



Intel(R) Thread Checker for Linux*

Guide to Diagnostics

Copyright © 2006 Intel Corporation

All Rights Reserved

Document Number: 313494-001 US

Revision: 1.0

World Wide Web: <http://www.intel.com>



Disclaimer and Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

The software described in this document may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

This document as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Developers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Improper use of reserved or undefined features or instructions may cause unpredictable behavior or failure in developer's software code when running on an Intel processor. Intel reserves these features or instructions for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from their unauthorized use.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Chips, Core Inside, Dialogic, EtherExpress, ETOX, FlashFile, i386, i486, i960, iCOMP, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, MMX logo, Optimizer logo, OverDrive, Paragon, PDCharm, Pentium, Pentium II Xeon, Pentium III Xeon, Performance at Your Command, Pentium Inside, skool, Sound Mark, The Computer Inside., The Journey Inside, VTune, Xeon, Xeon Inside and Xircom are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands may be claimed as the property of others.

Copyright © 2006, Intel Corporation.

Revision History

Document Number	Revision Number	Description	Revision Date
313494 US	001	Initial release.	June 2006



Contents

1	About this Document.....	5
1.1	Intended Audience	5
1.2	Using This Guide to Diagnostics.....	5
1.3	About Diagnostics.....	5
1.3.1	Severity	6
1.4	Conventions and Symbols.....	7
1.5	Related Information.....	7
2	API Violation Diagnostics	8
2.1	A Condition Variable Object is Expected	8
2.2	A Mutex Object is Expected.....	8
2.3	A Reader-Writer Lock Object is Expected	9
2.4	Argument is a Non-Existent Object	9
2.5	Argument is an Invalid Value	10
2.6	Function Call Fails	10
2.7	Incorrect NULL (Zero) Argument.....	10
3	Data Access Diagnostics.....	11
3.1	A Thread is Accessing the Stack of Another Thread (Cross-thread Stack Access) or Illegal Memory	11
3.2	Non-Reentrant API Function.....	12
3.3	Read -> Write Data Race.....	13
3.4	Write -> Read Data Race.....	14
3.5	Write -> Write Data Race	15
4	OpenMP* Usage Diagnostics.....	16
5	Synchronization Usage Diagnostics	17
5.1	A Sync Object was Acquired in the Wrong Order	17
5.2	A Thread Died while Owning a Sync Object	17
5.3	A Thread is Deadlocked	18
5.4	A Thread is Holding and Waiting	18
5.5	A Thread is Stalled	19
5.6	A Thread is Waiting	19
5.7	A Thread with a Time-Out is Holding and Waiting	19
5.8	A Thread with a Time-Out is Stalled	20
5.9	A Thread with a Time-Out is Waiting	21
5.10	The Sync Object Notify Operation was a No-op	21
5.11	Unlock Sync Object that is not Owned	21
6	Miscellaneous Diagnostics.....	22
6.1	Invalid Memory Access.....	22



- 6.2 Inaccessible Allocated Memory 22
- 6.3 Memory Limit Reached 23
- 6.4 Number of Diagnostics Exceeds Limit 23
- 6.5 Thread Termination 23
- 6.6 No Additional Information Available 24
- 7 Correction Strategies 25
 - 7.1 Correction Strategies for Wait Diagnostics..... 25
 - 7.2 Correction Strategies for Stall Diagnostics..... 25
 - 7.3 Correction Strategies for Read -> Write Diagnostics 26
 - 7.4 Correction Strategies for Holding and Waiting Diagnostics 27



1 About this Document

This documents contains details about different diagnostics identified by the Intel® Thread Checker. It also contains correction strategies for fixing certain types of threading issues.

1.1 Intended Audience

This document is intended for users of Intel® Thread Checker for Linux*.

