t food_lock;
28  int sleep_seconds = 0;
29
30
31  int
32  main (int argn,
33        char **argv)
34  {
35      int i;
36
37      if (argn == 2)
38          sleep_seconds = atoi (argv[1]);
39
40      pthread_mutex_init (&food_lock, NULL);
41      for (i = 0; i < PHILOS; i++)
42          pthread_mutex_init (&chopstick[i], NULL);
43      for (i = 0; i < PHILOS; i++)
44          pthread_create (&philo[i], NULL, philosopher, (void *)i);
45      for (i = 0; i < PHILOS; i++)
46          pthread_join (philo[i], NULL);
47      return 0;
48  }
49
50  void *
51  philosopher (void *num)
52  {
53      int id;
54      int i, left_chopstick, right_chopstick, f;
55
56      id = (int)num;
57      printf ("Philosopher %d is done thinking and now ready to eat.\n", id);
58      right_chopstick = id;
59      left_chopstick = id + 1;
```

```
60
61      /* Wrap around the chopsticks. */
62      if (left_chopstick == PHILOS)
63          left_chopstick = 0;
64
65      while (f = food_on_table ()) {
66
67          /* Thanks to philosophers #1 who would like to take a nap
68           * before picking up the chopsticks, the other philosophers
69           * may be able to eat their dishes and not deadlock.
70           */
71          if (id == 1)
72              sleep (sleep_seconds);
73
74          grab_chopstick (id, right_chopstick, "right ");
75          grab_chopstick (id, left_chopstick, "left");
76
77          printf ("Philosopher %d: eating.\n", id);
78          usleep (DELAY * (FOOD - f + 1));
79          down_chopsticks (left_chopstick, right_chopstick);
80      }
81
82      printf ("Philosopher %d is done eating.\n", id);
83      return (NULL);
84  }
85
86  int
87  food_on_table ()
88  {
89      static int food = FOOD;
90      int myfood;
91
92      pthread_mutex_lock (&food_lock);
93      if (food > 0) {
94          food--;
95      }
96      myfood = food;
97      pthread_mutex_unlock (&food_lock);
98      return myfood;
99  }
100
101 void
102 grab_chopstick (int phil,
103                 int c,
104                 char *hand)
105 {
106     pthread_mutex_lock (&chopstick[c]);
107     printf ("Philosopher %d: got %s chopstick %d\n", phil, hand, c);
108 }
109
110 void
111 down_chopsticks (int c1,
112                  int c2)
113 {
114     pthread_mutex_unlock (&chopstick[c1]);
115     pthread_mutex_unlock (&chopstick[c2]);
116 }
```

3.3 食事する哲学者の問題

食事する哲学者とは、昔からよく使われてきたシナリオで、仕組みは次のとおりです。0から4の番号が付けられた5人の哲学者が、考えながら円卓に座っています。やがて、個々の哲学者は空腹になり食事しようと考えます。テーブルには麺を乗せた大皿がありますが、各哲学者は使用できる箸を1本しか持っていません。食事するには、箸を共有する必要があります。各哲学者の(テーブルに向かって)右側の箸には、その哲学者と同じ番号が付けられています。

図3–1　食事する哲学者



P = Philosopher
C = Chopstick

各哲学者は最初に、自分の番号の付いた自身の箸に手を伸ばします。哲学者は、自身に割り当てられた箸を手にすると、隣の哲学者に割り当てられた箸に手を伸ばします。両方の箸を手にすると、食事できます。食事が終わると、箸をテーブルの元の位置に、左右に1本ずつ戻します。このプロセスは、麺がなくなるまで繰り返されます。

3.3.1 哲学者がデッドロックに陥るしくみ

デッドロックが実際に起こるのは、哲学者全員が自身の箸を手に持ち、隣の哲学者の箸が使用できるようになるのを待機している、次のような状況です。

- 哲学者0は箸0を手に持って箸1を待機している
- 哲学者1は箸1を手に持って箸2を待機している
- 哲学者2は箸2を手に持って箸3を待機している

- 哲学者 3 は箸 3 を手に持って箸 4 を待機している
- 哲学者 4 は箸 4 を手に持って箸 0 を待機している

この状況では誰も食事できず、哲学者たちはデッドロック状態に陥ります。プログラムを何度も実行するとわかります。つまりこのプログラムは、ハングアップすることもあれば、最後まで実行できることもあるのです。次の実行例は、このプログラムがハングアップする様子を示しています。

```
prompt% cc din_philo.c
prompt% a.out
Philosopher 0 is done thinking and now ready to eat.
Philosopher 2 is done thinking and now ready to eat.
Philosopher 2: got right  chopstick 2
Philosopher 2: got left chopstick 3
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 4 is done thinking and now ready to eat.
Philosopher 4: got right  chopstick 4
Philosopher 2: eating.
Philosopher 3 is done thinking and now ready to eat.
Philosopher 1 is done thinking and now ready to eat.
Philosopher 0: got right  chopstick 0
Philosopher 3: got right  chopstick 3
Philosopher 2: got right  chopstick 2
Philosopher 1: got right  chopstick 1
(hang)
```

Execution terminated by pressing CTRL-C

3.3.2 哲学者 1 の休眠時間の導入

デッドロックを回避する 1 つの方法は、哲学者 1 が自分の箸に手を伸ばす前に待機するというものです。コードの観点からは、自分の箸に手を伸ばす前に、指定した時間 (sleep_seconds)、哲学者 1 を休眠状態にすることができます。十分休眠した場合、プログラムは実デッドロックなしに終了できます。実行可能ファイルに対する引数として休眠する秒数を指定できます。引数を指定しない場合、哲学者は休眠しません。

次の擬似コードは各哲学者のロジックを示します。

```
   while (there is still food on the table)
      {
        if (sleep argument is specified and I am philosopher #1)
           {
             sleep specified amount of time
           }

        grab right fork
        grab left fork
        eat some food
```

```
        put down left fork
        put down right fork
      }
```

次のリストは、哲学者 1 が自分の箸に手を伸ばすまでに 30 秒間待機するようにしたプログラムを 1 回実行した様子を示しています。プログラムの実行は完了し、5 人の哲学者全員が食事し終わります。

```
% a.out 30
Philosopher 0 is done thinking and now ready to eat.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 4 is done thinking and now ready to eat.
Philosopher 4: got right  chopstick 4
Philosopher 3 is done thinking and now ready to eat.
Philosopher 3: got right  chopstick 3
Philosopher 0: eating.
Philosopher 2 is done thinking and now ready to eat.
Philosopher 2: got right  chopstick 2
Philosopher 1 is done thinking and now ready to eat.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
...
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
```

