`lock_lint` *User's Guide*

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 801-6692-10
Revision A, August 1994

© 1994 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's font suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS
Sun, the Sun logo, Sun Microsystems, Sun Microsystems Computer Corporation, Solaris, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. UNIX is a registered trademark of Novell, Inc., in the United States and other countries; X/Open Company, Ltd., is the exclusive licensor of such trademark. OPEN LOOK® is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCstorage, SPARCware, SPARCcenter, SPARCclassic, SPARCcluster, SPARCdesign, SPARC811, SPARCprinter, UltraSPARC, microSPARC, SPARCworks, and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOKand Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME

Please
Recycle

Adobe PostScript

Contents

1. Introduction 1

Background 1

`lock_lint` Overview 2

Collecting Information for `lock_lint` 3

`lock_lint` User Interface 4

2. How to Use lock_lint 5

Compiling Code 5

lock_lint Subcommands 5

Starting and Exiting `lock_lint` 6

Tool State 6

Suggested Approach for Checking an Application 6

Program Knowledge Management 8

Loading Sources for Analysis 8

Function Management 8

Variable Management 9

Lock Management 9
Analysis. 9
Post Analysis Queries 10
3. lock_lint Subcommand Reference 11
lock_lint Subcommands 11
analyze 13
assert 16
declare 18
disallow. 20
disallows 21
exit 21
files 21
funcptrs 21
funcs 23
help 27
ignore 27
load 28
locks. 28
members 30
order 31
pointer calls 32
reallow. 33
reallows. 33
refresh 34

restore 34
save 34
saves 35
start 35
sym 37
unassert 37
vars 38
Names 40
Inversions 43
A. Limitations of lock_lint 47
Limitations 47
B. Managing the Interface Environment 53
Shell Interface 53
Temporary Files 54
Makefile Rules 55

Tables

Table 1-1 Calls for Manipulating mutex locks 2

Table 1-2 Calls for Manipulating readers-writer locks 2

Table 1-3 Calls for Manipulating condition variables 3

Table 3-1 Subcommand Exit Status 12

Preface

`lock_lint` performs a static analysis of the use of mutex and multiple readers/single writer locks and looks for inconsistent use of these locking techniques. These inconsistent locking techniques include references to uninitialized variables.

In looking for use of locks, `locklint` detects the most common cause of data races: failure to hold the appropriate lock while accessing a variable.

The "`lock_lint` User's Guide" describes how to use `lock_lint` to verify that your multithreaded C program is using mutex and multiple readers/single writer locks consistently.

Who Should Use This Book

`lock_lint` analyzes multithreaded C programs. The information presented in this manual assumes that the reader is an experienced C programmer familiar with your application and has some experience with using `libthread` calls..

How This Book Is Organized

The lock_lint User's Guide is organized into the following chapters and appendixes:

Chapter 1, "Introduction," is an overview of lock_lint

Chapter 2, "How to Use lock_lint ," explains the preliminaries for using lock_lint.

Chapter 3, "lock_lint Subcommand Reference ," describes the lock_lint subcommands

Appendix A, "Limitations of lock_lint ," describes the limitations of lock_lint.

Appendix B, "Managing the Interface Environment ," provides background information about the interface environment and the ancillary files used by lock_lint.

What Typographic Changes and Symbols Mean

The following table describes the type faces and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`system% su` `Password::`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Code samples are included in boxes and may display the following:

`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`

Introduction 1

Background

In the multithreading model, a process consists of one or more threads of control which share a common address space and most other process resources. The various threads of control may execute in parallel on a multiprocessor; even on a uniprocessor the interleaving of progress on various threads is non-deterministic. Threads must acquire and release locks associated with shared data in order to reliably produce the intended results; where they fail to do so, data may become corrupted. This is a type of "data race" — a situation in which a program may produce different results when run repeatedly with the same input.

Data races are easy problems to introduce—simply accessing a variable without first acquiring the appropriate lock can cause one—and they are generally very difficult to find. Visible symptoms of a data race generally manifest only if two threads access the improperly protected data at nearly the same time; hence a data race may easily run for months without showing any signs of a problem. It would be extremely difficult to exhaustively test the various concurrent states for even a simple multithreaded program, so conventional testing and debugging are not an adequate defense against data races.

Most processes have several shared resources and operations within the application that may require accesses to more than one of the resources while maintaining the *atomicity* of its actions. This means that the operation will need to grab a lock for each of the resources before performing the operation. If

1

different operations use some common set of resources but the order in which they acquire the locks is inconsistent, there is a potential for “deadlock”. For example, the classic case of deadlock is when two threads of control each hold a lock for a different resource and each of these threads is trying to acquire the lock for the resource held by the other thread.

`lock_lint` *Overview*

`lock_lint` performs a static analysis of the use of mutex and multiple readers/single writer locks and looks for inconsistent use of these locking techniques.

In looking for use of locks, `lock_lint` detects a common cause of data races: failure to hold the appropriate lock while accessing a variable.

Table 1-1, Table 1-2, and Table 1-3 list the routines of the Solaris libthread API recognized by `lock_lint` as manipulating these types of synchronization primitives:

Table 1-1 Calls for Manipulating mutex locks

User	**Kernel**
mutex_lock mutex_unlock mutex_trylock	mutex_enter mutex_exit mutex_tryenter

Table 1-2 Calls for Manipulating readers-writer locks

User	**Kernel**
rw_rdlock, rw_wrlock rw_unlock rw_tryrdlock, rw_trywrlock	rw_enter rw_exit rw_tryenter rw_downgrade rw_tryupgrade

Table 1-3 Calls for Manipulating condition variables

User	Kernel
cond_wait	cv_wait cv_wait_sig
cond_timedwait	cv_timedwait cv_timedwait_sig cv_wait_sig_swap
cond_signal	cv_signal

`lock_lint` reports several kinds of basic information about the modules it has analyzed.

- `lock_lint` reports on locking side effects of functions. Unknown side effects can lead to data races or deadlocks.
- `lock_lint` reports accesses to variables that were not consistently protected by at least one lock or violate assertions made about which locks are supposed to protect them. This information can point to a potential data race.
- `lock_lint` reports on cycles and inconsistent lock order acquisitions. This information can point to potential deadlocks.
- `lock_lint` also reports which lock (or set of locks) were consistently used to protect a variable. This can assist in judging the appropriateness of the chosen *`granularity`*.

`lock_lint` provides a wealth of subcommands for specifying assertions about the application. During the *`analysis`* phase, if `lock_lint` sees that any of these assertions are violated, it immediately reports the transgression. In fact, the recommended method for using `lock_lint` is to use assertions liberally and use the analysis phase to refine those assertions or to make sure that new code does not violate the established locking conventions of the program.

Collecting Information for `lock_lint`

The compiler gathers the information used by `lock_lint`. More specifically, you specify a command line option, `-Zll`, to the C compiler to generate a `.ll` file for your `.c` source code—one `.ll` file per source code file. The `.ll` file contains information about the flow of control in each function, as well as each access to a variable or operation on a mutex or readers/writer lock.

1

`lock_lint` *User Interface*

`lock_lint` uses a command line interface. See Chapter 3, "lock_lint Subcommand Reference" later in this manual.

`lock_lint` subcommands load the relevant `.ll` files. `lock_lint` commands help you analyze your code and discover which variables are not consistently protected by locks. You may make assertions to `lock_lint` about which variables are supposed to be protected by a lock and about which locks are supposed to be held whenever a function is called. Running the analysis with such assertions in place will show you where the assertions are violated.

How to Use `lock_lint` 2

The external interface to `lock_lint` consists of two pieces. Compiling the source code to be analyzed, producing the `lock_lint` database files (.ll) and the `lock_lint` command which is used to control a `lock_lint` session.

Compiling Code

In order for `lock_lint` to analyze your source code, the code must first be compiled using the `-Zll` option of the SunPro C compiler. The SunPro C compiler will then produce the `lock_lint` database files (.ll files), one for each .c file compiled. The .ll files will latter be loaded into `lock_lint` via the `load` subcommand.

`Lock_lint` sometimes needs a simpler view of the code in order to return meaningful results during analysis. To help aid in providing this simpler view, the `-Zll` option automatically defines the macro `__lock_lint`. Further discussions of the likely uses of `__lock_lint` can be found under *limitations* below.

`lock_lint` *Subcommands*

The interface to `lock_lint` consists of a a set of subcommands which can be specified via the `lock_lint` command:

```
lock_lint [subcommand]
```

where *subcommand* is one of a set of subcommands to `lock_lint` which are used to direct the analysis of the source code for data races and deadlocks, and to provide the results of that analysis.

