Нижегородский государственный университет им. Н.И.Лобачевского

Межфакультетская магистратура по системному и прикладному программированию для многоядерных компьютерных систем

Учебный курс «Технологии разработки параллельных программ»

Раздел «Отладка параллельной программы»

Intel Thread Checker – краткое описание

Разработчики: А.В.Сысоев, И.Б. Мееров

Нижний Новгород
2007

Содержание

1. Назначение Intel Thread Checker 3
2. Возможности Intel Thread Checker 4
3. Принцип сбора информации 4
4. Подготовка программы для анализа 5
5. Создание проекта в Intel Thread Checker 5
6. Сбор и анализ данных 5
7. Пример использования Intel Thread Checker 7
   7.1. Описание примера 7
   7.2. Изучение примера 7
   7.3. Подготовка программы для анализа 9
   7.4. Создание проекта в Intel Thread Checker 11
   7.5. Анализ собранной информации 11
   7.6. Причина и устранение 12
8. Заключение 13
9. Литература 13
   9.1. Использованные источники информации 13
   9.2. Рекомендуемая литература 13

Ни для кого не секрет, наличие инструментов делает жизнь проще. Тезис этот находит подтверждение как в повседневной жизни – чистить одежду удобнее и эффективнее щеткой, чем руками, заворачивать гайки гораздо проще ключом, чем пальцами – так и в профессиональной деятельности – кто сейчас возьмется строить, ладно, даже не дом, пусть всего лишь баню, при помощи одного лишь топора. Естественно программисты – не исключение. Значение инструментов, облегчающих путь от анализа постановки задачи до получения решения, готового к внедрению, трудно переоценить. Данный документ представляет собой краткое описание инструмента отладки параллельных программ, разработанного корпорацией Intel и носящего название Intel® Thread Checker.

В первом разделе документа приводится назначение рассматриваемого инструмента, характеризуются области его возможного применения. Во втором дается краткая характеристика принципов работы Intel® Thread Checker. В третьем и четвертом разделах приводится информация, необходимая для подготовки пользовательского проекта и инструмента для анализа. Пятый раздел посвящен вопросам сбора и анализа данных, полученных в результате работы Intel® Thread Checker. В шестом возможности инструмента рассмотрены на простом примере, входящем в поставку Intel® Thread Checker.

Итак, приступим.

1. Назначение Intel Thread Checker

Процесс отладки в общем случае можно разбить на следующие шаги:

- определение факта наличия ошибки;
- поиск (локализация) ошибки;
- выяснение причин ошибки;
- определение способа устранения ошибки;
- устранение ошибки.

Кажется, что на первом шаге никакой инструмент не требуется. Запускаем программу и либо на некоторых исходных данных получаем неверные результаты, либо обнаруживаем, что некоторая последовательность действий по использованию программы ведет к ее «падению» или «зависанию». Однако для параллельной программы даже этот очевидный шаг может иметь существенную сложность. На практике нередко встречаются ситуации, когда неработоспособность параллельной программы проявляется один раз на сотню и более запусков. Очевидно, в этом случае инструментальная поддержка лишней не будет.

Назначение Intel® Thread Checker (ITC) – поиск мест с возможным недетерминированным поведением многопоточной программы, написанной как на основе библиотеки потоков (Windows или POSIX threads), так и с использование технологии OpenMP. Соответственно ITC может быть использован как под операционными системами семейства Windows, так и под различными ОС семейства Linux. Принципы поиска ошибок рассмотрены ниже, здесь же укажем, что ITC неплохо справляется с задачей обнаружения факта ошибки в программе, даже если эта ошибка в текущем варианте исполнения программы и не проявила себя.

Второй шаг – поиск ошибки заключается в необходимости как можно более точной ее локализации, в идеале должна быть найдена переменная с неверным значением и/или строка кода, ведущая к краху программы. Типичный метод работы в этом пункте – использование режима трассировки в отладчике с наблюдением за состоянием переменных, регистров, стека вызова и т.д. «Плохая новость» – для многопоточных программ режим трассировки практически неприменим, поскольку автоматически меняет характер их выполнения, а значит, скрывает места, которые могут приводить к проблемам во время реальной работы. Кстати говоря, даже типичный способ локализации ошибки расстановкой операторов печати по тексту программы в этом случае нужно использовать с большой осторожностью – печать также вносит синхронизацию в выполнение программы.

Что же делать? Быть может, наилучшее из возможных решение реализовано в ITC. ITC не есть привычный всем отладчик с режимами трассировки, наблюдения и т.д. ITC выполняет анализ программы сам, без участия программиста, причем анализируется не только выполненный «прогон» программы, а все возможные варианты ее выполнения. В результате выясняются и показываются программисту места в программе, в которых содержатся ошибки (с той или иной долей вероятности, в большинстве случаев близкой к 100%).

Шаг третий – выяснение причин ошибки. Задача здесь – понять, почему ошибка возникла. Отсюда во многих случаях автоматически вытекает способ ее устранения (задача шага четвертого). Можно, конечно, выяснить условия, ведущие к проявлению ошибки (некое сочетание значений переменных, например) и просто вставить в код заплатку именно для этого случая. Данный вариант мы здесь не рассматриваем. На этом шаге ITC помогает тем, что каждое найденное им проблемное место

сопровождает комментарием, содержащим тип ошибки: гонка данных, несинхронизированный доступ к переменной, тупик и т.д.

В результате мы получаем место потенциальной ошибки, переменную, с которой связана проблема и описание ошибки. Остается лишь освоить типовые способы борьбы с типовыми ошибками и значительная их часть будет находиться и исправляться без грандиозных усилий.