```
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 0 is done eating.
Philosopher 4: got left chopstick 0
Philosopher 4: eating.
Philosopher 4 is done eating.
Philosopher 3: got left chopstick 4
Philosopher 3: eating.
Philosopher 3 is done eating.
Philosopher 2: got left chopstick 3
Philosopher 2: eating.
Philosopher 2 is done eating.
Philosopher 1: got right  chopstick 1
Philosopher 1: got left chopstick 2
Philosopher 1: eating.
Philosopher 1 is done eating.
%
```

Execution terminated normally

プログラムを数回実行して異なる休眠引数を指定してみてください。哲学者 1 が箸を取る前に短時間しか待機しないとどうなるか、あるいは長い時間待機させたらどうなるかを観察するために、実行可能ファイル `a.out` にいろいろな休眠引数を指定してみます。そして、休眠引数を使用した状態と使用しない状態で複数回プログラムを再実行します。プログラムがハングアップする場合も、最後まで実行する場合もあります。プログラムがハングアップするかどうかは、スレッドのスケジュールと、スレッドによるロックの要求のタイミングによって異なります。

3.4 スレッドアナライザを使用したデッドロックの検索方法

スレッドアナライザを使用して、プログラム内の潜在的デッドロックおよび実デッドロックを確認できます。スレッドアナライザは、Oracle Solaris Studio パフォーマンスアナライザが使用するものと同じ収集-分析モデルに従います。

スレッドアナライザを使用するには、次の 3 つの手順を行います。

- ソースコードをコンパイルする。
- デッドロック検出実験を作成する。
- 実験結果を検する。

3.4.1 ソースコードをコンパイルする

コードをコンパイルし、必ず `-g` を指定します。高度な最適化では、行番号や呼び出しスタックなどの情報が間違って報告される場合があるので、高度な最適化は指定しないでください。`-g -xopenmp=noopt` を付けて OpenMP プログラムをコンパイルし、`-g -mt` だけを付けて POSIX スレッドプログラムをコンパイルします。

これらのオプションについては、`cc`(1)、`CC`(1)、または `f95` (1) のマニュアルページを参照してください。

このチュートリアルの場合、次のコマンドを使用してコードをコンパイルします。

```
% cc -g -o din_philo din_philo.c
```

3.4.2 デッドロック検出実験を作成する

`-r deadlock` オプションを付けてスレッドアナライザの `collect` コマンドを使用します。このオプションは、プログラムの実行中にデッドロック検出実験を作成します。

このチュートリアルの場合、次のコマンドを使用して、`din_philo.1.er` というデッドロック検出実験を作成します。

```
% collect -r deadlock -o din_philo.1.er din_philo
```

複数のデッドロック検出実験を作成することによって、デッドロックを検出する可能性を高められます。実験ごとに異なるスレッド数と異なる入力データを使用してください。たとえば、`din_philo.c` コードで、次の行の値を変更できます。

```
13  #define PHILOS 5
14  #define DELAY 5000
15  #define FOOD 100
```

続いて、前述のようにコンパイルして、別の実験結果を収集できます。

詳しくは、`collect`(1) および `collector`(1) のマニュアルページを参照してください。

3.4.3 デッドロック検出実験を検証する

スレッドアナライザ、パフォーマンスアナライザ、er_print ユーティリティーで、デッドロック検出実験を検証できます。スレッドアナライザおよびパフォーマンスアナライザはどちらも GUI インタフェースを表示します。スレッドアナライザはデフォルトの簡略セットのタブを表示しますが、それ以外はパフォーマンスアナライザと同じです。

3.4.3.1 スレッドアナライザを使用したデッドロック検出実験結果の表示

スレッドアナライザを開始して、din_philo.1.er 実験結果を開くには、次のコマンドを入力します。

```
% tha din_philo.1.er
```

スレッドアナライザは、メニューバー、ツールバー、および各種表示用のタブを含む分割区画で構成されます。

デッドロック検出用に収集された実験結果を開くと、デフォルトで、左側の区画に次のタブが表示されます。

- 「デッドロック」タブ

  このタブには、潜在的デッドロックと実デッドロックの一覧が示されます。デフォルトでこのタブが選択されています。各デッドロックに関わるスレッドが示されます。これらのスレッドは、各スレッドがロックを保持し、チェーン内の次のスレッドが保持している別のロックを要求するという巡回チェーンを形成しています。
- 「デュアルソース」タブ

  「デッドロック」タブで巡回チェーン内のスレッドを選択し、続いて「デュアルソース」タブをクリックします。「デュアルソース」タブには、スレッドがロックを保持したソース位置と、同じスレッドがロックを要求したソース位置が示されます。スレッドがロックを保持し要求したソース行が強調表示されます。
- 「実験」タブ

  このタブには、実験でのロードオブジェクトが表示され、エラーおよび警告メッセージが一覧表示されます。

スレッドアナライザ画面の右側区画に次のタブが表示されます。

- 「デッドロック」タブから選択したデッドロックの概要情報を示した「概要」タブ。
- 「デッドロック」タブから選択したスレッドコンテキストの詳細情報を示した「デッドロックの詳細」タブ。

3.4.3.2 er_print を使用した、デッドロック検出実験結果の表示

er_print ユーティリティーは、コマンド行インタフェースを表示します。インタラクティブセッションで er_print ユーティリティーを使用して、セッション中にサブコマンドを指定します。コマンド行オプションを使用して、インタラクティブでない方法でもサブコマンドを指定できます。

次のサブコマンドは、 er_print ユーティリティーでデッドロックを調べるときに役立ちます。

- -deadlocks

  このオプションは、実験で検出された潜在的デッドロックおよび実デッドロックについて報告します。(er_print) プロンプトで deadlocks を指定するか、er_print コマンド行で -deadlocks を指定します。

- -ddetail *deadlock_id*

  このオプションは、指定した *deadlock_id* を持つデッドロックの詳細な情報を返します。(er_print) プロンプトで ddetail を指定するか、er_print コマンド行で -ddetail を指定します。指定された *deadlock_id* が **all** の場合、すべてのデッドロックの詳細情報が表示されます。それ以外では、最初のデッドロックを表す **1** などの単一のデッドロック番号を指定します。

- -header

  このオプションは、実験に関する記述的情報を表示し、すべてのエラーまたは警告を報告します。(er_print) プロンプトで header と指定するか、コマンド行で -header と指定します。

詳細は、collect(1)、tha(1)、analyzer(1)、および er_print(1) のマニュアルページを参照してください。

3.5 デッドロックの実験結果について

この節では、スレッドアナライザを使用して、食事する哲学者のプログラムでのデッドロックを調べる方法について説明します。

3.5.1 デッドロックが発生した実行の検証

次のリストには、実デッドロックになった食事する哲学者のプログラムの実行が示されています。

```
% cc -g -o din_philo din_philo.c
% collect -r deadlock -o din_philo.1.er din_philo
Creating experiment database din_philo.1.er ...
Philosopher 1 is done thinking and now ready to eat.
Philosopher 2 is done thinking and now ready to eat.
```