Starting and Exiting `lock_lint`

The very first subcommand of any `lock_lint` session must be the subcommand `start`. The purpose of the `start` subcommand is to start a user-specifiable subshell with the appropriate `lock_lint` context established. Since a `lock_lint` session is started within a subshell it is exited by exiting that subshell. For example, to exit `lock_lint` when using csh, use the command `exit`.

`Lock_lint` can also be invoked from the SPARCworks Manager. When started from the SPARCworks Manager, a command tool will be launched with the command `lock_lint start` using the current SPARCworks context.

Tool State

`Lock_lint`'s *state* consists of the set of databases loaded and the specified assertions. It can be very useful to be able to iteratively modify that state and rerun the analysis so to obtain optimal information on any potential data races and deadlocks that may exist. Since the analysis can only be done once for any particular state, the `save`, `restore` and `refresh` subcommands have been provided as a means to reestablish a state, modify that state and retry the analysis.

Suggested Approach for Checking an Application

Here's a rough outline of what to do:

- Load the .ll files using the `load` subcommand.
- Make assertions about locks protecting functions and variables using the `assert` subcommand.
- Make assertions about the order in which locks should be acquired in order to avoid deadlocks, using the `assert order` subcommand.

- Check that lock_lint has the right idea about which functions are roots. If the `funcs -o` subcommand does not show a root function as root, use the `declare root` subcommand to fix it. If `funcs -o` shows a non-root function as root, it's likely that the function should be listed as a function target using the `declare ... targets` subcommand.
- Describe any hierarchical lock relationships (if you have any—they are rare) using the `rwlock` subcommand.
- Ignore any functions or variables you want to exclude from the analysis using the `ignore` subcommand.
- Run the analysis using the `analyze` subcommand.
- Deal with the errors. This may involve modifying the source using #ifdef __lock_lint (see Appendix , "Limitations of lock_lint"). Restore `lock_lint` to a preanalysis state and rerun the analysis as necessary. Note that it is best to handle the errors in order -- problems with locks not being held on entry to a function, or problems with locks being released while not held, and the like, can cause lots of misleading messages about variables not being properly protected.
- Run the analysis using the analyze -v subcommand and repeat the above step.
- When the errors from the `analyze` subcommand are gone, use the command `lock_lint vars -h | fgrep \*` to see variables which are not properly protected by any lock. Rerun the analysis using appropriate assertions to find out where the variables are being accessed without holding the proper locks.

  Remember that you cannot run `analyze` twice, so it will probably help to save the state of lock_lint using the `save` subcommand before running `analyze`, and then restoring that state using `refresh` or `restore` before adding more assertions, etc. You may want to set up an alias for `analyze` which automatically does a `save` before analyzing.

2

Program Knowledge Management

Loading Sources for Analysis

As stated above, `lock_lint` acquires its information on the sources to be analyzed via a set of databases produced by the C compiler. The `lock_lint` database for each source file is stored in a separate file. In order to analyze a set of source files, use the `load` subcommand to load their associated database files. The `files` subcommand can be used to display a list of the source files represented by the loaded database files. Once a file is loaded, `lock_lint` knows about all the functions, global data, and external functions referenced in the associated source files.

Function Management

As part of the analysis phase, `lock_lint` builds a *call graph* for all the loaded sources. Information about the functions defined is available via the `funcs` subcommand. All functions that are not called by any of the loaded files are called *root* functions. It may be desirable to treat certain functions as *root* functions even though they are called within the loaded modules. This can be accomplished using the `declare root` subcommand. The user may also remove functions from consideration when building the call graph by issuing the `ignore` subcommand.

`lock_lint` also knows about all the references to *function pointers* and most of the assignments made to them. Information about the function pointers in the currently loaded files is available via the `funcptrs` subcommand. Information about the calls made via function pointers is available via the `pointer calls` subcommand. If there are possible function pointer assignments that `lock_lint` could not discover, they may be specified with the `declare ... targets` subcommand.

By default, during analysis, `lock_lint` will try to examine all possible execution paths. If the code uses function pointers it's possible that many of the execution paths will not actually be followed in normal operation of the code. This can result in the reporting of deadlocks that do not really occur. To prevent this, the `lock_lint disallow`, and `reallow` subcommands can be used to inform `lock_lint` of the execution paths that will never occur. To print out existing constraints, use the `reallows` and `disallows` subcommands.

Variable Management

The `lock_lint` database also contains information about all the global variables accessed in the source code. Information about these variables is available via the `vars` subcommands.

As stated above, one of the jobs of `lock_lint` is to determine if variable accesses are consistently protected. If, for some reason, the user wishes to not worry about a particular variable, the accesses of that variable may be removed from consideration by using the `ignore` subcommand.

Lock Management

As we stated above, the most efficient use of `lock_lint` is to refine the assertions the programmer makes about the locks in the code. There are three types of assertions: *protection*, *ordering*, and *side effects*.

Protection assertions state what is protected by a given lock. A variation of the `assert` subcommand is used to assert that a given lock protects some piece of data or a function. The `assert ... covers` subcommand asserts that a given lock protects another lock; this is used for hierarchical locking schemes.

Order assertions specify the order in which the given locks must be acquired. Lock orderings are specified with the `assert order` subcommand.

Side effect assertions state that a function has the side effect of releasing or acquiring a given lock. Use the `assert side effect` subcommand to specify side effects. In some cases, the programmer may want to make side effect assertions about an external function and the lock is not visible from the loaded module (e.g. it is static to the module of the external function). In such a case, the programmer can “create” a lock by using a form of the `declare` subcommand.

Analysis

The primary functionality of `lock_lint` is to report on lock usage inconsistencies which may lead to data races and deadlocks. The analysis of lock usage occurs when the `analyze` subcommand is used. When the `analyze` subcommand is executed, it goes through a series of phases which will report on the following types of problems:

1. Functions which produce side effects on locks or violate assertions made about side effects on locks. An example of a side effect is a function which changes the state of a mutex lock from locked to unlocked. This type of problem may lead to both data races and deadlocks.
2. Violations of assertions about which locks should be held upon entry to a function. This type of problem may lead to a data race.
3. Violations of assertions that a lock should be held when a variable is accessed. This type of problem may lead to a data race.
4. Violations of assertions that specify the order in which locks are to be acquired. This type of problem may lead to a deadlock.
5. Check to make sure the same, or asserted, mutex lock is used for all waits on a particular condition variable.
6. Miscellaneous other problems related to analysis of the source code in relation to assertions and locks.

Post Analysis Queries

After analysis `lock_lint` has several subcommands to aid in finding additional locking inconsistencies and for aiding in the formation of appropriate `declare`, `assert`, `ignore`, etc. subcommands which can be specified after restoring `lock_lint`**'s** state, prior to rerunning the analysis.

1. One such subcommand is the `order` subcommand which can be used to make inquiries about the order in which locks have been acquired. This information is particularly useful in understanding lock ordering problems, and making assertions about those orders so that `lock_lint` can more accurately diagnose potential deadlocks.
2. Another such subcommand is `vars`. The `vars` subcommand gives information such as which locks are consistently held when a variable is read or written, and when there is an absence of any locks consistently protecting a variable. This information can be useful in avoiding data races and making assertions as to how variables are to be protected.

`lock_lint` *Subcommand Reference* 3

`lock_lint` *Subcommands*

The following brief list identifies the `lock_lint` subcommands:

- `analyze` — tests the loaded files for lock inconsistencies.
- `assert` — specifies what `lock_lint` should expect to see regarding accesses and modifications to locks and variables.
- `declare` — passes information to `lock_lint` that it cannot deduce.
- `disallow` — excludes the specified calling sequence in the analysis.
- `disallows` — lists the calling sequences that are excluded from the analysis.
- `files` — lists the source code files loaded via the `load` subcommand.
- `funcptrs` — lists information about function pointers.
- `funcs` — lists information about specific functions.
- `help` — provides information about the specified keyword
- `ignore` — excludes the specified functions and variables from analysis.
- `load` — specifies the `.ll` files to be loaded.
- `locks` — lists information about locks.
- `members` — lists members of the struct specified by the struct tag.
- `order` — shows information about the order in which locks are acquired.

- `pointer calls` — lists calls made through function pointers.
- `reallow` — allows exceptions to the `disallow` subcommand.
- `reallows` — lists the calling sequences reallowed through the `reallow` subcommand.
- `refresh` — restores and resaves the latest saved state.
- `restore` — restores the latest saved state.
- `save` — saves the current state on a stack.
- `saves` — lists the states saved on the stack through the `save` subcommand.
- `start` — starts the `lock_lint` session.
- `sym` — lists the fully-qualified names of functions and variables associated with the specified name.
- `unassert` — removes some assertions specified through the `assert` subcommand.
- `vars` — lists information about variables.

Many `lock_lint` subcommands require you to specify names of locks, variables, pointers, and functions. In C, it is possible for names to be ambiguous. See the discussion of "Names" later in this chapter for details on specifying names to `lock_lint` subcommands.