Единственное в чем ITC совсем не может помочь – это шаг пятый. Устранять найденную ошибку все-таки придется программисту самостоятельно.

2. Возможности Intel Thread Checker

Согласно [2] ITC обнаруживает ошибки следующих видов: *гонки данных* (*data races*), *тупики* (*deadlocks*), *потоки в состоянии ожидания* (*stalled threads*), *потерянные сигналы* (*lost signals*), *заброшенные замки* (*abandoned locks*).

Приведем краткое описание каждого вида.

- **Гонки данных**. Возникают, когда несколько потоков работают с разделяемыми данными и конечный результат зависит от соотношения скоростей потоков. Пусть, например, один поток выполняет над общей переменной `x` операцию `x = x + 3`, а второй поток – операцию `x = x + 5`. Данные операции для каждого потока фактически разбиваются на три отдельных подоперации: считать `x` из памяти, увеличить `x`, записать `x` в память. В зависимости от взаимного порядка выполнения потоками подопераций финальное значение переменной `x` может быть больше исходного на 3, 5 или 8. Гонка данных возможна и в случае, когда один поток пишет в переменную, а остальные только читают из нее.
- **Тупики**. Взаимная блокировка потоков, ожидающих наступление некоторого события для продолжения работы. Типичный пример тупика, когда нулевой поток занял для использования ресурс 1 и ожидает предоставления ему ресурса 2, а первый поток занял ресурс 2 и ожидает предоставления ему ресурса 1.
- **Потоки в состоянии ожидания**. Одно из состояний потока в многозадачной операционной системе – ожидание. Поток переходит в него, когда для продолжения выполнения ему требуется наступление некоторого внешнего события. Если пребывание потока в этом состоянии продолжается слишком долго, ITC рапортует об ошибке типа stalled thread. Интервал времени, по истечении которого выдается данная диагностика, может быть задан в настройках ITC.
- **Потерянные сигналы**. Возникают, когда поток ожидает наступление некоторого события, произошедшего прежде, чем поток пришел в состояние готовности к его приему и обработке. В результате поток никогда не сможет выйти из состояния ожидания.
- **Заброшенные замки**. Возникают в ситуации, когда поток захватил некоторый ресурс (критическую секцию, мьютекс) и был снят с выполнения по той или иной причине. В результате ресурс не может быть освобожден. Если он требуется другому потоку, это приведет к бесконечному ожиданию.

3. Принцип сбора информации

Анализ программы, выполняемый ITC, основан на процедуре инструментации. *Инструментация* – вставка обращений к библиотеке ITC для записи действий, потенциально способных привести к ошибкам: работа с памятью, вызовы операций синхронизации и работа с потоками [2]. Может выполняться автоматически на уровне исполняемого модуля (а также dll-библиотеки) и/или по указанию программиста на уровне исходного кода. Для достоверности получаемых результатов крайне желательно, чтобы во время сборки анализируемой программы была выключена оптимизация (сборка в конфигурации **debug** не обязательна).

В процессе анализа контролируется:

- доступ к памяти;
- операции синхронизации;
- операции создания потоков.

Необходимо отметить, что неисполняемые участки (не вызываемые функции, ветки условных переходов и т.д.) никак не проверяются, то есть под анализ не подпадают.

4. Подготовка программы для анализа

Использование отладчика Intel® Thread Checker возможно в двух режимах:

- *Бинарная инструментация программы* – осуществляется автоматически в момент запуска **Активности**[1] (**Activity**) в проекте ITC. Рекомендуется в случае, если отсутствует доступ к исходным кодам или невозможна повторная сборка программы с нужными ITC настройками.
- *Компиляторная инструментация* – при сборке анализируемой программы необходимо указать ключ компилятора **/Qtcheck**. Позволяет ITC предоставить информацию о найденных ошибках с указанием имен переменных, с которыми эти ошибки связаны.

Сборка приложения для работы с ITC предполагает установку следующих опций проекта (или настроек в make-файле):

- Компиляция потоко-безопасного кода: **-MT[d]**, **-MD[d]**. Данные опции автоматически устанавливаются при сборке в конфигурации **debug**. При сборке в конфигурации **release** указанные опции необходимо устанавливать вручную.
- Использование debug опций: **-Z[i,I,7]**, **-Od**. Замечание аналогичное предыдущему пункту.
- Связывание с ключом **/fixed:no**. Необходимо указывать явно.

Дополнительно необходимо отметить, что при использовании ключа **/Qtcheck** в среде разработки (IDE) требуется указать путь к библиотекам ITC. Обычно этот путь имеет вид **C:\Program Files\Intel\VTune\Analyzer\Lib**.

5. Создание проекта в Intel Thread Checker

Работа в ITC выполняется в рамках **проекта**. Для его создания используется команда меню **File→New Project**. В главном окне мастера настройки проекта (см. рис. 1) необходимо выполнить всего лишь два действия. Первое – указать исполняемый файл (**Launch an application**). Второе (необязательное) – указать аргументы командной строки.