```
Philosopher 3 is done thinking and now ready to eat.
Philosopher 0 is done thinking and now ready to eat.
Philosopher 1: got right  chopstick 1
Philosopher 3: got right  chopstick 3
Philosopher 0: got right  chopstick 0
Philosopher 1: got left chopstick 2
Philosopher 3: got left chopstick 4
Philosopher 4 is done thinking and now ready to eat.
Philosopher 1: eating.
Philosopher 3: eating.
Philosopher 3: got right  chopstick 3
Philosopher 4: got right  chopstick 4
Philosopher 2: got right  chopstick 2
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 1: got right  chopstick 1
Philosopher 4: got left chopstick 0
Philosopher 4: eating.
Philosopher 0: got right  chopstick 0
Philosopher 3: got left chopstick 4
Philosopher 3: eating.
Philosopher 4: got right  chopstick 4
Philosopher 2: got left chopstick 3
Philosopher 2: eating.
Philosopher 3: got right  chopstick 3
Philosopher 1: got left chopstick 2
Philosopher 1: eating.
Philosopher 2: got right  chopstick 2
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 1: got right  chopstick 1
Philosopher 4: got left chopstick 0
Philosopher 4: eating.
Philosopher 0: got right  chopstick 0
Philosopher 3: got left chopstick 4
Philosopher 3: eating.
...
Philosopher 4: got right  chopstick 4
Philosopher 2: got left chopstick 3
Philosopher 2: eating.
Philosopher 2: got right  chopstick 2
Philosopher 3: got right  chopstick 3
(hang)
```

Execution terminated by pressing CTRL-C

次のコマンドを入力して、`er_print` ユーティリティーで実験結果を検証します。

```
% er_print din_philo.1.er
(er_print) deadlocks
```

```
Deadlock #1, Potential deadlock
  Thread #2
    Lock being held:        0x215a0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215b8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #3
    Lock being held:        0x215b8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215d0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
```

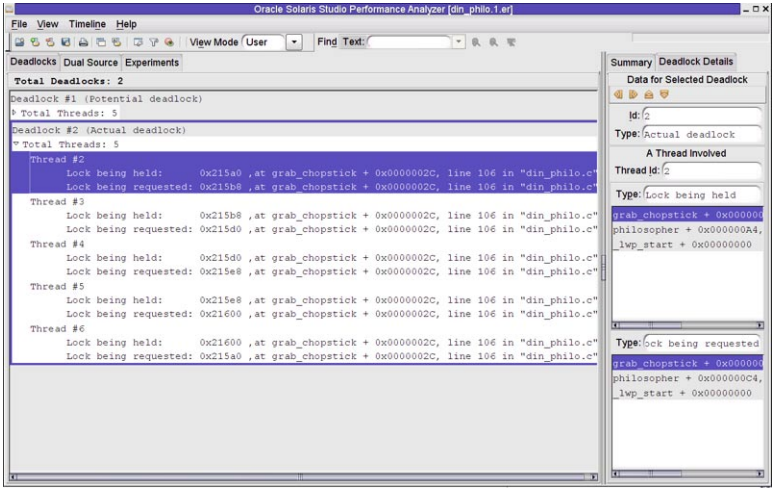
```
  Thread #4
    Lock being held:        0x215d0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215e8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #5
    Lock being held:        0x215e8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x21600, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #6
    Lock being held:        0x21600, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215a0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"

Deadlock #2, Actual deadlock
  Thread #2
    Lock being held:        0x215a0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215b8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #3
    Lock being held:        0x215b8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215d0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #4
    Lock being held:        0x215d0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215e8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #5
    Lock being held:        0x215e8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x21600, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #6
    Lock being held:        0x21600, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested:   0x215a0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"

Deadlocks List Summary: Experiment: din_philo.1.er Total Deadlocks: 2
(er_print)
```

次のスクリーンショットは、スレッドアナライザで表示されたデッドロック情報を示します。

図 3–2　din_philo.c で検出されたデッドロック



スレッドアナライザは、din_philo.c の 2 つのデッドロックを報告します (1 つは潜在的デッドロック、もう 1 つは実デッドロック)。より詳しく調べると、2 つのデッドロックは同一であるとわかります。

デッドロックに関する巡回チェーンは次のとおりです。

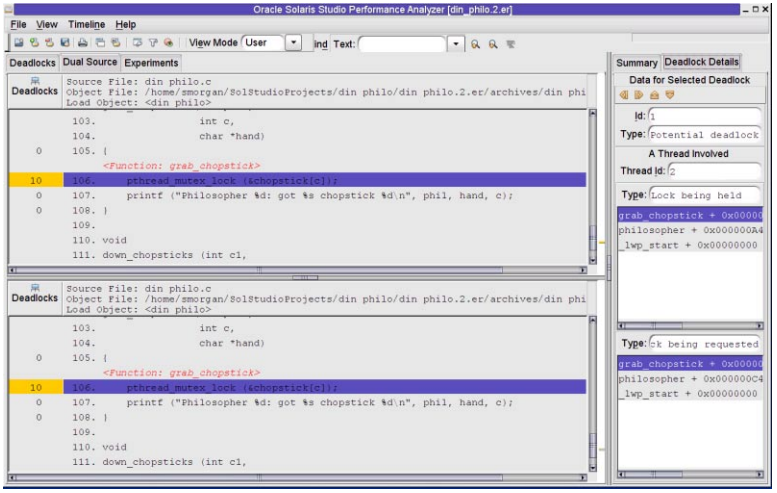
スレッド 2: アドレス 0x215a0 でロックを保持し、アドレス 0x215b8 でロックを要求
スレッド 3: アドレス 0x215b8 でロックを保持し、アドレス 0x215d0 でロックを要求
スレッド 4: アドレス 0x215d0 でロックを保持し、アドレス 0x215e8 でロックを要求
スレッド 5: アドレス 0x215e8 でロックを保持し、アドレス 0x21600 でロックを要求
スレッド 6: アドレス 0x21600 でロックを保持して、アドレス 0x215a0 でロックを要求

チェーン内の最初のスレッド (スレッド #2) を選択し、続いて「デュアルソース」タブをクリックすると、スレッド #2 がアドレス 0x215a0 でロックを取得したソースコード内の位置と、アドレス 0x215b8 でロックを要求したソースコード内の位置が表示されます。

次のスクリーンショットには、スレッド #2 の「デュアルソース」タブが示されています。スクリーンショットの上半分には、スレッド #2 が行 106 で pthread_mutex_lock() を呼び出すことによって、アドレス 0x215a0 でロックを取得したことが表示されています。スクリーンショットの下半分には、同じスレッドが行 106 で pthread_mutex_lock() を呼び出すことによって、アドレス 0x215b8 でロックを要求したことが表示されています。pthread_mutex_lock() への 2 つの呼び出しは、それぞれ別のロックを引数として使用しています。一般的に、ロック取得オペレーションとロック要求オペレーションは同じソース行に存在できません。

デフォルトのメトリック (排他的デッドロックメトリック) がスクリーンショットの各ソース行の左側に表示されます。このメトリックは、デッドロックに関与したロック取得またはロック要求オペレーションが、そのソース行で報告された回数を示します。デッドロックチェーンの一部となるソース行のみが、このメトリックについて 0 より大きい値を持ちます。

図 3–3　din_philo.c での潜在的デッドロック



3.5.2　潜在的デッドロックがあるにもかかわらず完了した実行の検証

十分に大きな休眠引数を指定した場合、食事する哲学者プログラムは、実デッドロックを回避でき、通常どおりに終了します。ただし、通常どおりに終了したからといって、プログラムにデッドロックがないことを意味するわけではありません。単に、保持されたロックと要求されたロックが、所与の実行中にデッドロックチェーンを形成しなかったことを意味するだけです。他の実行でタイミングが変更すれば、実デッドロックが生じる可能性があります。次のリストは、40 秒の休眠時間によって、通常どおりに終了する食事する哲学者プログラムの実行を示しています。ただし、`er_print` ユーティリティーとスレッドアナライザは潜在的デッドロックを報告します。