1.2 Using This Guide to Diagnostics

This Guide to Diagnostics organizes descriptions of Thread Checker diagnostics according to the following categories:

Table 1 Document Organization

Section	Description
Diagnostics by Category	Detailed descriptions of diagnostics according to the following categories: <ul style="list-style-type: none"> • API Violation Diagnostics • Data Access Diagnostics • OpenMP* Usage Diagnostics • Synchronization Usage Diagnostics • Miscellaneous Diagnostics
Correction Strategies	Possible solutions for specific types of diagnostics.

1.3 About Diagnostics

Diagnostics indicate issues or events that are non-deterministic, and that may lead to indeterminate or wrong results. Usually they are due to a conflict (data race) which is due to two different threads accessing the same memory location at the same time without the proper synchronization needed to guarantee consistent results.



1.3.1 Severity

Diagnostics are flagged by severity. In general, resolving the issues marked as more severe first is the most effective method of examining a list of diagnostics. The following table describes the different severity categories in order of priority, from most severe to least severe.

Rank	Severity	Examples
4	Error	Data races, deadlocks, and other serious issues fall into this category.
3	Warning	Inaccessible memory.
2	Caution	A thread trying to release a lock which it does not own, or a notify operation that occurred when no other thread was waiting for it, making it a no-op.
1	Informational	Messages indicating the amount of stack space allocated.
0	Remark	Too many errors to display.



1.4 Conventions and Symbols

The following conventions are used in this document.

Table 2 Conventions and Symbols used in this Document

This type style	Indicates an element of syntax, reserved word, keyword, filename, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant.
This type style	Indicates the exact characters you type as input. Also used to highlight the elements of a graphical user interface such as buttons and menu names.
<i>This type style</i>	Indicates a placeholder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the placeholder.
[<i>items</i>]	Indicates that the items enclosed in brackets are optional.
{ <i>item</i> <i>item</i> }	Indicates that only one of the items listed between the braces to be selected. A vertical bar () separates the items.

1.5 Related Information

The following additional resources will help you get the most out of Thread Checker:

- **Intel® Thread Checker for Linux* Getting Started Guide.** This guide presents a threaded code example and teaches you how to use Intel® Thread Checker to identify and handle threading-related issues. After completing this guide, you should be ready to analyze and repair your own code using Thread Checker.
- **Command Line Usage Help.** A full listing of available Thread Checker command-line options are available in the help. To access the help, run:

```
> tcheck_cl --help
```
- **Release Notes.** Key product details, updated information on requirements, technical support, and known limitations is available in the product Release Notes. Open **Release Notes** from (for example) `/tcheck/doc`.
- **Code Samples.** Find code Samples in the `tcheck/samples` directory.
- **Intel® Thread Profiler.** After you check your code with Intel® Thread Checker, use the **Intel® Thread Profiler** to help you improve its performance. Find details about Thread Profiler and other Intel software development products at: <http://www.intel.com/software/products/>.



2 API Violation Diagnostics

This section contains descriptions of API Violation Diagnostics and offers possible solutions when available.

2.1 A Condition Variable Object is Expected

An argument passed to one of the `pthread_cond_destroy()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`, `pthread_cond_wait()`, or `pthread_cond_timedwait()` functions is not valid. The specified argument is identified by number from left to right, where the first argument is number 1.

The 1st Access shows no information. The 2nd Access source view shows the location of the function call where the incorrect argument is passed.

Cause	Possible Solution
A condition variable is used that has not been initialized at all or initialized incorrectly.	Verify that the specified argument is a pointer to a condition variable that has been successfully initialized with the <code>pthread_cond_init()</code> function.
A variable with the wrong type is being used as a function argument.	Be sure to pass properly initialized variables of type <code>pthread_cond_t* condVar</code> where expected.

2.2 A Mutex Object is Expected

An invalid argument is passed in one of the `pthread_mutex_*` family of functions. Functions `pthread_mutex_lock()`, `pthread_mutex_trylock()`, `pthread_mutex_unlock()` or `pthread_mutex_destroy()` expects a valid mutex synchronization object which has been initialized successfully by calling `pthread_mutex_init()`.

The argument number is counted from left to right, where the first argument is number one.

2nd Access source view shows the location of the procedure or function call where the argument is incorrect.