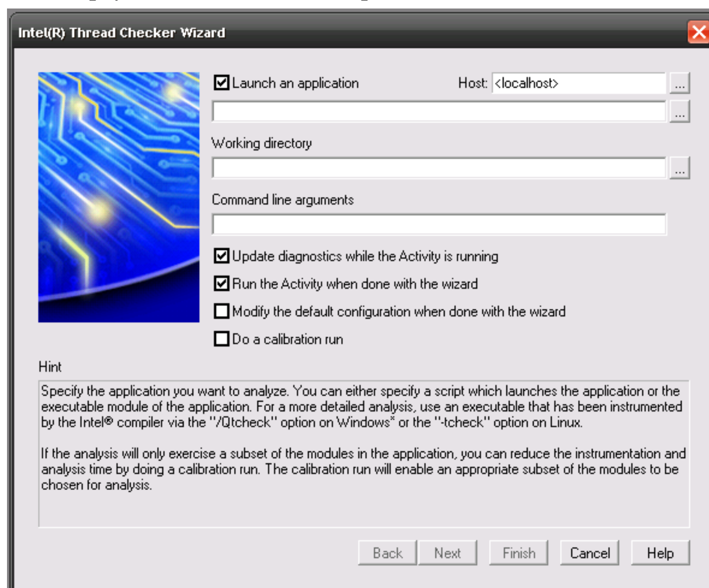


Рис. 1. Мастер настройки проекта в Intel Thread Checker (версия 3.0)

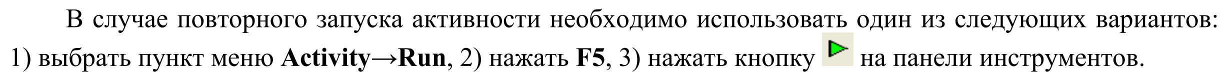
Рекомендуется при анализе программы, с одной стороны, использовать типичные размеры обрабатываемых данных, с другой, задавать их так, чтобы программа «убиралась» в оперативную память с учетом накладных расходов ITC, которые могут быть довольно значительными.

6. Сбор и анализ данных

После запуска в проекте ITC активности (при создании проекта это происходит автоматически) начинается инструментация исполняемого модуля, указанного для анализа, и используемых им

[1] Активность – термин из среды VTune, в рамках которой работает ITC. Фактически представляет собой «контейнер», содержащий, с одной стороны, настройки системы и параметры анализируемой программы, с другой, результаты проводимого анализа.

динамических библиотек. Затем модуль запускается и начинается процесс анализа. По завершении ITC формирует окно с информацией о найденных ошибках и подозрительных местах. Возможный его вид указан на рис. 2.

В случае повторного запуска активности необходимо использовать один из следующих вариантов: 1) выбрать пункт меню **Activity→Run**, 2) нажать **F5**, 3) нажать кнопку  на панели инструментов.

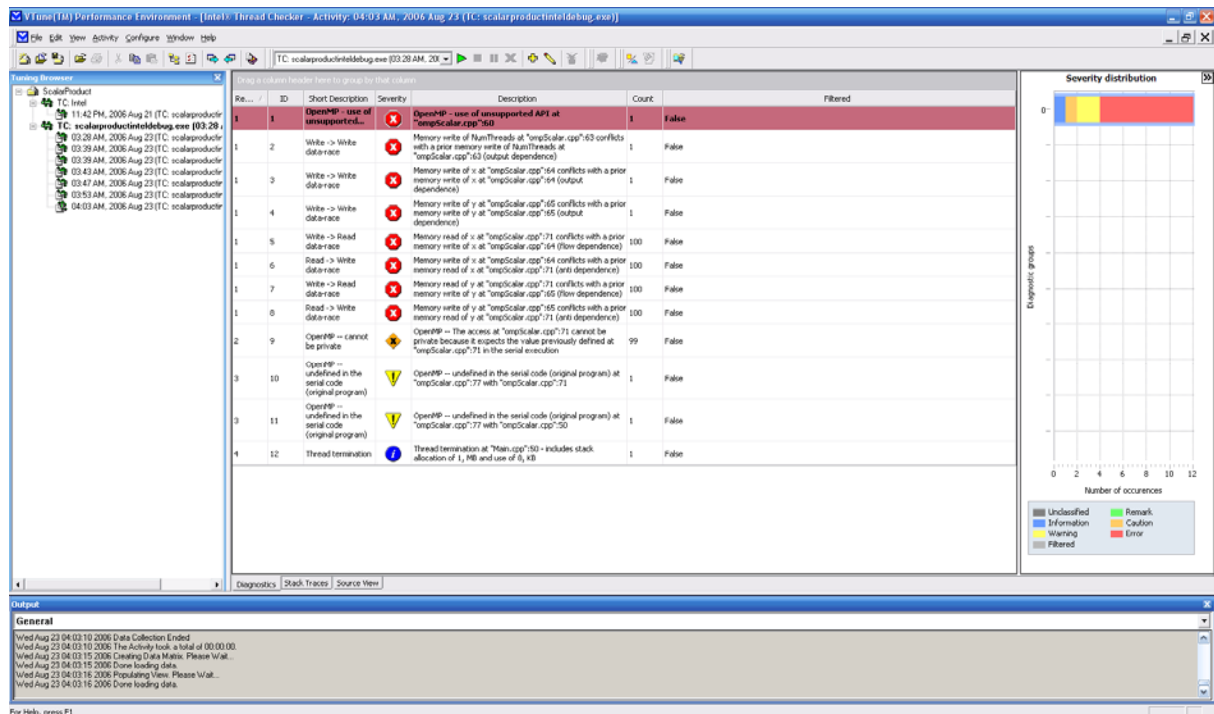


Рис. 2. Результат анализа – список диагностики

По каждой диагностике, выданной ITC, в случае если сборка выполнялась с приведенными в разделе 4 настройками, может быть получена дополнительная информация (см. раздел 7), вид которой показан на рис. 3.