```
% cc -g -o din_philo_pt din_philo.c
% collect -r deadlock -o din_philo_pt.1.er din_philo_pt 40
Creating experiment database tha.2.er ...
Philosopher 0 is done thinking and now ready to eat.
Philosopher 2 is done thinking and now ready to eat.
Philosopher 1 is done thinking and now ready to eat.
Philosopher 3 is done thinking and now ready to eat.
Philosopher 2: got right  chopstick 2
Philosopher 3: got right  chopstick 3
Philosopher 0: got right  chopstick 0
Philosopher 4 is done thinking and now ready to eat.
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 3: got left chopstick 4
Philosopher 3: eating.
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
```

```
Philosopher 0: got right  chopstick 0
Philosopher 2: got left chopstick 3
Philosopher 2: eating.
...
Philosopher 4: got right  chopstick 4
Philosopher 3: got right  chopstick 3
Philosopher 2: got right  chopstick 2
Philosopher 4: got left chopstick 0
Philosopher 4: eating.
Philosopher 4 is done eating.
Philosopher 3: got left chopstick 4
Philosopher 3: eating.
Philosopher 0: got right  chopstick 0
Philosopher 0: got left chopstick 1
Philosopher 0: eating.
Philosopher 3 is done eating.
Philosopher 2: got left chopstick 3
Philosopher 2: eating.
Philosopher 0 is done eating.
Philosopher 2 is done eating.
Philosopher 1: got right  chopstick 1
Philosopher 1: got left chopstick 2
Philosopher 1: eating.
Philosopher 1 is done eating.
%
```

Execution terminated normally

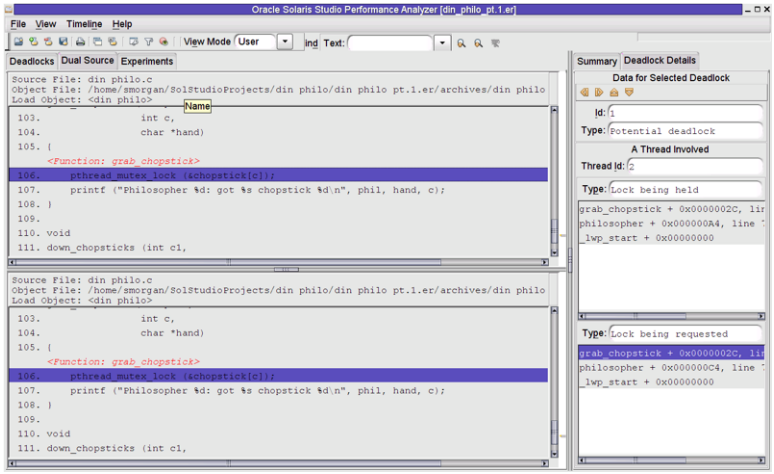
太字で示された次のコマンドを入力して、er_print ユーティリティーで実験結果を検証します。

```
% er_print din_philo_pt.1.er
(er_print) deadlocks
Deadlock #1, Potential deadlock
  Thread #2
    Lock being held:      0x215a0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested: 0x215b8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #3
    Lock being held:      0x215b8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested: 0x215d0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #4
    Lock being held:      0x215d0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested: 0x215e8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #5
    Lock being held:      0x215e8, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested: 0x21600, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
  Thread #6
    Lock being held:      0x21600, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"
    Lock being requested: 0x215a0, at: grab_chopstick + 0x0000002C, line 106 in "din_philo.c"

Deadlocks List Summary: Experiment: din_philo_pt.1.er Total Deadlocks: 1
(er_print)
```

次のスクリーンショットには、スレッドアナライザインタフェースでの潜在的デッドロック情報が示されています。

図 3–4　din_philo.c での潜在的デッドロック



3.6　デッドロックの修正と誤検知について

潜在的デッドロックと実デッドロックを取り除くには、哲学者は食事しようとする前にトークンを受け取る必要があるとするトークンのシステムを使用した方法があります。使用可能なトークンの数は、テーブルの哲学者の人数より少なくする必要があります。哲学者はトークンを受け取ると、テーブルのルールに従って食事できます。それぞれの哲学者は食事が終わればトークンを返し、プロセスを繰り返します。次の擬似コードは、トークンシステムを使用したときの、各哲学者のロジックを示します。

```
while (there is still food on the table)
   {
     get token
     grab right fork
     grab left fork
     eat some food
     put down left fork
     put down right fork
     return token
   }
```

以降の節では、トークンのシステムの 2 つの異なる実装について詳しく説明します。

3.6.1　トークンを使用した哲学者の規制

次のリストは、トークンシステムを使用する修正バージョンの食事する哲学者プログラムを示します。このソリューションには 4 つのトークン (食事する人数より 1 少

ない) が組み入れられ、したがって同時に 4 人の哲学者しか食事できません。このバージョンのプログラムは din_philo_fix1.c と呼ばれます。

ヒント – サンプルアプリケーションをダウンロードした場合、din_philo_fix1.c ファイルを SolarisStudioSampleApplications/ThreadAnalyzer/din_philo ディレクトリからコピーできます。

```
 1  /*
 2   * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights Reserved.
 3   * @(#)din_philo_fix1.c 1.3 (Oracle) 10/03/26
 4   */
 5
 6  #include <pthread.h>
 7  #include <stdio.h>
 8  #include <unistd.h>
 9  #include <stdlib.h>
10  #include <errno.h>
11  #include <assert.h>
12
13  #define PHILOS 5
14  #define DELAY 5000
15  #define FOOD 100
16
17  void *philosopher (void *id);
18  void grab_chopstick (int,
19                       int,
20                       char *);
21  void down_chopsticks (int,
22                        int);
23  int food_on_table ();
24  void get_token ();
25  void return_token ();
26
27  pthread_mutex_t chopstick[PHILOS];
28  pthread_t philo[PHILOS];
29  pthread_mutex_t food_lock;
30  pthread_mutex_t num_can_eat_lock;
31  int sleep_seconds = 0;
32  uint32_t num_can_eat = PHILOS - 1;
33
34
35  int
36  main (int argn,
37        char **argv)
38  {
39      int i;
40
41      pthread_mutex_init (&food_lock, NULL);
42      pthread_mutex_init (&num_can_eat_lock, NULL);
43      for (i = 0; i < PHILOS; i++)
44          pthread_mutex_init (&chopstick[i], NULL);
45      for (i = 0; i < PHILOS; i++)
46          pthread_create (&philo[i], NULL, philosopher, (void *)i);
47      for (i = 0; i < PHILOS; i++)
48          pthread_join (philo[i], NULL);
```