Cause	Possible Solution
A mutex is not initialized.	Verify that the specified argument is successfully



	initialized with the <code>pthread_mutex_init()</code> function.
A synchronization object of the wrong type was passed.	Use the appropriate API for the synchronization object.

2.3 A Reader-Writer Lock Object is Expected

The `pthread_rwlock*` family of functions such as the `pthread_rwlock_destroy()`, `pthread_rwlock_rdlock()`, `pthread_rwlock_tryrdlock()`, or `pthread_rwlock_trywrlock()` functions expect a pointer to an initialized read-write lock. The argument number is counted from left to right, where the first argument is number 1.

The 1st Access shows no information. The 2nd Access source view shows the location of the called function where the argument is incorrect.

Cause	Possible Solution
A read-write lock variable was used that was not been initialized at all or initialized incorrectly.	Verify that the specified argument is a pointer to a read-write lock variable that has been successfully initialized with the <code>pthread_rwlock_init()</code> function.
A variable with the wrong type is being used as a function argument.	Be sure to pass only variables of type <code>pthread_rwlock_t* rwlockVar</code> .

2.4 Argument is a Non-Existent Object

The specified argument is the wrong type or has not been properly initialized for the procedure or function call. The specified argument is identified by number from left to right, where the first argument is number one.

Show the location of the procedure or function call where the argument is incorrect in the 2nd Access source view.

Cause	Possible Solution
The argument in the specified position is the wrong type for the indicated function.	Use an argument of the correct type for this function.
The argument in the specified position is not correctly initialized for the indicated function.	Verify that the specified argument is correctly initialized for this function.



2.5 Argument is an Invalid Value

A value passed as an argument is not valid. The specified argument is identified by number from left to right, where the first argument is number 1.

The 2nd Access shows the location of the API function call where the argument value is incorrect.

2.6 Function Call Fails

A system API call failed and returned an error code.

The 1st Access shows no information. The 2nd Access source view shows where the failed call was made.

Consult the documentation for the system API concerned to determine the possible causes for the failure.

2.7 Incorrect NULL (Zero) Argument

A NULL (zero) argument in the specified position is incorrect for the called function.

The specified argument is identified by number from left to right, where the first argument is number one.

Show the location of the procedure or function call where the argument is incorrect in the 2nd Access source view.



3 *Data Access Diagnostics*

This section contains descriptions of Data Access diagnostics and possible solutions when available.

3.1 **A Thread is Accessing the Stack of Another Thread (Cross-thread Stack Access) or Illegal Memory**

A cross-thread stack access occurs when a thread is accessing the stack of a different thread, even if the access is ordered or synchronized.

A thread's stack is private and usually should not be accessed by any other threads. When a thread's stack is accessed by another thread without mutual cooperation and awareness of both threads, unintended behaviors such as invalid or illegal memory access or thread's stack corruption may occur.

Sometimes cross-thread stack access is acceptable or even desirable. For example, a child thread that accesses a parent's threads stack while the parent is waiting for the child to complete is an acceptable cross-thread stack access.

To avoid cross-thread stack access, use a global variable with synchronization for inter-thread communications. Not exporting any thread stack memory address is also a good programming practice.



This diagnostic may indicate one or more of these situations:

Cause	Possible Solution(s)
The software has indexed out of bounds of an array	This is usually an off-by-one error. In the C/C++ language, arrays are indexed from 0 to N-1, where the declared size of the array is N. To comply with this definition iterations in C/C++ usually are defined as: for (i = 0; i < N; ++i) sum += array[i]
An invalid pointer dereference occurred. When dereferencing a pointer, it's possible that the pointer contains a corrupt or invalid value. For example, the pointer may be incremented past the end or decremented before the beginning of dynamically allocated memory. This pointer address value can be the address of memory that is marked invalid to access.	This is usually a logic error in the sequential algorithm. Make sure when dereferencing a pointer that the pointer refers to a valid object. Also be sure that pointer arithmetic stays within the bounds of the dynamically allocated memory.
This is sometimes a stale reference to memory. If a thread was holding a pointer to the heap, and another thread released (via the <code>free()</code> function) the memory back to the heap, then all future accesses to this memory are invalid.	A clear concept of object ownership is needed to ensure that one thread does not mistakenly release the object back to the heap, while another thread considers the pointer to be valid. Try using reference counting as a solution to this problem.