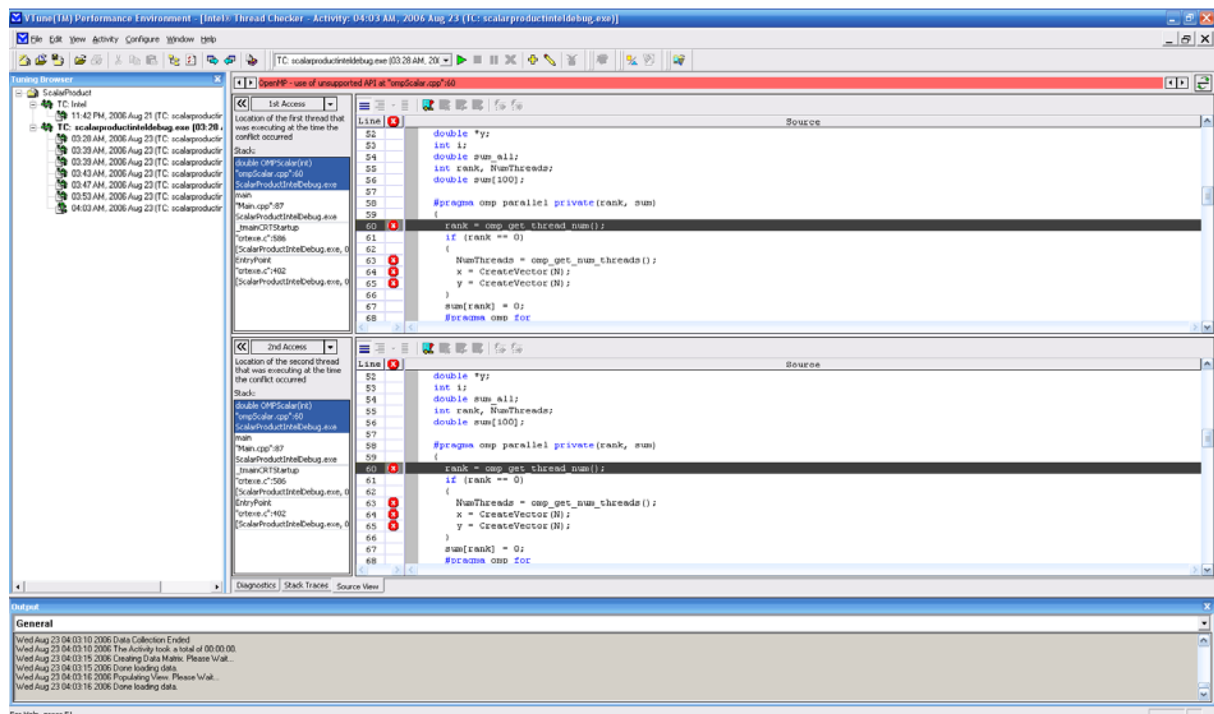


Рис. 3. Результат анализа – диагностика в исходном коде

7. *Пример использования Intel Thread Checker*

Для начального знакомства с ITC рассмотрим пример, входящий в поставку инструмента версии 3.0 и описанный в [3].

7.1. Описание примера

Наличие у каждого потока в многопоточном приложении доступа к общему ВАП[2] процесса позволяет потокам эффективно обмениваться данными, с одной стороны, и является основным источником ошибок, с другой. Гонки данных, они же *конфликты доступа* (*storage conflicts*), поджидают зазевавшегося программиста буквально на каждом шагу. Поиск таких ошибок методом «пристального взгляда» – задача весьма сложная.

Рассматриваемый пример создает 4 потока, каждый из которых увеличивает значение общей глобальной переменной **globalX** и использует критическую секцию для синхронизации доступа к ней. Однако, несмотря на наличие критической секции, в коде все-таки содержится конфликт доступа. Нам предстоит его обнаружить, выяснить причину и исправить ситуацию.

7.2. Изучение примера

Откройте проект **DataRaces**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**,
- в меню **File** выполните команду **Open→Project/Solution…**,
- в диалоговом окне **Open Project** выберите папку **C:\ITCLabs\DataRaces**,
- дважды щелкните на файле **DataRaces.dsw** или, выбрав файл, выполните команду **Open**;
- согласитесь с предложением конвертировать проект **DataRaces.dsp** в новый формат.

После открытия проекта в окне **Solution Explorer** дважды щелкните на файле исходного кода **DataRaces.c**, как это показано на рис. 4. После этих действий программный код, с которым предстоит работать, будет открыт в редакторе кода **Microsoft Visual Studio 2005**.

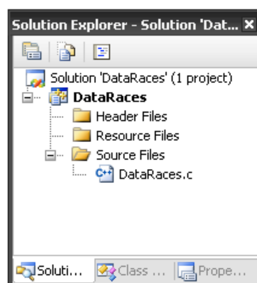


Рис. 4. Открытие файла DataRaces.c

Интерес в этом небольшом файле представляет потоковая функция **increment**.

```
int globalX = 0;

DWORD WINAPI increment (void *arg)
{
  CRITICAL_SECTION cs;

  InitializeCriticalSection (&cs);

  EnterCriticalSection (&cs);
    globalX++;
  LeaveCriticalSection (&cs);

  DeleteCriticalSection (&cs);

  return 0;
}
```

На первый взгляд код выглядит корректно. Доступ к переменной **globalX** защищен. Посмотрим, что покажет запуск программы.

Соберите и запустите пример, выполнив следующие действия:

- в меню **Build** выполните команду **Build Solution**,

[2] ВАП – виртуальное адресное пространство процесса в операционной системе.

- проигнорируйте предупреждение компилятора Microsoft о неизвестной опции **/Qtcheck**;
- в меню **Debug** выполните команду **Start Without Debugging**.

Убедитесь, что вывод на экран соответствует представленному на рис. 5.