```
49      return 0;
50  }
51
52  void *
53  philosopher (void *num)
54  {
55      int id;
56      int i, left_chopstick, right_chopstick, f;
57
58      id = (int)num;
59      printf ("Philosopher %d is done thinking and now ready to eat.\n", id);
60      right_chopstick = id;
61      left_chopstick = id + 1;
62
63      /* Wrap around the chopsticks. */
64      if (left_chopstick == PHILOS)
65          left_chopstick = 0;
66
67      while (f = food_on_table ()) {
68          get_token ();
69
70          grab_chopstick (id, right_chopstick, "right ");
71          grab_chopstick (id, left_chopstick, "left");
72
73          printf ("Philosopher %d: eating.\n", id);
74          usleep (DELAY * (FOOD - f + 1));
75          down_chopsticks (left_chopstick, right_chopstick);
76
77          return_token ();
78      }
79
80      printf ("Philosopher %d is done eating.\n", id);
81      return (NULL);
82  }
83
84  int
85  food_on_table ()
86  {
87      static int food = FOOD;
88      int myfood;
89
90      pthread_mutex_lock (&food_lock);
91      if (food > 0) {
92          food--;
93      }
94      myfood = food;
95      pthread_mutex_unlock (&food_lock);
96      return myfood;
97  }
98
99  void
100 grab_chopstick (int phil,
101                 int c,
102                 char *hand)
103 {
104     pthread_mutex_lock (&chopstick[c]);
105     printf ("Philosopher %d: got %s chopstick %d\n", phil, hand, c);
106 }
107
```

```
108
109
110  void
111  down_chopsticks (int c1,
112                   int c2)
113  {
114      pthread_mutex_unlock (&chopstick[c1]);
115      pthread_mutex_unlock (&chopstick[c2]);
116  }
117
118
119  void
120  get_token ()
121  {
122      int successful = 0;
123
124      while (!successful) {
125          pthread_mutex_lock (&num_can_eat_lock);
126          if (num_can_eat > 0) {
127              num_can_eat--;
128              successful = 1;
129          }
130          else {
131              successful = 0;
132          }
133          pthread_mutex_unlock (&num_can_eat_lock);
134      }
135  }
136
137  void
138  return_token ()
139  {
140      pthread_mutex_lock (&num_can_eat_lock);
141      num_can_eat++;
142      pthread_mutex_unlock (&num_can_eat_lock);
143  }
```

この修正バージョンの食事する哲学者プログラムをコンパイルし、複数回それを実行してみます。トークンのシステムは、箸を使用して食事しようとする人の人数を制限し、これによって実デッドロックおよび潜在的デッドロックを回避します。

コンパイルするには、次のコマンドを使用します。

```
cc -g -o din_philo_fix1 din_philo_fix1.c
```

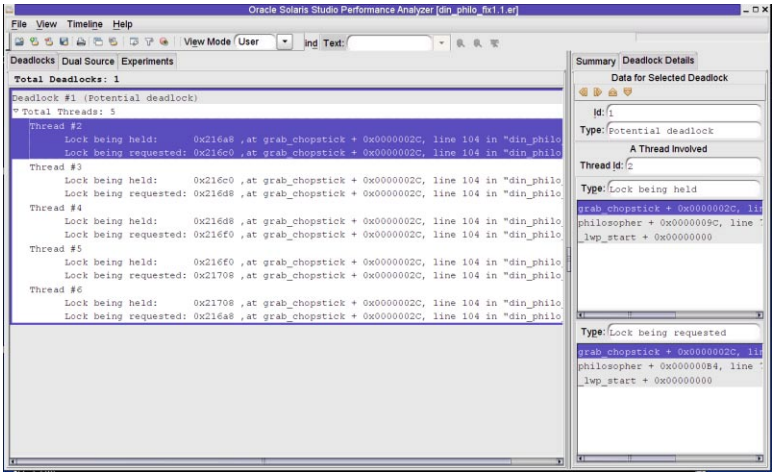
実験結果を収集する

```
collect -r deadlock din_philo_fix1 -o din_philo_fix1.1.er
```

3.6.1.1 誤検知レポート

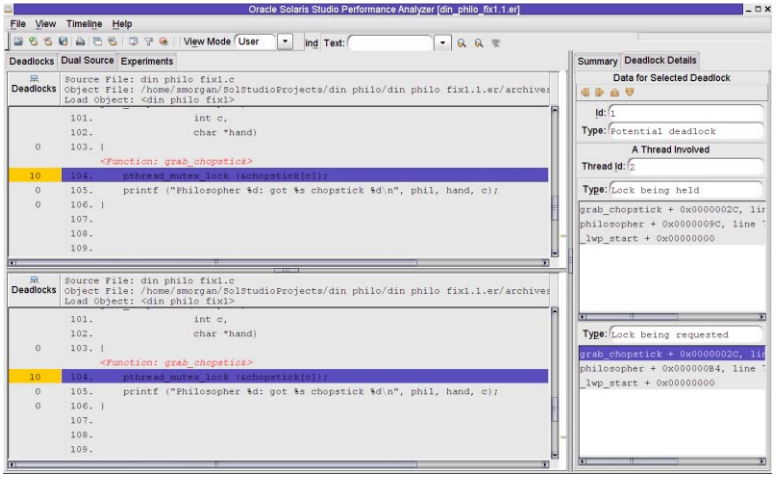
トークンのシステムを使用するときでも、スレッドアナライザは、まったく存在していないのにこの実装の潜在的デッドロックを報告します。これが誤検知です。潜在的デッドロックについて詳しく説明した次のスクリーンショットを見てください。

図 3–5　潜在的デッドロックの誤検知レポート



チェーンの最初のスレッド (スレッド #2) を選択し、「デュアルソース」タブをクリックして、スレッド #2 がアドレス `0x216a8` でロックを保持していたソースコード位置と、アドレス `0x216c0` でロックを要求したソースコードでの位置を確認します。次の図には、スレッド #2 の「デュアルソース」タブが示されています。

図 3–6　誤検知の潜在的デッドロックのソース



`din_philo_fix1.c` の `get_token()` 関数は、`while` ループを使用してスレッドを同期します。スレッドは、トークンの取得に成功するまで、`while` ループ外に出ません (こ

れは、num_can_eat が 0 より大きいときに起こります)。while ループは同時に食事する人を 4 人に制限します。ただし、while ループによって実装された同期は、スレッドアナライザには認識されません。スレッドアナライザは、5 人の哲学者全員が同時に箸を掴んで食事しようとしていると想定しているので、潜在的デッドロックを報告します。次の節では、スレッドアナライザが認識する同期を使用することによって、同時に食事する人の人数を制限する方法について詳しく説明します。

3.6.2 トークンの代替システム

次のリストには、トークンのシステムを実装する代替方法が示されています。この実装方法でも 4 つのトークンを使用するので、同時に 4 人しか食事しようとしません。ただし、この実装方法は、sem_wait() と sem_post() セマフォールーチンを使用して、食事する哲学者の人数を制限します。このバージョンのソースファイルは din_philo_fix2.c と呼ばれます。

ヒント – サンプルアプリケーションをダウンロードした場合、din_philo_fix2.c ファイルを SolarisStudioSampleApplications/ThreadAnalyzer/din_philo ディレクトリからコピーできます。

次のリストでは、din_philo_fix2.c について詳しく説明します。