Table 3-1 lists the exit status of `lock_lint` subcommands:

Table 3-1 Subcommand Exit Status

Value	Meaning
0	Normal
1	System error
2	User error, such as incorrect options or undefined name
3	Multiple errors
5	lock_lint detected error: violation of an assertion, potential data race or deadlock may have been found, unprotected data references, etc.
10	Licensing error

analyze

`analyze` has the following syntax:

```
analyze [-hv]
```

Analyze the loaded files for lock inconsistencies which may lead to data races and deadlocks. This subcommand may produce a great deal of output, so you may want to redirect the output to a file. This subcommand can only be run once for each saved state. (See the `save` subcommand).

-h (history) produce detailed information for each phase of the analysis. No additional errors are issued.

-v (verbose) generate additional messages during analysis:

- Writable variable read while no locks held!
- Variable written while no locks held!
- No lock consistently held while accessing variable!

Use of this option may dramatically increase the amount of output from the analysis, particularly when analyzing code which has not been analyzed before; and for which the `assert read only` subcommand has not been used to identify variables which are read-only; and for which no assertions have been made about how writable variables are protected. The nature of these messages is such that they are likely to reflect situations which are often not real problems; therefore, it is often helpful to first analyze the code without this option, to show just the messages which are more likely to represent real problems.

Each problem encountered by the analysis is reported on one or more lines, the first of which begins with an asterisk. Where possible, a complete traceback of the calls taken to arrive at the point of the problem is given. The analysis goes through the following phases:

Checking for functions with variable side effects on locks

If a *disallow* sequence were to sometimes specify a function with locking side effects to not be analyzed, that would cause bizarre things to happen. Therefore it's not allowed; if such *disallow* sequences are found, they will be reported and analysis will not proceed.

Preparing locks to hold order info

Here `lock_lint` processes the asserted lock order information available to it. If `lock_lint` detects a cycle in the asserted lock order, this gets reported as an error.

Checking for function pointers with no targets

During this phase `lock_lint` will report on any function pointers for which `lock_lint` has not been able to figure out any possible target functions, and for which you have not provided any possible targets using the `declare` *`func_ptr`* `targets` subcommand.

Removing accesses to ignored variables

To improve performance, `lock_lint` removes references to ignored variables at this point. (This affects the output of the `vars` subcommands.)

Preparing functions for analysis

During this phase, `lock_lint` determines what side effects, if any, each function has on locks. (A side effect is a change in a lock's state which is not reversed before returning.) An error results if

- the side effects do not match what `lock_lint` expects
- the side effects are different depending upon the path taken through the function
- a function with such side effects is recursive

`Lock_lint` expects that a function will have no side effects on locks, except where side effects have been added using the `assert side effect` subcommand.

Preparing to recognize calling sequences to allow/disallow

Here `lock_lint` is processing the various `allow/disallow` subcommands that were issued, if any. No errors or warnings are reported here.

Checking locking side effects in function pointer targets

Calls through function pointers may target several functions. All functions which are targets of a particular function pointer must have the same side effects on locks (if any). If a function pointer has targets which differ in their side effects, analysis will not proceed.

Checking for consistent use of locks with condition variables

Here `lock_lint` checks that all waits on a particular condition variable use the same mutex. Also, if you asserted that particular lock to protect that condition variable, `lock_lint` makes sure you used that lock when waiting on the condition variable.

Determining locks consistently held when each function is entered

During this phase, `lock_lint` reports violations of assertions that locks should be held upon entry to a function (see `assert` subcommand). Also, errors such as locking a mutex lock which is already held, or releasing a lock which is not held, are reported. Note that locking an anonymous lock, such as foo::lock, more than once is not considered an error, unless the "declare one" command has been used to indicate otherwise. (see "Names" for details on anonymous data).

Determining locks consistently held when each variable is accessed

During this phase, `lock_lint` reports violations of assertions that a lock should be held when a variable is accessed (see `assert` subcommand). Also, any writes to read-only variables are reported.

It's rare that this would happen, but it is possible during this phase to get messages that certain functions were never called. This occurs if a set of functions call each other, such that all of the functions are called (which means that none are root functions), but none of them is called from outside the set (which means they will never get called at all). The `declare root` subcommand can be used to fix this situation for a subsequent analysis.

Using the `disallow` subcommand to disallow all sequences which reach a function will also cause a message that the function is never called.

Once the analysis is done, you can find still more potential problems in the output of the `vars` and `order` subcommands.

assert

`assert` has the following syntax:

```
assert side effect  mutex          acquired in    func ...
assert side effect  rwlock [read]  acquired in    func ...
assert side effect  lock           released in    func ...
assert side effect  rwlock         upgraded in    func ...
assert side effect  rwlock         downgraded in  func ...
assert mutex|rwlock protects       var ...
assert mutex        protects                          func ...
assert rwlock       protects       [reads in]         func ...
assert order        lock lock ...
assert read only    var ...
assert rwlock       covers         lock ...
```

These subcommands tell `lock_lint` how the programmer expects locks and variables to be accessed and modified in the application being checked. During analysis any violations of such assertions are reported.

Note – If a variable is asserted more than once, only the last `assert` takes effect.

Several of the assertions state that given functions have *side effects* with respect to lock states. A *side effect* is a change made by a function in the state of a lock which is not reversed before the function returns. If a function contains locking side effects and no assertion is made about the side effects of a function, or the side effects differ from those that are asserted, a warning will be issued during the analysis. The analysis will then continue as if the unexpected side effect never occurred. Warnings will also be issued if the side effects produced by a function could differ from call to call (i.e. conditional side effects). The keywords *acquired in*, *released in*, *upgraded in* and *downgraded in*, describe the type of locking side effect being asserted about the function. The keywords correspond to the side effects available via the threads library interfaces and the DDI and DKI Kernel Functions (see `mutex`(3T), `rwlock`(3T), `mutex`(9F) and `rwlock`(9F)). Note that the side effect assertion for rwlocks takes an optional argument `read`; if read is present, the side effect is that the function(s)

acquire(s) read level access for that lock. If the optional keyword is not present, the side effect is specified to be that the function(s) acquire(s) write level access for that lock

Asserting that a mutex lock protects a variable causes `lock_lint` to issue an error during analysis whenever the variable is accessed without holding the mutex lock. Asserting that a readers-writer lock protects a variable causes `lock_lint` to issue an error during analysis whenever the variable is read without holding the lock for read access, or written without holding the lock for write access. Subsequent assertions as to which lock protects a variable override any previously made assertions; that is, only the last lock asserted to protect a variable will be used during analysis.

Asserting that a mutex lock protects a function causes `lock_lint` to issue an error during analysis whenever the function is called without holding the lock. For root functions, the analysis will be performed as if the root function was called with this assertion being true.

Asserting that a readers-writer lock protects a function causes `lock_lint` to issue an error during analysis whenever the function is called without holding the lock for write access. Asserting that a readers-writer lock protects reads in a function causes `lock_lint` to issue an error during analysis whenever the function is called without holding the lock for read access. For root functions, the analysis will be performed as if the root function was called with this assertion being true.

To avoid flooding the output with too many violations of a single `assert... protects` subcommand, a maximum of 20 violations of any given assertion will be shown. This limit does not apply to the `assert order` subcommand.

The `assert order` subcommand may be used to inform `lock_lint` of the order in which locks should be acquired. That is, `lock_lint` assumes that the program avoids deadlocks by adhering to a well-known lock order. Using this subcommand one can make `lock_lint` aware of the intended order, so that violations of the order can be printed during analysis.

The `assert read only` subcommand states that the given set of variables are expected to never be written by the application; `lock_lint` reports any writes to the variables. Note that unless a variable is read-only, reading the variable while no locks are held will elicit an error, since `lock_lint` assumes that the variable could be written by another thread at the same time.

The `assert` *`rwlock`* `covers` subcommand informs `lock_lint` of the existence of a hierarchical locking relationship. A readers-writer lock may be used in conjunction with other locks (mutex or readers-writer) in the following way to increase performance in certain situations:

- A certain readers-writer lock, called the "cover", must be held while any of a set of other "covered" locks is held. That is, it is illegal (under these conventions) to hold a covered lock while not also holding the cover, at least for read access.
- While holding the cover for write access, any variable protected by one of the covered locks may be accessed without holding the covered lock. This works because it is impossible for another thread to hold the covered lock, since it would also have to be holding the cover. The time saved by not locking the covered locks can increase performance if there is not excessive contention over the cover.

Describing this relationship to `lock_lint` using this "`assert` *`rwlock`* `covers`" subcommand prevents error messages from being issued when variables are accessed while holding the cover for write access but not the covered lock. It also enables checks that a covered lock is never held when its cover is not.