3.2 Non-Reentrant API Function

A race condition exists on an API function that is not thread-safe. In this case, two threads called the specified API function either without synchronization or with incorrect synchronization. Most commonly the specified API function maintains an internal state that is preserved across calls.

The 1st Access source view shows where the first thread calls to the API function. The 2nd Access source view shows the call by the thread that triggered this diagnostic.

Cause	Possible Solution(s)
Two threads sharing the same file pointer are causing problems. For example, a thread calls <code>fseek()</code> or <code>rewind()</code> to reposition the file pointer then begins reading the file at the new location. A second thread calling either of these functions with the same file pointer disrupts the read position of the first thread. Simply enforcing mutual exclusion to <code>fseek()</code> or <code>rewind()</code> alone does not solve the problem.	Designate a single thread for all file I/O. Open the file multiple times to create a new file pointer for each thread.
Use of non-thread-safe <code>rand()</code> function. The	Use a thread-safe version of the function or



<p><code>rand()</code> function maintains a variable that seeds a pseudo-random number sequence. For a given initial seed value, the sequence is reproducible. If two threads call <code>rand()</code> concurrently, this sequence can become non-deterministic. To further complicate matters, simply enforcing single-thread-at-a-time access to this function does not solve the problem because the interleaving of calls to the <code>rand()</code> function causes the per-thread pseudo-random number sequence to be non-deterministic.</p>	<p>change the algorithm to guarantee a reproducible pseudo-random sequence regardless of the number of threads.</p>
<p>Use of a function that maintains internal state, including the time formatting functions such as <code>ctime()</code>, <code>localtime()</code>, <code>gmtime()</code> and <code>asctime()</code>, the string function <code>strtok()</code>, and the utility function <code>tmpnam()</code>.</p>	<p>For more information, see http://gcc.gnu.org/onlinedocs.</p>

3.3 Read -> Write Data Race

Read->write data races occur when one thread reads a [shared](#) memory location (address) while another thread concurrently writes the same memory location. The shared memory location may be referred to by (variable) name, pointer, or even a function such as `memcpy()`.

The following example uses a variable name:

```
1st access by first thread
S1: privateA = sharedX
2nd access by second thread
S2: sharedX = privateB
```

If `sharedX` is a variable visible to all threads and `privateA` and `privateB` are local variables visible only to the thread where each was declared, concurrent execution of the above statements by multiple threads results in a "race" on the value to be read from `sharedX`.

Since the order of execution among threads is unpredictable, it is unknown what value will be read from `sharedX` and written into `privateA`. This results in non-deterministic software, or software prone to produce different results each time it is executed.

The statement S1 shows in the 1st Access source view; and S2 shows in the 2nd Access source view.

To correct this diagnostic, see Section 7.3, *Correction Strategies for Read -> Write Diagnostics*.



3.4 Write -> Read Data Race

Write->Read data races occur when one thread writes a shared memory location (address) while another thread concurrently reads the same memory location. The shared memory location may be referred to by (variable) name, pointer, or even a function such as `memcpy()`.

The following example uses a variable name:

```
1st access by first thread
S1: sharedX = privateA
2nd access by second thread
S2: privateB = sharedX
```

If `sharedX` is a variable visible to all threads and `privateA` and `privateB` are local variables visible only to the thread where each was declared, concurrent execution of the above statements by multiple threads results in a "race" on the value to be read from `sharedX`.

Since the order of execution among threads is unpredictable, it is unknown which value will be available in `sharedX` to be stored into `privateB`. This results in non-deterministic software, or software prone to produce different results each time it is executed.

The statement S1 shows in the 1st Access source view. S2 shows in the 2nd Access source view.