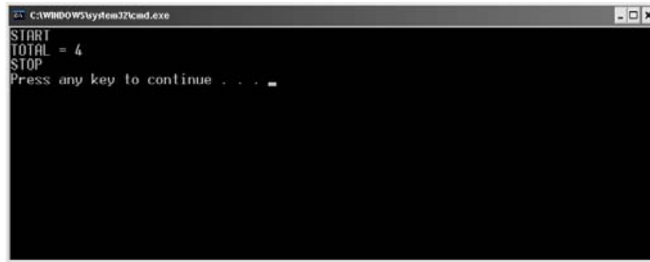


Рис. 5. Результаты работы примера DataRaces (исходный вариант)

Кажется, все верно. Число потоков равно четырем. Начальное значение переменной **`globalX`**, как нетрудно убедиться, равно нулю. Таким образом, результат верен. Повторите запуск программы несколько раз и убедитесь, что результат стабильно равен 4.

Может быть, программа корректна? Подумаем, что нужно для того, чтобы гонки данных могли проявиться. Необходимо, чтобы имел место разный порядок выполнения потоков во времени. Посмотрим, возможно ли это в данном примере. Конечно же, нет! Вот участок функции **`main`**, запускающий потоки:

```
  for (i = 0; i < NTHREADS; i++)
  {
    h[i] = CreateThread (0, 0, increment, NULL, 0, NULL);
  }
```

Нетрудно понять, что потоки создаются последовательно, по мере выполнения цикла. Учитывая крайнюю простоту, а значит, и малое время выполнения потоковой функции, скорее всего и работа потоков происходит последовательно. Конфликт доступа к переменной **`globalX`**, даже если он имеет место, не успевает проявиться.

Что ж, изменим немного код. Пусть каждый поток увеличивает значение переменной **`globalX`** не один раз, а многократно.

```
DWORD WINAPI increment (void *arg)
{
  int i;
  CRITICAL_SECTION cs;

  InitializeCriticalSection (&cs);

  for (i = 0; i < 10000; i++)
  {
    EnterCriticalSection (&cs);
    globalX++;
    LeaveCriticalSection (&cs);
  }

  DeleteCriticalSection (&cs);

  return 0;
}
```

Добавьте в код выделенные в предыдущем фрагменте строки, соберите и запустите пример. Убедитесь, что вывод на экран соответствует представленному на рис. 6.

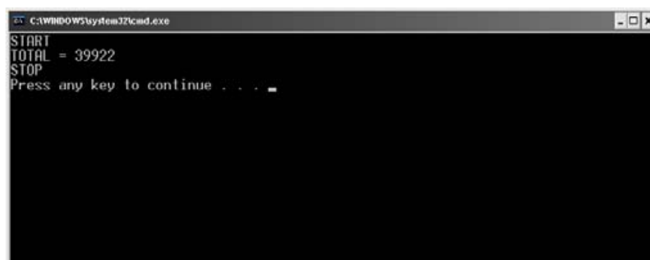


Рис. 6. Результаты работы примера DataRaces (версия с циклом)[3]

[3] Конкретное значение в строке «TOTAL = …» может отличаться от указанного

Не правда ли, неожиданно?! Вместо 40000 на экране совсем другое число! Повторите запуск несколько раз и убедитесь, что результат работы программы будет меняться. Типичная ситуация для гонки данных.

Итак, приложив некоторые усилия, мы прошли два этапа в рассмотренном в п. 1 процессе отладки. Наличие ошибки обнаружено, локализация ее также не вызывает сложностей в силу малого размера кода в примере. Осталось понять, в чем же причина ошибки. Но прежде посмотрим, как с ее обнаружением справится ITC.

7.3. Подготовка программы для анализа

Выполните следующие действия для подготовки программы к анализу.

1. Верните код примера к исходному состоянию.
2. Конвертируйте проект для использования компилятора Intel® C++ Compiler. В окне **Solution Explorer** выберите файл проекта, щелкните правой кнопкой мыши и в контекстном меню выполните команду **Convert to use Intel® C++ Project System**.
3. В меню **Project** выберите пункт **Properties**, в появившемся окне настроек проекта в дереве слева выберите узел **Configuration Properties→C/C++→General** (рис. 7). В открывшейся таблице справа убедитесь, что значение поля **Debug Information Format** равно **Program Database (/Zi)**.

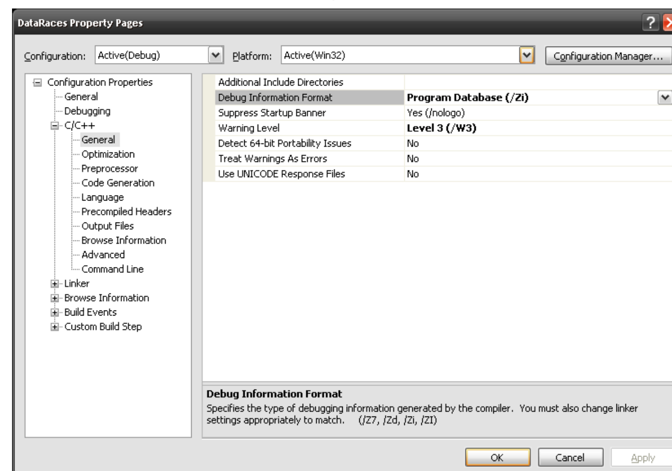


Рис. 7. Указание формата отладочной информации

4. В дереве слева выберите узел **Configuration Properties→C/C++→Optimization** (рис. 8). В открывшейся таблице справа убедитесь, что значение поля **Optimization** равно **Disabled(/Od)**.