`declare`

`declare` has the following syntax:

```
declare   mutex         mutex ...
declare   rwlocks       rwlock ...
declare   func_ptr      targets     func ...
declare   nonreturning  func ...
declare   one           tag ...
declare   readable      var ...
declare   root          func ...
```

These subcommands are used to tell `lock_lint` things it can't deduce from the source presented it.

The `declare mutex, rwlocks, and root` subcommands are typically used when analyzing libraries without a supporting harness. The subcommands `declare mutex` and `declare rwlocks` create mutex and reader-writer locks of the given names. These symbols can be used in subsequent `assert` subcommands.

The `declare` *func_ptr* `targets` adds the specified functions to the list of functions which could be called when a call is made through the specified function pointer. Note that `lock_lint` manages to gather a good deal of information about function pointer targets on its own by watching initializations and assignments. For example, for the code

```
struct foo { int (*fp)(); } foo1 = { bar };
```

`lock_lint` does the equivalent of the command

```
lock_lint declare foo::fp targets  bar
```

Note – `lock_lint` does not yet do the following (for the above example):

```
lock_lint declare foo1.fp targets  bar
```

However, it does manage to do both for assignments to function pointers. See “Names”.

The `declare nonreturning` subcommand tells `lock_lint` that the specified functions do not return. Armed with this knowledge, `lock_lint` will not give errors about lock state after calls to such functions.

The `declare one` subcommand tells `lock_lint` that only one unnamed instance exists of each structure whose tag is specified. This knowledge makes it possible for `lock_lint` to give an error if a lock in that structure is acquired multiple times without being released. Without this knowledge, `lock_lint` does not complain about multiple acquisitions of anonymous locks (e.g. foo::lock), since two different instances of the structure could be involved. (See “help names”).

The `declare readable` subcommand tells `lock_lint` that the specified variables may be safely read without holding any lock, thus suppressing the errors which would ordinarily occur for such unprotected reads.

The `declare root` subcommand is used to tell `lock_lint` that it should try to analyze the given functions as a root function; by default, if a function is called from any other function, `lock_lint` will not attempt to analysis that function as the root of a calling sequence. A "root" function is a starting point for the analysis; functions which are not called from within the loaded files are naturally roots. For example, functions which are never called directly but are the initial starting point of a thread, i.e. the target function of a thread_create call. However, a function which *is* called from within the loaded files might also be called from outside the loaded files, in which case you should use this subcommand to cause `lock_lint` to also use the function as a starting point in the analysis.

disallow

`disallow` has the following syntax:

```
disallow func ...
```

This subcommand tells `lock_lint` that the specified calling sequence should not be analyzed. For example, to prevent `lock_lint` from analyzing any calling sequence in which f() calls g() calls h(), use the subcommand

```
lock_lint disallow f g h
```

The reason you might want to use this subcommand is that function pointers can make a program appear to follow many calling sequences which do not in practice occur, and bogus locking problems (particularly deadlocks) can appear in such sequences. (See also the `reallow` subcommand.)

disallows

`disallows` has the following syntax:

```
disallows
```

Lists the calling sequences which are disallowed, as specified using the `disallow` subcommand.

exit

There is *no* exit subcommand for `lock_lint`. To exit `lock_lint` use the `exit` command of the shell you are using. When the shell exits, `lock_lint start` exits. (See the `start` subcommand.)

files

List the source code files as represented by the .ll files loaded via the `load` subcommand.

funcptrs

`funcptrs` has the following syntax:

```
funcptrs [-botu] func_ptr ...
funcptrs [-blotuz]
```

List information about the function pointers used in the loaded files. One line is produced for each function pointer.

-b (bound) list only function pointers to which function targets have been bound. That is, suppress the display of function pointers for which there are no bound targets.

-l (long) is equivalent to -ot.

-o (other) presents the following information about each function pointer:

Calls=# indicates the number of places in the loaded files this function pointer is used to call a function.

=nonreturning indicates that a call through this function pointer never returns (none of the functions targeted ever return).

-t (targets) lists the functions currently bound as targets to each function pointer listed, as follows:

```
targets={ func ... }
```

-u (unbound) list only those function pointers to which no function targets are bound. That is, suppress the display of function pointers for which there are bound targets.

-z (zero) list function pointers for which there are no calls. Without this option information is given only on function pointers through which calls are made.

```
funcptrs [-botu] func_ptr ...
```

List information about the specified function pointers. By default, this variant of the subcommand gives all the details about the function pointers, as if -ot had been specified.

```
funcptrs [-blotuz]
```

List information about all function pointers through which calls are made. If -z is used, even function pointers through which no calls are made are listed.

funcs

`funcs` has the following syntax:

```
funcs [-adehou]                          func ...
funcs [-adehilou]
funcs [-adehlou] [directly] called by    func ...
funcs [-adehlou] [directly] calling      func ...
funcs [-adehlou] [directly] reading      var ...
funcs [-adehlou] [directly] writing      var ...
funcs [-adehlou] [directly] accessing    var ...
funcs [-adehlou] [directly] affecting    lock ...
funcs [-adehlou] [directly] inverting    lock ...
```

List information about the functions defined and/or called in the loaded files. Exactly one line is printed for each function.

-a (asserts) show information about which locks are supposed to be held on entry to each function, as set by the `assert` subcommand. When such assertions have been made, they show as:

```
            asserts={ lock ... }
       read_asserts={ lock ... }
```

An asterisk will appear before the name of any lock which was not consistently held upon entry (after analysis).

-e (effects) show information about the side effects each function has on locks (e.g. “acquires mutex lock foo”). If a function has such side effects, they are shown as:

```
side_effects={ effect [, effect] ... }
```

Using this option prior to analysis shows `side_effects` asserted via an `assert side effect` subcommand. After analysis information on `side_effects` discovered and during the analysis is also shown.

-d (defined) show only those functions which are *defined* in the loaded files. That is, suppress the display of undefined functions.

-h (held) show information about which locks were consistently held when the function was called (after analysis). Locks consistently held for read (or write) on entry show as:

```
        held={ lock ... }+{ lock ... }
   read_held={ lock ... }+{ lock ... }
```

The first list in each set is the list of locks *consistently* held when the function was called; the second is a list of *inconsistently* held locks—locks which were sometimes held when the function was called, but not every time.

-i (ignored) list ignored functions.

-l (long) is equivalent to -aeoh.

-o (other) where applicable, present the following information about each function:

`=ignored`	indicates that `lock_lint` has been told to ignore the function using the `ignore` subcommand.
=nonreturning	indicates that a call through this function pointer never returns (none of the functions targeted ever return).
`=rooted`	indicates that the function was made a root using the `declare root` subcommand.
`=root`	indicates that the function is naturally a root (is not called by any function).
`=recursive`	indicates that the function makes a call to itself.
`=unanalyzed`	indicates that the function was never called during analysis (and is therefore unanalyzed). This differs from `=root` in that this can happen when `foo` calls `bar` and `bar` calls `foo`, and no other function calls either `foo` or `bar`, and neither have been rooted (see `=rooted`). So, because `foo` and `bar` are not roots, and they can never be reached from any root function, they have not been analyzed.

`calls=#` indicates the number of places in the source code, as represented by the loaded files, where this function is called. Note that these calls may not actually be analyzed; for example, a `disallow` subcommand may prevent a call from ever really taking place.

-u (undefined) show only those functions which are *undefined* in the loaded files. That is, suppress the display of defined functions.

```
funcs [-adehou]   func ...
```

List information about individual functions. By default, this variant of the subcommand gives all the details about the functions, as if -aeho had been specified.

```
funcs [-adehilou]
```

List information about all unignored functions. If -i is used, even ignored functions are listed.

```
funcs [-adehlou] [directly] called by func ...
```

List only those functions which may be called as a result of calling the specified functions. If `directly` is used, only those functions called by the specified functions will be listed. If `directly` is not used, any functions *those* functions called will also be listed, and so on.

```
funcs [-adehlou] [directly] calling  func ...
```

List only those functions which, when called, may result in one or more of the specified functions being called. See notes below on `directly`.

```
funcs [-adehlou] [directly] reading  var ...
```

List only those functions which, when called, may result in one or more of the specified variables being read. See notes below on `directly`.

```
funcs [-adehlou] [directly] writing  var ...
```

List only those functions which, when called, may result in one or more of the specified variables being written. See notes below on `directly`.

```
funcs [-adehlou] [directly] accessing var ...
```

List only those functions which, when called, may result in one or more of the specified variables being accessed (read or written). See notes below on `directly`.

```
funcs [-adehlou] [directly] affecting lock ...
```

List only those functions which, when called, may result in one or more of the specified locks being affected (acquired, released, upgraded, or downgraded). See notes below on `directly`.

```
funcs [-adehlou] [directly] inverting lock ...
```

List only those functions which invert one or more of the specified locks. (See "Inversions"). If `directly` is used, only those functions which themselves invert one or more of the locks (actually release them) are listed. If `directly` is not used, any function which is called with a lock already held, and then calls another function which inverts the lock, is also listed, and so on. For example, in the following code, f3() directly inverts lock m, and f2() indirectly inverts it:

```
f1() { mutex_unlock(&m); f2(); mutex_lock(&m); }
f2() { f3(); }
f3() { mutex_unlock(&m); mutex_lock(&m); }
```

Except where stated otherwise, those variants which allow the keyword `directly` will, when `directly` is used, only list the functions which *themselves* fit the description. If `directly` is not used, all the functions which call those functions will be listed, and any functions which call *those* functions, and so on.

help

`help` has the following syntax:

```
help [keyword]
```

Without a keyword, `help` displays the subcommand set.