To correct this diagnostic, try one or more of the following:

- Allocate a private copy of the shared variable for each thread.
- For OpenMP*, consider declaring the variable in a `private`, `firstprivate`, or `lastprivate` clause or making it `threadprivate`.
- Use `alloca()` to allocate memory from each thread's stack, thus making a private copy for each thread.
- Synchronize access to the shared variable using one of the following methods:
 - For OpenMP*, `atomic` or `locks (omp_lock_t)`.
- This diagnostic may indicate you are not using a safe concurrent algorithm. A Write->Read data race indicates that a value is being communicated between the two threads. If you can not eliminate the communication you may need to execute portions of the code with data races in serial or you may need to rewrite your code to be correct for concurrency.
- For OpenMP*, consider declaring the shared variable as a `reduction` if the variable is being used to collect values from multiple iterations of a parallel loop.
- Consider using a local, private variable to collect values from each thread's loop iteration, and then using a mutex to merge each thread's private variable into the shared variable.



3.5 Write -> Write Data Race

Write->write data races occur when more than one thread updates a [shared](#) memory location (address) without synchronization. The shared memory location may be referred to by (variable) name, pointer, or even a function such as `memcpy()`.

The following example uses a variable name:

```
1st access by first thread
S1: sharedX = privateA
2nd access by second thread
S2: sharedX = privateB
```

If `sharedX` is a variable visible to all threads and `privateA` and `privateB` are different local variables visible only to the thread where each was declared, concurrent execution of the above statements by multiple threads causes a "race" on whether the value of `privateA` or `privateB` gets written to `sharedX`. Since the order of execution among threads is unpredictable, the software is non-deterministic or prone to produce different results each time it is executed.

The statement S1 shows in the 1st Access source view; and S2 shows in the 2nd Access source view.

To correct this diagnostic, see Section 7.3, *Correction Strategies for Read -> Write Diagnostics*.



4 *OpenMP* Usage Diagnostics*

Please contact Intel(R) Premier Support at <http://premier.intel.com/> to request information on OpenMP* Usage diagnostics.



5 Synchronization Usage Diagnostics

This section contains descriptions of Synchronization Usage diagnostics and possible solutions when available.

5.1 A Sync Object was Acquired in the Wrong Order

A thread locked more than one synchronization object in a different order than another thread. This situation could potentially cause deadlock depending on the order in which the threads execute.

Consider a thread, thread1, that executes statement S1 and then S2 in order without releasing the first lock obtained during S1:

```
S1: acquire lock(A)  
S2: acquire lock(B)
```

Another thread, thread2, executes statement S3 and then S4 in order and without releasing the first lock obtained during S3:

```
S3: acquire lock(B)  
S4: acquire lock(A)
```

Statement S1 from thread1 shows in the 1st Access source view. The 2nd Access source view shows S2. The source view Access for thread2 is reported in a different diagnostic.

Fix a bad locking hierarchy by having all threads acquire multiple locks in the same order and release them in the reverse order.

5.2 A Thread Died while Owning a Sync Object

The 1st Access view shows no information. The 2nd Access view shows where the abandoned synchronization object was acquired by the dead thread.



5.3 A Thread is Deadlocked

A thread is waiting for a resource that it can never acquire and is therefore not able to proceed. This condition is known as deadlock.

This diagnostic occurs when both of the following conditions occur:

A thread is attempting to gain access to a mutex held by another thread.

The object in question (for example, mutex) would not become available even if the execution were allowed to continue for a longer period of time, provided both threads remain alive.

Every object in a deadlock cycle has its own deadlock diagnostic.

Thread1 which is the owner of the object under contention is shown in the **1st Access** source view. The waiting thread, thread2, shows in the 2nd Access source view.

To correct this flaw, the object in contention and the reason thread2 has deadlocked must be identified and altered.