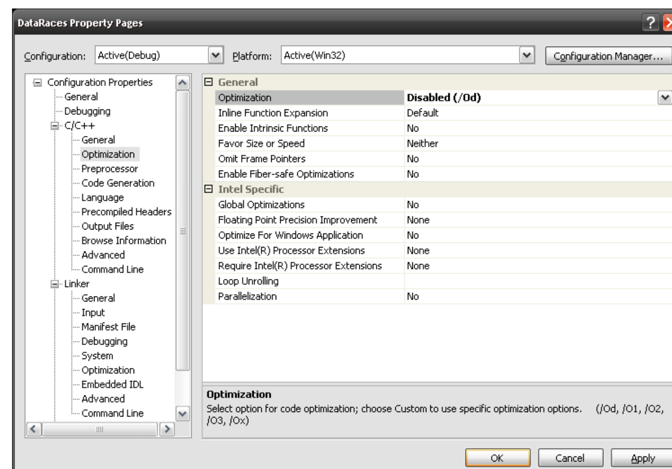


Рис. 8. Отключение оптимизации

5. В дереве слева выберите узел **Configuration Properties→C/C++→Code Generation** (рис. 9). В открывшейся таблице справа убедитесь, что значение поля **Runtime library** установлено в **Multi-threaded Debug DLL (/MD)**.

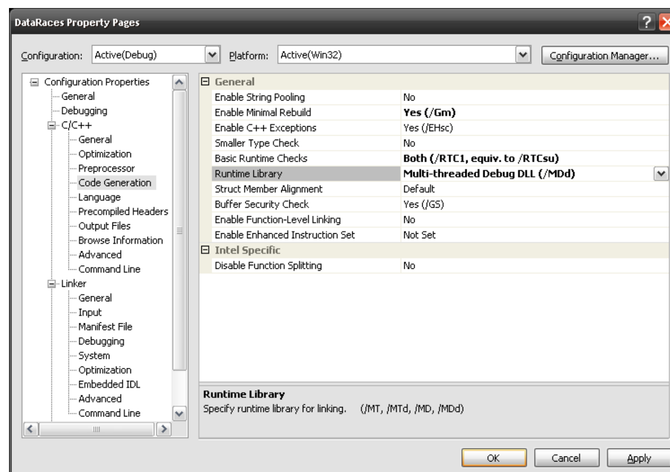


Рис. 9. Выбор потокобезопасных библиотек

6. В дереве слева выберите узел **Configuration Properties→Linker→Command Line** (рис. 10). Убедитесь, что сборка программы выполняется с использованием опции компоновщика **/FIXED:NO**.

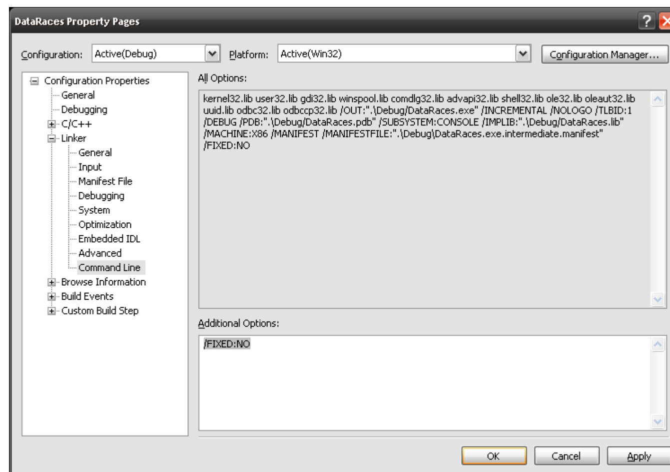


Рис. 10. Установка опций компоновщика

7. Убедитесь, что включена компиляторная инструментация кода. В дереве слева выберите узел **Configuration Properties→C/C++→Command Line** (рис. 11). Убедитесь, что в поле **Additional Options** установлен ключ компилятора **/Qtcheck**.

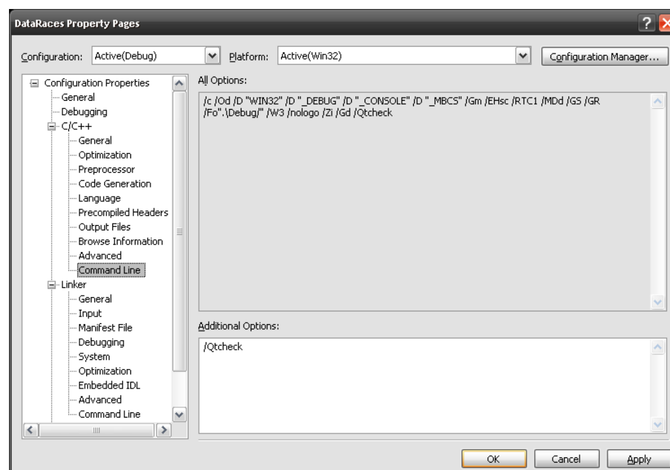
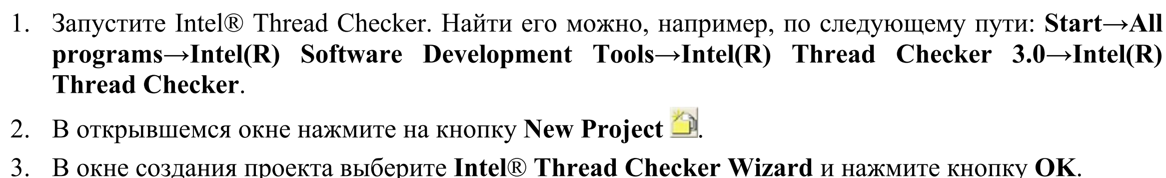


Рис. 11. Установка компиляторного режима инструментации

После выполнения указанных действий программа готова к анализу в ITC.

7.4. Создание проекта в Intel Thread Checker

1. Запустите Intel® Thread Checker. Найти его можно, например, по следующему пути: **Start→All programs→Intel(R) Software Development Tools→Intel(R) Thread Checker 3.0→Intel(R) Thread Checker**.
2. В открывшемся окне нажмите на кнопку **New Project** .
3. В окне создания проекта выберите **Intel® Thread Checker Wizard** и нажмите кнопку **OK**.