With a keyword, `help` gives helpful information relating to the specified keyword. The keyword may be the first word of any `lock_lint` subcommand. There are also a few miscellaneous keywords for which help is available:

```
condvars
example
ifdef
inversions
limitations
locking
makefile
names
overview
shell
```

If environment variable `PAGER` is set, that program will be used as the pager for `help`. If `PAGER` is not set, `more` is used.

ignore

`ignore` has the following syntax:

```
ignore func|var ... [ in func ... ]
```

This subcommand tells `lock_lint` to exclude certain functions and variables from the analysis. This exclusion may be limited to specific functions using the `in` *`func`* `...` clause; otherwise the exclusion applies to all functions.

The commands

```
lock_lint funcs -io | grep =ignored
lock_lint vars -io | grep =ignored
```

will show which functions and variables will be ignored.

load

`load` has the following syntax:

```
load file ...
```

Load the specified .ll files. The extension may be omitted, but if an extension is specified, it must be .ll. Absolute and relative paths are allowed. Don't forget that you are talking to a shell, so the following are perfectly legal (depending upon your shell's capabilities):

```
lock_lint load *.ll
lock_lint load ../foo/abcdef{1,2}
lock_lint load `find . -name \*.ll -print`
```

locks

`locks` has the following syntax:

```
locks [-co] lock ...
locks [-col]
locks [-col] [directly] affected by func ...
locks [-col] [directly] inverted by func ...
```

List information about the locks of the loaded files. Note that only those variables which are actually *used* in lock manipulation routines are shown; locks which are simply declared but never manipulated will not be shown.

-c (cover) show information about lock hierarchies. Such relationships are described using the `assert` *rwlock* `covers` subcommand. (When locks are arranged in such a hierarchy, the covering lock must be held, at least for read access, whenever any of the covered locks is held. While holding the covering lock for write access, it is unnecessary to acquire any of the covered locks.) If a lock covers other locks, those locks show as:

```
covered={ lock ... }
```

If a lock is covered by another lock, the covering lock shows as

```
cover=lock
```

-l (long) is equivalent to -co.

-o (other) causes the type of the lock to be shown as (<type>) where <type> is *mutex*, *rwlock*, or *ambiguous type* (used as a mutex in some places and as an rwlock (readers-writer) in other places).

```
locks [-co] lock ...
```

List information about individual locks. By default, this variant of the subcommand gives all the details about the locks, as if -co had been specified.

```
locks [-col]
```

List information about all locks.

```
locks [-col] [directly] affected by func ...
```

List only those locks which may be affected (acquired, released, upgraded, or downgraded) as a result of calling the specified functions. If the keyword `directly` is used, only functions which use the threads library routines directly to affect a lock (acquire, release, upgrade, or downgrade) are listed. If the keyword `directly` is not used, any function which calls a function which affects a lock will be listed, and any function calling that function will be listed, and so on.

```
locks [-col] [directly] inverted by func ...
```

List only those locks which may be inverted by calling one of the specified functions. See “help inversions”. If the keyword “directly” is used, only those locks which are directly inverted by the specified functions (that is, these functions actually release and reacquire the locks using a threads library routine) are listed. If the keyword “directly” is not used, a lock which is held by one of the specified functions and inverted by some function called from it (etc.) is also listed. For example, in the following code f1 directly inverts m1, and indirectly inverts m2.

```
f1() { mutex_unlock(&m1); f2(); mutex_lock(&m1); }
f2() { f3(); }
f3() { mutex_unlock(&m2); mutex_lock(&m2); }
```

members

`members` has the following syntax:

```
members struct_tag
```

List the members of the struct with the specified tag, one per line. For structures which were not assigned a tag, the notation “*file@line*” is used (e.g. `x.c@29`), where the file and line number are the source location of the struct’s declaration.

`members` is particularly useful to use as input to other `lock_lint` subcommands. For example, when trying to assert that a lock protects all the members of a struct the following command suffices:

```
lock_lint assert foo::lock protects `lock_lint members foo`
```

Note – The `members` subcommand does not list any fields of the struct which are defined to be of type: mutex_t, rwlock_t, krwlock_t, or kmutex_t.

order

`order` has the following syntax:

```
order [lock [lock]]
order summary
```

The `order` subcommand lists information about the order in which locks are acquired by the code being analyzed. It may only be run after the `analyze` subcommand.

The first form of the subcommand shows the details about lock pairs. For example, the command

```
lock_lint order foo bar
```

will show whether an attempt was ever made to acquire lock `bar` while holding lock `foo`. The output will look something like the following:

```
:foo :bar seen (first never write-held), valid
```

First the output tells whether such an attempt actually occurred (`seen` or `unseen`). If the attempt occurred, but never with one or both of the locks write-held, a parenthetical message to that effect appears, as in the example. In this case, `foo` was never write-held while acquiring `bar`.

If an assertion was made about the lock order, the output shows whether the specified order is `valid` or `invalid` according to the assertion. If there was no assertion about the order of `foo` and `bar`, or if both orders were asserted (presumably because the user wanted to see all places where one of the locks was held while acquiring the other), the output will say neither `valid` nor `invalid`.

The second form of the subcommand, `order summary`, shows in a concise format the order in which locks are acquired. For example, the subcommand might show

```
:f :e :d :g :a
:f :c :g :a
```

In this example, there are two orderings because there is not enough information to allow locks `e` and `d` to be ordered with respect to lock `c`.

pointer calls

`pointer calls` has the following syntax:

```
pointer calls
```

List calls made through function pointers in the loaded files. Each call is shown as:

function [*location of call*] `calls through funcptr` *func_ptr*

For example,

```
foo.c:func1 [foo.c,84] calls through funcptr bar::read
```

which means at line 84 of foo.c, which is in func1 of foo.c, the function pointer bar::read (member read of a pointer to struct of type bar) is used to call a function.

reallow

`reallow` has the following syntax:

```
reallow func ...
```

This subcommand allows one to make exceptions to `disallow` subcommands. For example, to prevent `lock_lint` from analyzing any calling sequence in which `f()` calls `g()` calls `h()`, except when `f()` is called by `e()` which was called by `d()`, use the commands

```
lock_lint disallow f g h
lock_lint reallow d e f g h
```

In some cases you might want to state that a function should only be called from a particular function, as in this example

```
lock_lint disallow f
lock_lint reallow e f
```

Note – A `reallow` subcommand only suppresses the effect of a `disallow` subcommand if the sequences end the same.

For example, after the following commands, the sequence `d e f g h` would still be disallowed:

```
lock_lint disallow e f g h
lock_lint reallow d e f g
```

reallows

`reallows` has the following syntax:

```
reallows
```

Lists the calling sequences which are reallowed, as specified using the `reallow` subcommand.

`refresh`

`refresh` has the following syntax:

```
refresh
```

Pop the saved state stack restoring `lock_lint` to the state of the top of the saved state stack. The description associated with that state is printed. Then immediately resave (push) the state again (see the restore and save subcommands).

`restore`

`restore` has the following syntax:

```
restore
```

Pop the saved state stack restoring `lock_lint` to the state of the top of the saved state stack. The description associated with that state is printed.

The saved state stack is a LIFO (Last-In-First-Out) stack. Once a saved state is restored (popped) from the stack that state is no longer on the saved state stack. If the state needs to be saved and restored repeatedly simply save the state again immediately after restoring it, or use the `refresh` subcommand.

`save`

`save` has the following syntax:

```
save description
```

Save the current state of the tool on a stack. The specified description is attached to the state. Saved states form a LIFO (Last-In-First-Out) stack, so that the last state saved is the first one restored.