```
 1  /*
 2   * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights Reserved.
 3   * @(#)din_philo_fix2.c 1.3 (Oracle) 10/03/26
 4   */
 5
 6  #include <pthread.h>
 7  #include <stdio.h>
 8  #include <unistd.h>
 9  #include <stdlib.h>
10  #include <errno.h>
11  #include <assert.h>
12  #include <semaphore.h>
13
14  #define PHILOS 5
15  #define DELAY 5000
16  #define FOOD 100
17
18  void *philosopher (void *id);
19  void grab_chopstick (int,
20                       int,
21                       char *);
22  void down_chopsticks (int,
23                        int);
24  int food_on_table ();
25  void get_token ();
26  void return_token ();
27
28  pthread_mutex_t chopstick[PHILOS];
```

```
29    pthread_t philo[PHILOS];
30    pthread_mutex_t food_lock;
31    int sleep_seconds = 0;
32    sem_t num_can_eat_sem;
33
34
35    int
36    main (int argn,
37          char **argv)
38    {
39        int i;
40
41        pthread_mutex_init (&food_lock, NULL);
42        sem_init(&num_can_eat_sem, 0, PHILOS - 1);
43        for (i = 0; i < PHILOS; i++)
44            pthread_mutex_init (&chopstick[i], NULL);
45        for (i = 0; i < PHILOS; i++)
46            pthread_create (&philo[i], NULL, philosopher, (void *)i);
47        for (i = 0; i < PHILOS; i++)
48            pthread_join (philo[i], NULL);
49        return 0;
50    }
51
52    void *
53    philosopher (void *num)
54    {
55        int id;
56        int i, left_chopstick, right_chopstick, f;
57
58        id = (int)num;
59        printf ("Philosopher %d is done thinking and now ready to eat.\n", id);
60        right_chopstick = id;
61        left_chopstick = id + 1;
62
63        /* Wrap around the chopsticks. */
64        if (left_chopstick == PHILOS)
65            left_chopstick = 0;
66
67        while (f = food_on_table ()) {
68            get_token ();
69
70            grab_chopstick (id, right_chopstick, "right ");
71            grab_chopstick (id, left_chopstick, "left");
72
73            printf ("Philosopher %d: eating.\n", id);
74            usleep (DELAY * (FOOD - f + 1));
75            down_chopsticks (left_chopstick, right_chopstick);
76
77            return_token ();
78        }
79
80        printf ("Philosopher %d is done eating.\n", id);
81        return (NULL);
82    }
83
84    int
85    food_on_table ()
86    {
87        static int food = FOOD;
```

```
 88        int myfood;
 89
 90        pthread_mutex_lock (&food_lock);
 91        if (food > 0) {
 92            food--;
 93        }
 94        myfood = food;
 95        pthread_mutex_unlock (&food_lock);
 96        return myfood;
 97    }
 98
 99    void
100    grab_chopstick (int phil,
101                    int c,
102                    char *hand)
103    {
104        pthread_mutex_lock (&chopstick[c]);
105        printf ("Philosopher %d: got %s chopstick %d\n", phil, hand, c);
106    }
107
108    void
109    down_chopsticks (int c1,
110                     int c2)
111    {
112        pthread_mutex_unlock (&chopstick[c1]);
113        pthread_mutex_unlock (&chopstick[c2]);
114    }
115
116
117    void
118    get_token ()
119    {
120        sem_wait(&num_can_eat_sem);
121    }
122
123    void
124    return_token ()
125    {
126        sem_post(&num_can_eat_sem);
127    }
```

この新しい実装方法は、セマフォー num_can_eat_sem を使用して、同時に食事できる哲学者の人数を制限します。セマフォー num_can_eat_sem は、哲学者の人数より 1 少ない 4 に初期化されます。食事しようとする前に、哲学者は get_token() を呼び出し、続いてこれが sem_wait(&num_can_eat_sem) を呼び出します。sem_wait() の呼び出しは、呼び出した哲学者をセマフォーの値が正になるまで待機させ、続いてセマフォーの値を 1 を引いて変更します。哲学者は食事を終えると、return_token() を呼び出し、続いてこれが sem_post(&num_can_eat_sem) を呼び出します。sem_post() は 1 を追加してセマフォーの値を変更します。スレッドアナライザは、sem_wait() および sem_post() の呼び出しを認識し、哲学者全員が同時に食事しようとしているわけではないと判断します。

注 – 適切なセマフォールーチンとリンクするように、`-lrt` を付けて `din_philo_fix2.c` をコンパイルする必要があります。

`din_philo_fix2.c` をコンパイルするには、次のコマンドを使用します。

```
cc -g -lrt -o din_philo_fix2 din_philo_fix2.c
```

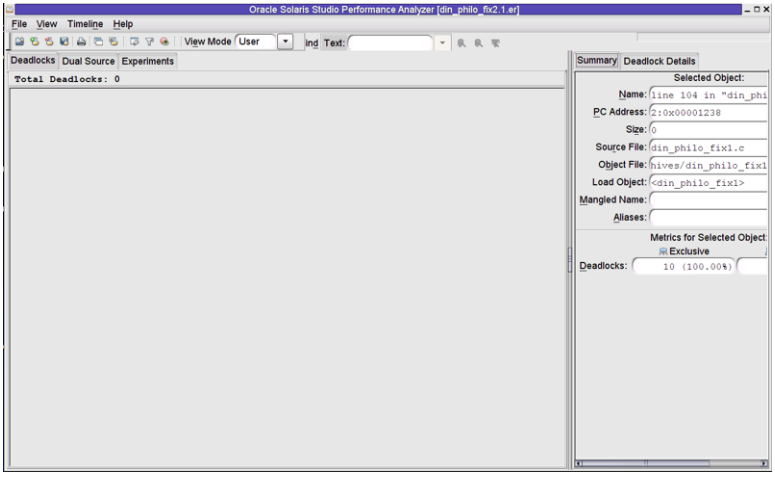
プログラム `din_philo_fix2` のこの新しい実装方法を複数回実行すると、どの場合も通常どおり終了し、ハングアップしないことがわかります。

この新しいバイナリで実験を作成する