Cause	Possible Solution(s)
Bad locking hierarchy where thread1 and thread2 each hold a different lock and each thread needs a lock (for example, mutex) held by the other thread in order to proceed.	Fix bad locking hierarchies by having threads acquire multiple locks in the same order and release them in the reverse order.
Each thread is waiting for the termination of the other.	<ul style="list-style-type: none"> • Modify the algorithm to have only a single thread wait for the termination of other threads. • Use an alternate means to notify other threads when a thread has reached a desired point in execution or terminated.
The deadlock involving the indicated object is difficult to correct.	Examine each deadlock diagnostic with the same context to enumerate all of the objects that are involved in the deadlock cycle. It may be easier to remove the contention on a different object that is part of the same deadlock cycle.

5.4 A Thread is Holding and Waiting

A thread, thread2, is idle longer than the **Minimum wait time to report as stall** as specified by setting `TC_OPTIONS=stall=<integer>`. Thread2 continues to wait indefinitely because the wait timeout value is infinite. In addition, another thread, thread1, is waiting to acquire a synchronization object (for example, mutex) owned by thread2.



Thread Checker expands run-time so set the minimum wait time longer than you would need under normal execution conditions.

The 1st Access shows the location where the synchronization object is acquired by thread2. The 2nd Access source view shows where thread2 is waiting. Note that thread1 is not shown in the source view.

See Section 7.4, *Correction Strategies for Holding and Waiting Diagnostics*.

5.5 A Thread is Stalled

A thread is idle longer than the **Minimum wait time to report as stall** as specified by setting `TC_OPTIONS=stall=<integer>`.

A stall diagnostic typically occurs because a thread is attempting to gain access to a mutex owned by another thread.

Since Intel® Thread Checker expands run-time, set the **Minimum wait time to report as stall** to be longer than you would need under normal execution conditions.

See Section 7.2, *Correction Strategies for Stall Diagnostics* for tips on avoiding this diagnostic.

5.6 A Thread is Waiting

A thread is idle longer than the **Minimum wait time to report as stall** as specified by setting `TC_OPTIONS=stall=<integer>`.

A wait diagnostic typically occurs because a thread is attempting to gain access to a semaphore, event or other wait object.

Since Thread Checker expands run-time, set **Minimum wait time to report as stall** to longer than you would need under normal execution conditions.

See Section 7.1, *Correction Strategies for Wait Diagnostics* for tips on correcting these diagnostics.

5.7 A Thread with a Time-Out is Holding and Waiting

A thread, thread2, is idle longer than the Minimum wait time to report as stall as specified by setting `TC_OPTIONS=stall=<integer>`. Thread2 eventually continues its execution after the timeout interval (`stall` parameter) is exceeded. Concurrently, thread1 is waiting to acquire a synchronization object (for example, mutex) owned by thread2.



Remember that Thread Checker expands run-time so set the minimum wait time longer than you would need under normal execution conditions. To avoid this diagnostic and similar diagnostics everywhere increase the minimum wait time to be greater than the timeout interval.

The 1st Access shows the location where the synchronization object is acquired by thread2. The 2nd Access source view shows where thread2 is waiting. Note that thread1 is not shown in the Source View.

If by design thread2 is expected to wait for an extended period of time, then this diagnostic may be benign. However thread waits for an unanticipated length of time may indicate a design flaw. See Section 7.4, *Correction Strategies for Holding and Waiting Diagnostics*.

5.8 A Thread with a Time-Out is Stalled

A thread is idle longer than the **Minimum wait time to report as stall** as specified by setting `TC_OPTIONS=stall=<integer>`.

A stall diagnostic typically occurs because a thread is attempting to gain access to a mutex owned by another thread.

Since Intel® Thread Checker expands run-time, set the **Minimum wait time to report as stall** to be longer than you would need under normal execution conditions.

See Section 7.2, *Correction Strategies for Stall Diagnostics* for tips on avoiding this diagnostic.



5.9 A Thread with a Time-Out is Waiting

A thread is idle longer than the **Minimum wait time to report as stall** as specified by setting `TC_OPTIONS=stall=<integer>`.

Since Thread Checker expands run-time, set **Minimum wait time to report as stall** to longer than you would need under normal execution conditions.