A common use of this subcommand is to save the state of the tool before running the `analyze` subcommand, which may only be run once on a given state. For example, you might do the following:

```
%: lock_lint load *.ll
%: lock_lint save Before Analysis
%: lock_lint analyze
    <output from analyze>
%: lock_lint vars -h | grep \*
    <apparent members of struct foo are not consistently protected>
%: lock_lint refresh
Before Analysis
%: lock_lint assert lock1 protects `lock_lint members foo`
%: lock_lint analyze
    <output now contains info about where the assertion is violated>
```

saves

`saves` has the following syntax:

```
saves
```

List the descriptions of the states saved on the saved stack via the `save` subcommand. The descriptions are shown from top to bottom with the first description being the most recently saved state that has not been restored. And the last description being the oldest state saved that has not been restored.

start

`start` has the following syntax:

```
start [cmd]
```

This subcommand starts a `lock_lint` session. A `lock_lint` session must be started prior to using any other `lock_lint` subcommand. By default, the `start` subcommand establishes `lock_lint`'s context and starts a subshell for the user, as specified via $SHELL, within that context. The only piece of the

`lock_lint` context exported to the shell is the environment variable LL_CONTEXT. LL_CONTEXT contains the path to the temporary directory of file used to maintain a `lock_lint` session.

cmd specifies a command and its path and options. By default, if *cmd* is not specified, the value of $SHELL is used.

Note – To exit a `lock_lint` session use the exit command of the shell you are using.

The following discussion shows variations of the `start` subcommand.

Examples:

```
lock_lint start
```

`Lock_lint`**'s** context is established and LL_CONTEXT is set. Then the program identified by $SHELL is executed. Normally, this will be the user's default shell. `Lock_lint` subcommands can now be entered. Upon exiting the shell, the `lock_lint` context will be removed and the `lock_lint start` subcommand will complete.

```
lock_lint start /bin/csh -c foo
```

With this variant, the `lock_lint` context is established and LL_CONTEXT is set. Then, the command /bin/csh -c foo is executed. This results in executing the csh command file foo, which could contain `lock_lint` commands. Upon completing the execution of the commands in foo by /bin/csh, the `lock_lint` context will be removed and the `lock_lint` start command will complete.

```
lock_lint start foo
```

If you use a shell script to start `lock_lint`, insert `#!` in the first line of the script to define the name of the interpreter that processes that script. For example, to specify the C-shell the first line of the script could be:

```
#! /bin/csh

lock_lint start /bin/ksh
```

After establishing the `lock_lint` context and setting LL_CONTEXT, the command /bin/ksh is executed. This results in the user interacting with an interactive korn shell. Upon exiting the korn shell, the `lock_lint` context will be removed and the `lock_lint start` command will complete.

sym

`sym` has the following syntax:

```
sym name ...
```

Lists the fully-qualified names of various things the specified name(s) could refer to within the loaded files. For example, `foo` might refer both to variable `x.c:func1/foo` and to function `y.c:foo`, depending on context.

unassert

`unassert` has the following syntax:

```
unassert vars var ...
```

Undo any assertion about locks protecting the specified variables. Note that there is no way to remove an assertion about a lock protecting a function.

3

vars

`vars` has the following syntax:

```
vars [-aho] var ...
vars [-ahilo]
vars [-ahlo] protected by lock
vars [-ahlo] [directly] read by       func ...
vars [-ahlo] [directly] written by    func ...
vars [-ahlo] [directly] accessed by   func ...
```

List information about the variables of the loaded files. Note that only those variables which are actually *used* are shown; variables which are simply declared in the program but never accessed will not be shown.

-a (assert) show information about which lock is supposed to protect each variable, as specified by the `assert` *mutex*|*rwlock* `protects` subcommand. The information is shown as follows:

```
assert=lock
```

If the assertion is violated, then after analysis this will be preceded by an asterisk, i.e. `*assert=<lock>`.

-h (held) show information about which locks were consistently held when the variable was accessed. This information will only be shown after the `analyze` subcommand has been run. If the variable was never accessed, this information will not be shown. When it is shown, it looks like this:

```
held={ <lock> ... }
```

If no locks were consistently held and the variable was written, this will be preceded by an asterisk, i.e. `*held={ }`.

Unlike the `funcs` subcommand, the `vars` subcommand lists a lock as protecting a variable even if the lock was not actually held, but was simply covered by another lock (see `assert` *rwlock* `covers`).

-i (ignored) causes even *ignored* variables to be listed.

-l (long) is equivalent to -aho.

-o (other) where applicable, show the following information about each variable:

`=cond_var`	indicates that this variable is used as a condition variable.
`=ignored`	indicates that `lock_lint` has been told to ignore the variable explicitly via an `ignore` subcommand.
`=read-only`	means that `lock_lint` has been told (by `assert read only`) that the variable is read-only, and will complain if it is written. If it is written, then after analysis this will be followed by an asterisk, i.e. `=read-only*`.
`=readable`	indicates that `lock_lint` has been told by a `declare readable` subcommand that the variable may be safely read without holding a lock.
`=unwritten`	may appear after analysis, meaning that while the variable was not declared read-only, it was never written.

```
vars [-aho] var ...
```

List information about individual variables. By default, this variant of the subcommand gives all the details about the variables, as if -aho had been specified.

```
vars [-ahilo]
```

List information about all unignored variables. If -i is used, even ignored variables are listed.

```
vars [-ahlo] protected by lock
```

List only those variables which are protected by the specified lock. This subcommand may only be run after the `analyze` subcommand has been run.

```
vars [-ahlo] [directly] read by func ...
```

List only those variables which may be read as a result of calling the specified functions. See notes below on *directly*.

```
vars [-ahlo] [directly] written by func ...
```

List only those variables which may be written as a result of calling the specified functions. See notes below on *directly*.

```
vars [-ahlo] [directly] accessed by func ...
```

List only those variables which may be accessed (read or written) as a result of calling the specified functions.

Those variants which list the variables accessed by a list of functions can be told to list only those variables which are `directly` accessed by the specified functions. Otherwise the variables accessed by those functions, the functions they call, the functions *those* functions call, and so on, will be listed.

Names

Many `lock_lint` subcommands require you to specify names of locks, variables, pointers, and functions. In C, it is possible for names to be ambiguous -- for example, there may be several variables named foo, one of them extern and others static. The C language does not provide a way of referring to ambiguously named variables which are hidden by the scoping rules; in `lock_lint`, however, a way of referring to such variables is needed. Therefore every symbol in the code being analyzed is given a formal name, which is the name `lock_lint` uses when referring to the symbol. For example, the formal name for a function is as follows:

```
:func             an extern function
file:func         a static function
```

The formal name for a variable (whether it is being used as a lock, a pointer, or an actual variable) is:

```
:var              an extern variable
file:var          a static variable with file scope
:func/var         a variable defined in an extern function
```

```
file:func/var     a variable defined in a static function
tag::mbr          a member of an unnamed struct
file@line::mbr    a member of an unnamed, untagged struct
```

In addition, any of these may be followed by an arbitrary number of

```
.mbr
```

specifications to denote members of a structure.

Some examples may help illustrate all of this:

"`:bar`" refers to external variable or function "`bar`".

"`:main/bar`" refers to static variable "`bar`" which is defined within extern function "`main`".

"`zot.c:foo/bar.zot`" refers to member "`zot`" of static variable "`bar`", which is defined within static function "`foo`" in file "`zot.c`".

"`foo::bar.zot.bim`" refers to member "`bim`" of member "`zot`" of member "`bar`" of a struct with tag "`foo`", where no name is associated with that instance of the struct (it was accessed through a pointer).

While `lock_lint` refers to symbols in this way, you are not required to. You may use as little of the name as is required to unambiguously identify it. For example, you could refer to "`zot.c:foo/bar`" as "`zot.c:foo/bar`", or "`foo/bar`" as long as there is only one function "`foo`" defining a variable "`bar`", or "`bar`" as long as there is only one variable by that name.

C allows the programmer to declare a structure without assigning it a tag. When you use a pointer to such a structure, `lock_lint` must make up a tag by which to refer to the structure. The name it generates is "`filename@line_number`". For example, if you declare a structure without a tag at line 42 of file "`foo.c`", and then refer to member "`bar`" of an instance of that structure using a pointer, as in

```
typedef struct { ... } FOO;
FOO *p;
func1() { p->bar = 0; }
```

`lock_lint` sees that as a reference to "`foo.c@42::bar`".

Because members of a union share the same memory location, `lock_lint` treats all members of a union as the same variable. This is accomplished by using a member name of "%" regardless of which member is accessed. Since bit fields typically involve sharing of memory between variables, they are handled similarly: "%" is used in place of the bit field member name.

When you list locks and variables, you are only seeing those locks and variables which are actually used within the code represented by the .ll files. No information is available via `lock_lint` on locks, variables, pointers, and functions declared but not used. No information is available for accesses through pointers to simple types, such as this one:

```
int *ip = &i;
*ip = 0;
```

When simple names (e.g. "`foo`") are used, there is the possibility of conflict with keywords in the subcommand language. Such conflicts can be resolved by surrounding the word with double quotes, but remember that you are typing commands to a shell, and shells typically consume the outermost layer of quotes. Therefore you have to escape the quotes, as in this example:

```
lock_lint ignore foo in func \"func\"
```

If two files with the same basename are included in an analysis, and these two files contain static variables by the same name, confusion can result (`lock_lint` will think they are the same variable).