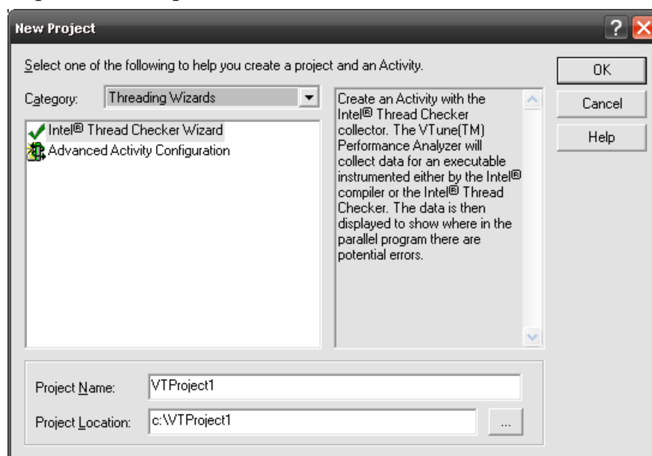


Рис. 12. Выбор типа проекта

4. В окне мастера в поле **Launch an application** укажите путь к исполняемому файлу **C:\ITCLabs\DataRaces\Debug\DataRaces.exe**.

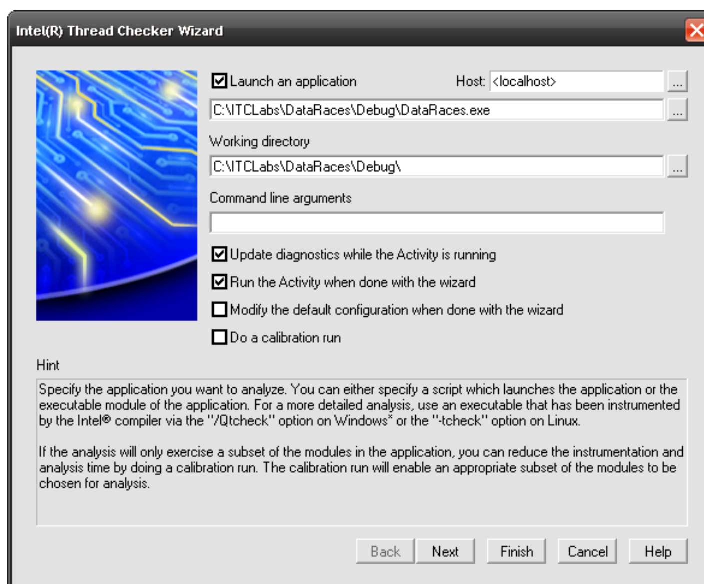


Рис. 13. Выбор программы для анализа

5. Нажмите кнопку **Finish**.

После этого запустится ITC, произведет инструментацию программы и начнет анализ.

7.5. Анализ собранной информации

По окончании процесса сбора информации ITC представит результаты в виде, показанном на рис. 14. Итак, мы видим, что ITC нашел в предложенном коде 3 ошибки. При ближайшем рассмотрении видно, что все они указывают на одну и ту же переменную **`globalX`**. Краткие комментарии к диагностикам показывают, что ITC указал все возможные комбинации неверного обращения к переменной **`globalX`**:

- когда первый поток записывает новое значение в переменную **`globalX`**, а второй в это время читает из нее;
- когда первый поток читает из переменной **`globalX`**,а второй в это время в нее пишет;

- когда оба потока одновременно пишут в переменную **`globalX`**.

Несмотря на то, что реально работало 4 потока, очевидно, что для демонстрации ошибки достаточно двух. Именно так ITC всегда и комментирует гонки данных.

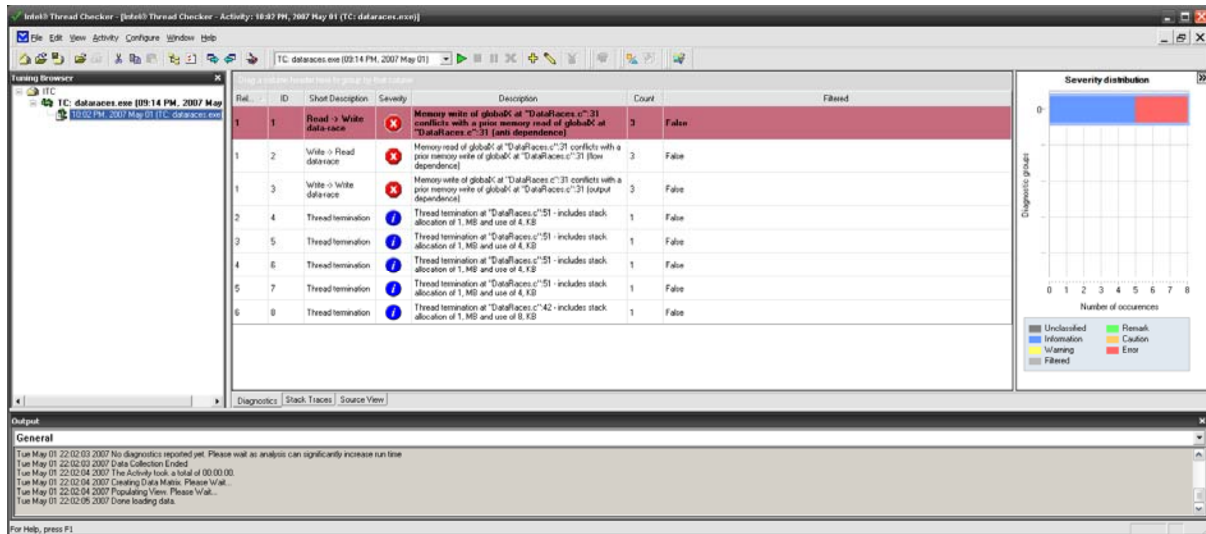


Рис. 14. Результат анализа примера DataRaces – Diagnostics

При наличии отладочной информации ITC может показать в исходном коде местоположение ошибки. Выберите любую из найденных ошибок и двойным щелчком по ней перейдите к просмотру исходного кода (рис. 15).