```
collect -r deadlock -o din_philo_fix2.1.er din_philo_fix2
```

次の図に示すように、`din_philo_fix2.1.er` の実験から、スレッドアナライザが実デッドロックも潜在的デッドロックも報告していないことがわかります。

図 3–7　din_philo_fix2.c で報告されないデッドロック



スレッドアナライザが認識するスレッドおよびメモリー割り当て API のリストについては、付録 A「スレッドアナライザで認識される API」を参照してください。

◆ ◆ ◆ 付 録 A

スレッドアナライザで認識される API

スレッドアナライザは、OpenMP 指令、POSIX スレッド、および Solaris スレッドで提供されるほとんどの標準同期 API および構文を認識できます。ただし、このツールはユーザー定義の同期を認識できないため、このような同期を採用した場合、誤検出データ競合が報告される場合があります。たとえば、アセンブリ言語でコードを直接書いて実装したスピンロックは、このツールでは認識できません。

A.1 スレッドアナライザユーザー API

コードにユーザー定義の同期が含まれる場合、この同期を識別するために、スレッドアナライザがサポートするユーザー API をプログラムに挿入します。このように識別することによって、スレッドアナライザは同期を認識でき、誤検知の数を減らすことができます。スレッドアナライザのユーザー API は `libtha.so` に定義されます。その一覧を次の表に示します。

表 A–1 スレッドアナライザユーザー API

`tha_notify_acquire_lock()`	プログラムがユーザー定義のロックを取得しようとする直前に呼び出せます。
`tha_notify_lock_acquired()`	ユーザー定義のロックが正しく取得された直後に呼び出せます。
`tha_notify_acquire_writelock()`	プログラムが書き込みモードでユーザー定義の読み取り/書き込みロックを取得しようとする直前に呼び出せます。
`tha_notify_writelock_acquired()`	書き込みモードでユーザー定義の読み取り/書き込みロックが正しく取得された直後に呼び出せます。
`tha_notify_acquire_readlock()`	プログラムが読み取りモードでユーザー定義の読み取り/書き込みロックを取得しようとする直前に呼び出せます。

表 A–1 スレッドアナライザユーザー API (続き)

tha_notify_readlock_acquired()	読み取りモードでユーザー定義の読み取り/書き込みロックが正しく取得された直後に呼び出せます。
tha_notify_release_lock()	ユーザー定義のロックまたは読み取り/書き込みロックが解除される直前に呼び出せます。
tha_notify_lock_released()	ユーザー定義のロックまたは読み取り/書き込みロックが正しく解除された直後に呼び出せます。
tha_notify_sync_post_begin()	ユーザー定義の送信同期が実行される直前に呼び出せます。
tha_notify_sync_post_end()	ユーザー定義の送信同期が実行された直後に呼び出せます。
tha_notify_sync_wait_begin()	ユーザー定義の待機同期が実行される直前に呼び出せます。
tha_notify_sync_wait_end()	ユーザー定義の待機同期が実行された直後に呼び出せます。
tha_check_datarace_mem()	データ競合検出の実行中に、指定されたメモリーブロックへのアクセスを監視または無視することをスレッドアナライザに指示します。
tha_check_datarace_thr()	データ競合検出の実行中に、1 つ以上のスレッドによるメモリーアクセスを監視または無視することをスレッドアナライザに指示します。

C/C++ バージョンと Fortran バージョンの API が用意されています。それぞれの API 呼び出しは単一の引数 ID を取り、その値は同期オブジェクトを一意に識別します。

C/C++ バージョンの API では、引数の型は `uintptr_t` であり、これは 32 ビットモードでは 4 バイト長、64 ビットモードでは 8 バイト長になります。API を呼び出すときには、`#include <tha_interface.h>` を C/C++ ソースファイルに追加する必要があります。

Fortran バージョンの API では、引数の型は `tha_sobj_kind` の整数であり、これは 32 ビットおよび 64 ビットモードで 8 バイト長になります。このバージョンの API を呼び出すときには、`#include "tha_finterface.h"` を Fortran ソースファイルに追加する必要があります。

同期オブジェクトが一意に識別されるよう、引数の ID には同期オブジェクトごとに異なる値を割り当てる必要があります。これを行う 1 つの方法は、同期オブジェクトのアドレスの値を ID として使用することです。次のコード例では、API を使用して誤検出データ競合を回避する方法を示しています。

例 A–1 スレッドアナライザ API を使用して誤検知のデータの競合を回避する例

```
# include <tha_interface.h>
...
/* Initially, the ready_flag value is zero */
...
/* Thread 1: Producer */
100   data = ...
101   pthread_mutex_lock (&mutex);
      tha_notify_sync_post_begin ((uintptr_t) &ready_flag);
102   ready_flag = 1;
      tha_notify_sync_post_end ((uintptr_t) &ready_flag);

103   pthread_cond_signal (&cond);
104   pthread_mutex_unlock (&mutex);


/* Thread 2: Consumer */
200   pthread_mutex_lock (&mutex);
      tha_notify_sync_wait_begin ((uintptr_t) &ready_flag);
201   while (!ready_flag) {
202       pthread_cond_wait (&cond, &mutex);
203   }
      tha_notify_sync_wait_end ((uintptr_t) &ready_flag);
204   pthread_mutex_unlock (&mutex);
205   ... = data;
```

ユーザー API については、libtha(3) のマニュアルページを参照してください。

A.2 認識されるその他の API

以降の節では、スレッドアナライザが認識するスレッド API について詳しく説明します。

A.2.1 POSIX スレッド API

これらの API については、Oracle Solaris ドキュメントの『『マルチスレッドのプログラミング』』を参照してください。

```
pthread_mutex_lock()
pthread_mutex_trylock()
pthread_mutex_unlock()
pthread_rwlock_rdlock()
pthread_rwlock_tryrdlock()
pthread_rwlock_wrlock()
pthread_rwlock_trywrlock()
pthread_rwlock_unlock()
```

```
pthread_create()
pthread_join()
pthread_cond_signal()
pthread_cond_broadcast()
pthread_cond_wait()
pthread_cond_timedwait()
pthread_cond_reltimedwait_np()
pthread_barrier_init()
pthread_barrier_wait()
pthread_spin_lock()
pthread_spin_unlock()
pthread_spin_trylock()
pthread_mutex_timedlock()
pthread_mutex_reltimedlock_np()
pthread_rwlock_timedrdlock()
pthread_rwlock_reltimedrdlock_np()
pthread_rwlock_timedwrlock()
pthread_rwlock_reltimedwrlock_np()
sem_post()
sem_wait()
sem_trywait()
sem_timedwait()
sem_reltimedwait_np()
```

A.2.2 Solaris スレッド API

これらの API については、Oracle Solaris ドキュメントの『『マルチスレッドのプログラミング』』を参照してください。

```
mutex_lock()
mutex_trylock()
mutex_unlock()
rw_rdlock()
rw_tryrdlock()
rw_wrlock()
rw_trywrlock()
rw_unlock()
thr_create()
thr_join()
cond_signal()
cond_broadcast()
cond_wait()
cond_timedwait()
```

```
cond_reltimedwait()
sema_post()
sema_wait()
sema_trywait()
```

A.2.3 メモリー割り当て API

```
calloc()
malloc()
realloc()
valloc()
memalign()
```

メモリー割り当て API については、`malloc`(3C) のマニュアルページを参照してください。

A.2.4 メモリー操作 API

```
memcpy()
memmove()
memchr()
memcmp()
memset()
```

メモリー操作 API については、`memcpy`(3C) のマニュアルページを参照してください。

A.2.5 文字列操作 API

```
strcat()
strncat()
strlcat()
strcasecmp()
strncasecmp()
strchr()
strrchr()
strcmp()
strncmp()
strcpy()
strncpy()
strlcpy()
strcspn()
```