See Section 7.1, *Correction Strategies for Wait Diagnostics* for tips on correcting these diagnostics.

5.10 The Sync Object Notify Operation was a No-op

The synchronization function had no effect. For example, a thread calls `pthread_cond_signal()` while no other threads are waiting for the conditional variable.

The 1st Access shows no information. The 2nd Access source view shows the location where the synchronization function is used but has no effect.

5.11 Unlock Sync Object that is not Owned

A thread is trying to release a lock on a mutex or reader-writer lock synchronization object that it does not have locked.

The 1st Access source view shows no information. The source location where the thread is attempting to un-lock the object will be shown in the 2nd Access source view.

This diagnostic may indicate one or more of these situations:

Cause	Possible Solutions
A thread is releasing a mutex or reader-writer lock it did not acquire.	Always pair the acquire and release operations for mutexes to ensure that every release operation is preceded by an acquire operation by the same thread.



6 Miscellaneous Diagnostics

This section contains descriptions of diagnostics that do not fit into any of the main categories.

6.1 Invalid Memory Access

This diagnostic only occurs if the value for **Allocated memory guard padding** is set by the environment variable `TC_OPTIONS=pad=<non-zero integer>`. When enabled, additional guard memory is allocated and marked as being invalid to access. If these guard memory locations are accessed, then this diagnostic is generated.

The **1st Access** shows the location of a previous access to this same memory location, or it may show no information.

The 2nd Access source view shows the location where the invalid memory access occurred.

6.2 Inaccessible Allocated Memory

Allocated memory has not been freed or de-allocated. This diagnostic only occurs if the value for **Allocated memory guard padding** is not zero by setting `TC_OPTIONS=pad=<non-zero integer>`.

This diagnostic typically occurs when memory allocated from the heap was not freed. Or the memory was freed, but the module that freed it was not instrumented by the Intel® Thread Checker.

Consult the following table to see how to free memory allocated from the heap.

Platform	Memory Allocator	To Free Memory Use
C++ language	new operator	delete operator
C language	malloc(), calloc(), or realloc() functions	free() function
Fortran90	allocate intrinsic	deallocate intrinsic



6.3 Memory Limit Reached

The memory used by the runtime analysis engine reached the preset limit and memory has been reclaimed.

The runtime analysis engine allocates heap memory for its own use during runtime. You can limit the size of the heap used by the analysis engine by setting `TC_OPTIONS=heap=<integer>`.

When the heap limit is set and heap allocated by the analysis engine reaches the limit:

- the analysis engine starts reclaiming memory to bring the total amount of memory allocated down below the preset limit
- Intel(R) Thread Checker generates this diagnostic message
- the analysis engine might fail to detect errors

Be careful when you set the limit. If the limit is too low, the runtime analysis engine reclaims the memory too often, affecting the performance. If the limit is too high, Thread Checker may run out of memory before reaching the limit.

6.4 Number of Diagnostics Exceeds Limit

The number of diagnostics generated is greater than the maximum limit specified in the configuration options. You can change the **Maximum messages of each type to record** value by setting `TC_OPTIONS=max=<integer>`.

The number shown in the diagnostic message is the **Maximum messages of each type to record** value for this Activity. The **Diagnostic type** in this message is the terse form of one of the diagnostic messages in your Thread Checker results. Once this diagnostic appears no more messages of the given type are listed.

6.5 Thread Termination

A thread terminated when the thread executed a `return` from its entry function, called a thread termination function.

The first number in the message is the number of bytes that were initially allocated to the stack of the thread. The second number is the number of bytes that were actually used during the thread's lifetime.

This diagnostic occurs whenever a thread that the Intel® Thread Checker monitored terminates. If your software is using threads that do not generate this diagnostic, verify that the module which created the thread is being instrumented. To do this, check your command line option for setting with `--module_instrumentation_level`, if it is set to minimum for the module which contains the thread.