If the definition for a struct with no tag is duplicated, `lock_lint` will not recognize it as the same struct. The problem is that `lock_lint` makes up a tag based on the file and line number where the struct is defined (like `x.c@24`), and that tag will be different for the two copies of the definition.

If a function contains multiple automatic variables by the same name, `lock_lint` cannot tell them apart. Since `lock_lint` ignores automatic variables except when they are used as function pointers, this does not come up often. In the following code, for example, `lock_lint` uses the name `:foo/fp` for both function pointers:

```
int foo(void (*fp)()) {
    (*fp)();
    {
        void (*fp)() = get_func();
        (*fp)();
         ...
```

Inversions

A function is said to “invert” a lock if the lock is already held when the function is called, and the function releases the lock, such as:

```
foo() {
    mutex_unlock(&mtx);
    ...
    mutex_lock(&mtx);
}
```

Lock inversions are a potential source of insidious race conditions, since observations made under the protection of a lock may be invalidated by the inversion. In the following example, if `foo`() inverts `mtx`, then upon its return `zort_list` may be NULL (another thread may have emptied the list while the lock was dropped):

```
ZORT* zort_list;
    /* VARIABLES PROTECTED BY mtx: zort_list */

void f() {
    mutex_lock(&mtx);
    if (zort_list == NULL) /* trying to be careful here */
    return;
    foo();
    zort_list->count++; /* but zort_list may be NULL here!! */
    mutex_unlock(&mtx);
}
```

Lock inversions may be found using the commands

```
lock_lint funcs [directly] inverting <lock> ...
lock_lint locks [directly] inverted by <func> ...
```

An interesting question to ask is “Which functions acquire locks which then get inverted by calls they make?” That is, which functions are in danger of having stale data? The following (Bourne shell) code can answer this question:

```
LOCKS=`locks‘
lock_lint funcs calling `lock_lint funcs inverting $LOCKS`
```

The following gives similar output, separated by lock:

```
for lock in `locks`
do
    echo “functions endangered by inversions of lock $lock”
    lock_lint funcs calling `lock_lint funcs inverting $lock`
done
```

3

3

Limitations of `lock_lint` A

Limitations

`Lock_lint` has trouble with certain types of things, and tries to do the best it can. At the root of many of these problems is the fact that `lock_lint` doesn't know the values of your variables. `Lock_lint` solves some of these problems by ignoring the problematic cause, or making simplifying assumptions. Some other problems can be avoided by using conditionally compiled code in the application. Towards this end, the compiler always defines the preprocessor macro `__lock_lint` when the `-Zll` switch is used. `Lock_lint` will have trouble deducing:

- which functions your function pointers point to. There are some assignments `lock_lint` cannot deduce (see the `declare` subcommand). The `declare` subcommand can be used to add new possible assignments to the function pointer.

  When `lock_lint` sees a call through a function pointer, it will test that call path for every possible value of that function pointer. If some calling sequences will never be executed, use the `disallow` and `reallow` to specify which ones will be.

- whether or not you locked a lock in code like this:

```
if (x) mutex_lock(&lock1);
```

In this case, two execution paths are created, one holding the lock, and one not holding the lock, which will probably cause the generation of a side effect message at the unlock call. The user may be able to work around this problem by using the __lock_lint macro to use an unconditional. For example

```
#ifdef __lock_lint
mutex_lock(&lock1);
#else
if (x) mutex_lock(&lock1);
#endif
```

Note lock_lint has no problem analyzing code like this:

```
if (x) {
    mutex_lock(&lock1);
    foo();
    mutex_unlock(&lock1);
}
```

In this case, there is only one execution path, along which the lock is held and released, causing no side effects.

- whether or not the lock was acquired in code like this:

```
rc = mutex_trylock(&lock1);
if (rc) ...
```

- which lock is being locked in code like this:

```
mutex_t* lockp;
mutex_lock(lockp);
```

  In such cases, the lock call is ignored.

- which variables and locks are being used in code like this (so it uses the names foo::lock and foo::bar -- see the discussion of "Names" in Chapter 3, "lock_lint Subcommand Reference")

```
struct foo* p;
mutex_lock(p->lock);
p->bar = 0;
```

- which element of an array is being accessed. This is treated analogously to the previous case; the index is ignored.
- anything about longjmps
- when you would exit a loop, or break out of a recursion (so it just stops proceeding down a path as soon as it finds itself looping or after one recursion).

Other types of things `lock_lint` has trouble with include:

- `Lock_lint` only analyzes the use of mutex locks and readers-writer locks. `lock_lint` performs limited consitency checks of mutex locks as used with condition variables. However, semaphores and condition variables are not recognized as locks by `lock_lint`. Even with this analysis, there are limits to what `lock_lint` can make sense of.
- There are situations where `lock_lint` thinks two different variables are the same variable, or that a single variable is two different variables. (See the discussion of "Names" in Chapter 3, "lock_lint Subcommand Reference.")
- It is possible to share automatic variables between threads (via pointers), but `lock_lint` assumes that automatics are unshared, and generally ignores them (the only situation in which they are of interest to `lock_lint` is when they are function pointers).

- `Lock_lint` complains about any functions that are not consistent in their side effects on locks. #ifdef's and assertions must be used to give `lock_lint` a simpler view of functions which may or may not have such a side effect.

During analysis, `lock_lint` may produce messages about a lock operation called "rw_upgrade". Such a call does not really exist, but `lock_lint` rewrites code like

```
if (rw_tryupgrade(&lock1)) { ... }
```

as

```
if () { rw_tryupgrade(&lock1); ... }
```

such that, wherever rw_tryupgrade() occurs `lock_lint` always assumes it succeeds.

One of the errors `lock_lint` flags is an attempt to acquire a lock which is already held. However, if the lock is unnamed (e.g. foo::lock), this error is suppressed, since the name refers not to a single lock but to a set of locks. However, if the unnamed lock always refers to the same lock, use the `declare one` subcommand so that `lock_lint` can report this type of potential deadlock.

If you have constructed your own locks out of these locks (for example, recursive mutexes are sometimes built from ordinary mutexes), `lock_lint` will not know about them. Generally you can use #ifdef to make it appear to

`lock_lint` as though an ordinary mutex is being manipulated. For recursive locks, use an unnamed lock for this deception, since errors won't be generated when it is recursively locked. For example,

```
void get_lock() {
     #ifdef __lock_lint
           struct bogus *p;
           mutex_lock(p->lock);
     #else
           <the real recursive locking code>
     #endif
}
```

A

Managing the Interface Environment

Shell Interface

The `lock_lint` interface is provided through the `lock_lint` command which is executed in a shell. By default, `lock_lint` uses $SHELL. Alternatively, `lock_lint` can execute any shell by specifying the shell to use on the `lock_lint start` command. For example,

```
lock_lint start /bin/ksh
```

might start a `lock_lint` session up in the Korn shell.

`Lock_lint` creates an environment variable called LL_CONTEXT which is visible in the child shell. If you are using a shell which provides for initialization, you can arrange to have `lock_lint` source a .ll_init file in your home directory, and then execute a .ll_init file in the current directory if it exists. If you use csh, you can do this by inserting the following code into your .cshrc file:

```
if ($?LL_CONTEXT) then
    if ( -x $(HOME)/.ll_init ) source $(HOME)/.ll_init
endif
```

Note that it is better *not* to have your .cshrc `source` the file in your current working directory, since others may want to run `lock_lint` on those same files, and they may not use the same shell you do. Since you are the only one who is going to use your $(HOME)/.ll_init, you *should* source that one, so that you can change the prompt and define aliases for use during your `lock_lint` session. The following version of ~/.ll_init does this for csh:

```
# Cause analyze subcommand to save state before analysis.
alias analyze "lock_lint save before analyze;\
    lock_lint analyze"
# Change prompt to show we are in lock_lint.
set prompt="lock_lint~$prompt"
```

When executing subcommands, remember that you can use pipes, redirection, backticks, and so on to accomplish your aims. For example, the following command asserts that lock "foo" protects all global variables (the formal name for a global variable begins with a colon):

```
lock_lint assert foo protects `lock_lint vars | grep ^:`
```

In general, the subcommands are set up for easy use with filters such as grep and sed. This is particularly true for `vars` and `funcs`, which put out a single line of information, for each variable or function. Each line contains the attributes (define and derived) for that variable or function. The following example shows which members of struct bar are supposed to be protected by member "lock":

```
lock_lint vars -a `lock_lint members bar` | grep =bar::lock
```

Since we are using a shell interface, a log of user commands can be obtained by using the shell's history function (the history level may need to be made large in the `.ll_init` file).