```
strspn()
strdup()
strlen()
strpbrk()
strstr()
strtok()
```

文字列操作 API については、`strcat`(3C) のマニュアルページを参照してください。

A.2.6 OpenMP API

スレッドアナライザは、バリアー、ロック、クリティカル領域、不可分 (アトミック) 領域、taskwait などの OpenMP 同期を認識します。

詳細は、『Oracle Solaris Studio 12.3: OpenMP API ユーザーガイド』を参照してください。

◆ ◆ ◆ 付 録 B

役に立つヒント

この付録には、スレッドアナライザを使用するときのヒントが記されています。

B.1 アプリケーションのコンパイル

実験結果の収集前にアプリケーションをコンパイルするためのヒント

- アプリケーションバイナリを構築するときに `-g` コンパイラオプションを使用します。これにより、スレッドアナライザでデータの競合の行番号情報を報告できます。
- アプリケーションバイナリを構築するときに、`-xO3` より低い最適化レベルでコンパイルします。コンパイラの変換は、行番号情報を歪め、結果をわかりにくくさせます。
- スレッドアナライザは、73 ページの「A.2.3 メモリー割り当て API」 で示したメモリー割り当てルーチンで割り込み処理をします。アーカイブバージョンのメモリー割り当てライブラリにリンクすると、誤検知のデータ競合が報告される場合があります。

B.2 データ競合検出用アプリケーションの計測

実験結果を収集する前に、データの競合検出用アプリケーションを計測するためのヒント

- コンパイラオプション `-xinstrument=datarace` が不正であることを示すエラーメッセージが、コンパイラから表示された場合、スレッドアナライザをサポートしていない古いバージョンの Sun Studio コンパイラを使用しています。使用しているコンパイラのバージョンは、コマンド `cc -Version` と入力すると確認できます。スレッドアナライザをサポートする最も古いバージョンは、2006 年 6 月の日付のものです。
- 次に示すように、データの競合の検出用にバイナリが計測されていない場合、`collect -r race` コマンドは、警告を発行します。

```
% collect -r races a.out
  WARNING: Target ‘a.out’ is not instrumented for datarace
  detection; reported datarace data may be misleading
```

- `nm` コマンドを使用して、`tha` ルーチンの呼び出しを検索することにより、データの競合の検出用にバイナリが計測されているかどうかを判断できます。名前が `__tha_` で始まるルーチンが表示されたら、バイナリは計測されています。例出力は次のとおりです。

  ソースレベルの計測

```
% cc -xopenmp -g -xinstrument=datarace source.c
% nm a.out | grep __tha_
  [71] | 135408| 0|FUNC |GLOB |0 |UNDEF |__tha_get_stack_id
  [53] | 135468| 0|FUNC |GLOB |0 |UNDEF |__tha_src_read_w_frame
  [61] | 135444| 0|FUNC |GLOB |0 |UNDEF |__tha_src_write_w_frame
```

  バイナリレベルの計測

```
% cc -xopenmp -g source.c
% discover -i datarace -o a.out.i a.out
% nm a.out.i | grep __tha_
  [88] | 0| 0|NOTY |GLOB |0 |UNDEF |__tha_read_w_pc_frame
  [49] | 0| 0|NOTY |GLOB |0 |UNDEF |__tha_write_w_pc_frame
```

- `discover` を使用するには、いずれかのコンパイラ最適化フラグ (`-xO1`、`-xO2`、`-xO3`、`-xO4`、`-xO5`) で入力バイナリをコンパイルする必要があります。コンパイラは、Oracle Solaris Studio 12 Update 1 以降のリリースのものである必要があります。オペレーティングシステムは、Oracle Solaris 10 Update 5 または Oracle Solaris 11 以上である必要があります。そうでない場合は、次に示すように警告が表示される場合があります。

```
% discover -i datarace -o a.out.i a.out
  discover (warning): a.out has no annotations. Results may be
  incomplete. See discover documentation for compiler flag/OS
  recommendation
```

  また、バイナリがコンパイラオプション `-xbinopt=prepare` を付けてコンパイルされた場合は、SPARC ベースのシステムで実行中の、以前の Solaris バージョンでも

discover ツールを使用できることがあります。このコンパイラオプションについては、cc(1)、CC(1)、または f95(1) のマニュアルページを参照してください。

B.3 collect を使用したアプリケーションの実行

データの競合およびデッドロックを検出するために、計測したアプリケーションを実行するためのヒント。

- Oracle Solaris システムにすべての必須パッチがインストールされていることを確認します。collect コマンドは、見つからない必須パッチを一覧表示します。OpenMP アプリケーションの場合、libmtsk.so の最新バージョンが必要です。
- -r race または -r deadlock 引数が認識されないというエラーメッセージが collect から表示された場合、スレッドアナライザをサポートしていない古いバージョンの collect を使用しています。使用している collect のバージョンは、collect -Version コマンドを入力することによって確認できます。スレッドアナライザをサポートする最も古いバージョンは、2006 年 6 月の日付のものです。
- 計測は、実行時間の大幅な減速 (50 倍以上) と、メモリー消費量の増大を引き起こす可能性があります。より小さなデータセットを使用することにより、実行時間を減らそうと試みることができます。また、スレッド数を増やすことによって、実行時間を減らそうと試みることもできます。
- データ競合を検出するには、アプリケーションが複数のスレッドを使用していることを確認します。OpenMP の場合、スレッド数は、環境変数 OMP_NUM_THREADS を、目的のスレッド数に設定し、環境変数 OMP_DYNAMIC を FALSE に設定することによって指定できます。

B.4 データの競合の報告

データの競合の報告のヒント

- スレッドアナライザは、実行時にデータの競合を検出します。アプリケーションの実行時の動作は、使用される入力データセットとオペレーティングシステムのスケジュールによって異なります。異なるスレッド数と、異なる入力データセットで collect 下でアプリケーションを実行します。また、ルールがデータの競合を検出するチャンスを最大にするために、単一のデータセットでの実験を繰り返します。
- スレッドアナライザは、単一のプロセスから生じた異なるスレッド間でのデータの競合を検出します。異なるプロセス間でのデータの競合は検出しません。
- スレッドアナライザは、データの競合でアクセスされた変数の名前を報告しません。ただし、2 つのデータの競合アクセスが行われたソース行を調べ、このソース行で変数が書き込まれ、読み取られたかを判断することによって、変数の名前を判別できます。

- 場合によっては、スレッドアナライザは、プログラムで実際には起きなかったデータの競合を報告することがあります。これらのデータの競合は誤検知と呼ばれます。これは通常、ユーザーが実装した同期が使用される場合や、メモリーがスレッド間でリサイクルされる場合に起こります。たとえば、スピンロックを実装する手製アセンブリがコードに含まれる場合、スレッドアナライザはこれらの同期ポイントを認識しません。スレッドアナライザのユーザー API に対する呼び出しをソースコードに挿入して、ユーザー定義の同期についてスレッドアナライザに通知します。詳しくは、36 ページの「2.5 誤検知」および付録 A「スレッドアナライザで認識される API」を参照してください。
- ソースレベルの計測を使用して報告されたデータの競合とバイナリレベルの計測を使用して報告されたデータの競合は、同じでない場合があります。バイナリレベルの計測の場合、それらがプログラム内で静的にリンクされているか、`dlopen()` によって動的に開かれているかにかかわらず、共有ライブラリは開いているときに、デフォルトで計測されます。ソースレベルの計測の場合、ライブラリは、そのソースが `-xinstrument=datarace` でコンパイルされている場合にのみ計測されます。