If a thread terminated unexpectedly, check the thread code for one of the following conditions:

- The thread entry function contains a `return` statement that was unexpectedly executed, perhaps for diagnostic handling.
- Run the code through a traditional debugger to check for any resulting exceptions.

6.6 No Additional Information Available

No additional information is available for this diagnostic. Please contact Intel(R) Premier Support at <http://premier.intel.com/> to request information on this diagnostic.



7 Correction Strategies

This section provides tips on correcting certain types of diagnostics.

7.1 Correction Strategies for Wait Diagnostics

If by design thread2 is expected to wait for an extended period of time, then this warning may be benign. However a thread waiting for an unanticipated length of time may indicate a design flaw.

Cause	Possible Solution
Thread2 is trying to acquire an object that another thread never releases.	Carefully check that all threads release or signal all objects before termination.
Thread2 may be trying to acquire an object that another thread is using for unanticipated lengths of time.	Verify that all threads do not hold any objects for longer than necessary because other threads can be delayed. Threads may be able to release objects earlier by copying global shared variables into local private variables while executing.
Thread2 has called a wait function where the wait time is more than the value of Minimum wait time to report as stall and is not infinite.	To correct this, increase the minimum wait time specified by setting <code>TC_OPTIONS=stall=<integer></code> .

7.2 Correction Strategies for Stall Diagnostics

For stall diagnostics, thread1, the owner of the synchronization object (for example, mutex) shows in the 1st Access source view. The 2nd Access shows where the stalled thread, thread2, is stalled.

If by design thread2 is expected to stall for an extended period of time, then this diagnostic may be benign. However, thread stalls of an unanticipated length of time may indicate a design flaw.



See the following table for possible diagnostic causes and solutions:

Cause	Possible Solutions
Thread2 is waiting for thread1 to complete, and thread1 is in an infinite loop.	Change the algorithm of thread1 so that it terminates.
Thread2 may be trying to acquire a synchronization object that thread1 is using for unanticipated lengths of time.	Check that thread1 does not hold any synchronization objects for longer than necessary to avoid delaying other threads. Thread1 may be able to release synchronization objects earlier by copying global shared variables into local private variables while it runs.
Thread2 has called a wait function where the wait time is more than the value of Minimum wait time to report as stall and is not infinite.	Increase the Minimum wait time to report as stall specified by setting <code>TC_OPTIONS=stall=<integer></code> .

7.3 Correction Strategies for Read -> Write Diagnostics

To correct this diagnostic, try one or more of the following strategies:

- Allocate a private copy of the shared variable for each thread.
 - For OpenMP*, consider declaring the variable in a `private`, `firstprivate`, or `lastprivate` clause or making it `threadprivate`.
 - Use `alloca()` to allocate memory from each thread's stack, thus making a private copy for each thread.
- Synchronize access to the shared variable:
 - For OpenMP*, use `atomic` or `locks (omp_lock_t)`.
- This diagnostic may indicate that you are not using a safe concurrent algorithm. If you must encapsulate large portions of the algorithm, then you may need to rewrite your code to be correct for concurrency.
 - For OpenMP*, consider declaring the shared variable as a `reduction`, if the variable is being used to collect values from multiple iterations of a parallel loop.



7.4 Correction Strategies for Holding and Waiting Diagnostics

Holding and waiting related diagnostics may indicate one or more of the following situations:

Cause	Possible Solutions
<p>There could be a synchronization ordering problem. Thread2 acquires a mutex and waits for an event object to be signaled by thread1. Thread1 is waiting to acquire the mutex from thread2 and when acquired will signal thread2 with the event object.</p>	<p>Be careful about coupling the use of multiple synchronization objects because it can cause stall situations that are very similar to deadlocks. In this particular example you should move the wait-on-event before the mutex is acquired or after it is released.</p>
<p>Thread2 may acquire a mutex and wait for thread1 to terminate, but thread1 wants to acquire and release the mutex before terminating.</p>	<p>Do not hold a synchronization object while waiting for notification from another thread. In this example, release the mutex before waiting and if needed, reacquire it after the notification.</p>