Temporary Files

`Lock_lint` puts temporary files in /var/tmp unless $TMPDIR is set.

Makefile Rules

To modify your makefile to produce .ll files, first use the rule for creating a .o from a .c to write a rule to create a .ll from a .c. For example, from

```
# Rule for making .o from .c in ../src.
%.o: ../src/%.c
    $(COMPILE.c) -o $@ $<
```

you might write:

```
# Rule for making .ll from .c in ../src.
%.ll: ../src/%.c
    cc $(CFLAGS) $(CPPFLAGS) $(FOO) $<
```

If you use a suffix rule, you'll need to define .ll as a suffix. For that reason some prefer to use % rules.

If the appropriate .o files are contained in a make variable FOO_OBJS, you can create FOO_LLS with the line:

```
FOO_LLS = ${FOO_OBJS:%.o=%.ll}
```

or, if they are in a subdirectory "ll":

```
FOO_LLS = ${FOO_OBJS:%.o=ll/%.ll}
```

Note that if you want to keep the .ll files in subdirectory ll/, you can have the makefile automatically create this file with the label:

```
.INIT:
    @if [ ! -d ll ]; then mkdir ll; fi
```

B

Index

Symbols

`.ll` file
    access to a variable, 3
    flow of control in each function, 3
    operation on a mutex, 3
    operation on readers-writer lock, 3

A

acquisition order of locks
    `assert order`, 6
`analyze`
    check asserted lock order, 14
    check for consistent use of locks with condition variables, 15
    check for function pointers with no targets, 14
    check for functions with variable side effects on locks, 13
    check locking side effects in function pointer targets, 15
    determine if locks consistently held when each function is entered, 15
    determine if locks consistently held when each variable is accessed, 15
    determine side effects, 14
    `-h` history
        produce detailed information about each analysis phase, 13
    process the various `allow/disallow` subcommands, 14
    remove references to ignored variables, 14
    `-v` verbose
        generate additional messages during analysis, 13
analyze the loaded files for lock inconsistencies
    `analyze`, 13
ascertain order of lock acquisition
    `order`, 10
ascertain root functions
    `funcs -o`, 7
ascertain variables not properly locked
    `vars -h`, 7
`assert`
    expected access to locks and variables, 15, 16
    expected modification of locks and variables, 16
    maximum violations of a single assert, 17
    mutex lock, 17

readers-writer lock, 17
specify order in which locks should be acquired, 17
specify that the given set of variables are never expected to be written, 17
assert locks
`assert`, 6
assertion
lock protects particular data or function, 9
order in which a given lock is acquired, 9
side effect of releasing or acquiring a lock, 9

C

checking an application
suggested approach, 6
collect `lock_lint` information
`.ll` file, 3
compiler option
-Zll
collect `lock_lint` information, 3
condition variable lock, 2
libthread API
kernel routines, 2
user routines, 2
conflict with keywords
escape double quotes, 42
constrain execution path
`disallow`, 8

D

data race
common cause, 2
definition of, 1
deadlock
definition of, 2
`declare func_ptr targets`
specify functions called through the particular function pointer, 19
`declare mutex`
analyze libraries without a supporting harness, 19
`declare nonreturning`
specify functions do not return, 19
`declare one`
tell `lock_lint` that only one unnamed instance exists of each structure whose tag is specified, 19
`declare readable`
tell `lock_lint` that the specified variables may be safely read without holding any lock, 20
`declare root`
analyze libraries without a supporting harness, 19
tell `lock_lint` that it should try to analyze the given functions as a root function, 20
`declare rwlocks`
analyze libraries without a supporting harness, 19
describe lock hierarchy
`rwlock`, 7
`disallow`
tell `lock_lint` that the specified calling sequence should not be analyzed, 20
`disallows`
list disallowed the calling sequences, 21
duplicate definition
struct with no tag, 42
duplicate name
automatic variables, 43
static variables, 42

E

exit `lock_lint`

shell exit, 6

F

`files`
list `.ll` source code files, 21
formal name
function, 40
members of a structure, 41
members of union, 42
reference to member of structure without tag, 41
variable, 40
`funcptrs`
`-b bound`
list function pointers bound to function targets, 21
list information about the function pointers, 21
`-o =ignored`
indicates lock_lint told to ignore the function, 24
`-o =nonreturning`
indicates call through this function pointer never returns, 22, 24
`-o =recursive`
indicates function calls itself, 24
`-o =root`
indicates function is s root, 24
`-o =rooted`
indicates function was made a root, 24
`-o =unanalyzed`
indicates function never called during analysis, 24
`-o calls=#`
indicates number of places function is called, 25
indicates number of places function pointer is used to call a function, 22
`-o other`
list number of places a particular function pointer calls a function, 21
`-t targets`
lists functions currently bound as targets, 22
`-u unbound`
list function pointers not bound to function targets, 22
`-z zero`
list function pointers for which there are no calls, 22
`funcs`
`-a asserts`
show information about locks held on entry to each function, 23
`-d defined`
show only those functions defined in loaded files, 23
`directly`, 27
`-e effects`
show information about side effects each function has on locks, 23
`-h held`
show information about locks consistently held when function was called, 24
`-i ignored`
list ignored functions, 24
`-l long`, 24
list information about functions in loaded files, 23
`-o other`
list additional information about a function, 24
`-u undefined`
show functions undefined in loaded files, 25

G

get information about function pointer calls
`pointer calls`, 8

get information about function pointers
funcptrs, 8
get information about locks protecting a variable
vars, 10
get information about variables
vars, 9

H

help
display subcommand set, 27
environment variable PAGER, 27
keyword, 27

I

ignore
exclude certain functions and variables from the analysis, 28
ignore access to variable
ignore, 9
ignore analysis of functions or variables
ignore, 7
inversion
defintion of, 43

L

libthread API, 2
mutex lock, 2
reader-writer lock, 2
list functions called as a result of calling specified functions, 25
list functions that directly call other functions, 25
list functions that invert specifed locks, 26
list functions that when called access specified variables, 26
list functions that when called affect specified locks, 26
list functions that when called read specified variables, 26
list functions that when called write specified variables, 26
list information about all unignored functions, 25
list information about all unignored variables, 39
list information about individual functions, 25
list information about individual variables, 39
list variables accessed by calling specified function, 40
list variables protected by specified lock, 39
list variables read by calling specified function, 40
list variables written by calling specified function, 40
load
load specified .ll files, 28
load .ll database filesload, 8
load .ll files
load, 6
lock_lint
collecting information for, 3
recommended way to use, 3
lock_lint report
inconsistent lock order acquisition, 3
lock_lint database, 9
lock_lint report
access to inconsistently locked variable, 3
assertion violations, 3
locking side effects, 3
lock_lint state
set of databases loaded, 6
specified assertions, 6
lock_lint subcommands, 5
locks
-c cover
show information about lock hierarchies, 29

list information about the locks of the loaded files, 29
list locks affected by calling specified functions, 30
list locks inverted by calling specified functions, 30
`-o other`
shows type of the lock, 29

M

`members`
list members of tagged struct, 30
multithreading model, 1
mutex lock
libthread API
kernel routines, 2
user routines, 2

O

`order`
list information about order in which locks are acquired, 31

P

`pointer calls`
list calls made through function pointers, 32
preprocessor macro
__lock_lint, 47

R

reader-writer lock
libthread API
kernel routines, 2
user routines, 2
`reallow`
allow exceptions to `disallow` subcommand, 33
`reallows`
list reallowed calling sequences, 34
`refresh`
pop the saved state stack, 34
remove execution path constraints
`reallow`, 8
report assertion violations
`analyze`, 9
report consistency of mutex lock on a particular condition variable
`analyze`, 10
report side effects on locks
`analyze`, 10
`restore`
pop the saved state stack, 34
run the analysis
`analyze`, 7
`analyze -v`, 7

S

`save`
save current stack state, 34
save `lock_lint` state
`save`, 7
`saves`
list descriptions of states saved on stack, 35
side effect
definition of, 16
`start`
execute csh command file, 36
execute korn shell, 37
start `lock_lint` session, 35
start `lock_lint`
SPARCworks Manager, 6
user-specified shell, 6
`sym`
list fully-qualified names of various symbols, 37
symbols
unambiguous identification, 41

T

target function as root
`declare ... targets`, 7

U

`unassert`
    undo assertions about locks protecting specified variables, 37

V

vars
    -h held
        show information about locks held when variable accessed, 38
`vars`
    `-a assert`
        show information about lock assertions, 38
    `-i ignored`
        `list ignored variables`, 38
    `-l long`, 39
    list information about variables, 38
    `-o other`, 39
view call graph for loaded sources
    `funcs`, 8