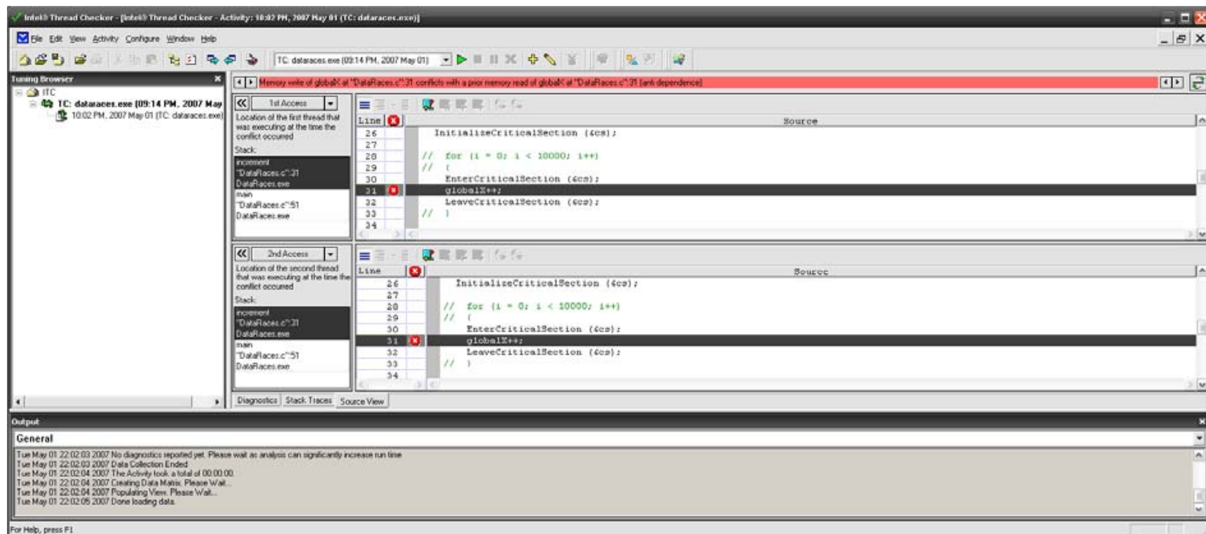


Рис. 15. Результат анализа примера DataRaces – Source View

Комментарий в красном поле над исходными текстами описывает ситуацию. В представленном на рисунке случае имеет место конфликт доступа типа «запись-чтение».

7.6. Причина и устранение

Осталась самая малость – понять, в чем причина гонки данных в рассматриваемом примере, и устранить ее. Сделать это на самом деле не так уж сложно. Противоречие с тем фактом, что обращение потоков к переменной **`globalX`** происходит внутри критической секции, а гонка данных все равно имеется, кажущееся. Проблема в данном случае не в критической секции, а в объекте, на котором она основана (**`CRITICAL_SECTION cs`**). Этот объект объявлен внутри потоковой функции, а значит, является локальным для каждого потока. То есть, когда один поток захватывает критическую секцию, он делает это для своего локального объекта **`cs`**, к которому остальные потоки не имеют доступа.

Исправить ситуацию можно несколькими способами, но в любом из них объявление объекта **`cs`** нужно вынести из потоковой функции **`increment`**, а инициализацию секции и освобождение ресурсов поместить в функцию **`main`**.

Возможный корректный вариант представлен ниже.

```
#include <stdio.h>
#include <windows.h>

#define NTHREADS 4

int globalX = 0;
CRITICAL_SECTION cs;

DWORD WINAPI increment (void *arg)
{
  EnterCriticalSection (&cs);
    globalX++;
  LeaveCriticalSection (&cs);

  return 0;
}

int main (int argc, char *argv[])
{
  HANDLE h[NTHREADS];
  DWORD  rc;
  int i;

  printf ("START\n");

  InitializeCriticalSection (&cs);

  for (i = 0; i < NTHREADS; i++)
  {
    h[i] = CreateThread (0, 0, increment, NULL, 0, NULL);
  }

  rc = WaitForMultipleObjects (NTHREADS, h, TRUE, INFINITE);

  DeleteCriticalSection (&cs);

  printf ("TOTAL = %d\n", globalX);
  printf ("STOP\n");
}
```

8. Заключение

Ни для кого не секрет, наличие инструментов делает жизнь проще. Перефразируем: наличие Intel® Thread Checker существенно скрашивает суровые будни разработчика многопоточных программ. Убедиться в этом еще раз читатель сможет в лабораторной работе «Отладка параллельной программы с использованием Intel Thread Checker», в которой приводится дополнительная информация по ITC.

9. Литература

9.1. Использованные источники информации

1. Intel® Thread Checker for Windows*. Getting Started Guide. Version 3.0. — Intel Corporation, 2006.
2. Intel® Thread Checker Help. Version 3.0. — Intel Corporation, 2006.
3. Intel® Thread Checker. Guide to Sample Code. Version 3.0. — Intel Corporation, 2006.

9.2. Рекомендуемая литература

4. Andrews, G. R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming.. – Reading, MA: Addison-Wesley (русский перевод Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Издательский дом «Вильямс», 2003).
5. Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.