Using Race Coverage with GCT

Brian Marick
Testing Foundations

Documentation for version 1.4 of GCT
Document version 1.2

This manual describes how to measure *race coverage*, an extension of routine coverage useful for system testing of multi-threaded or multi-processor programs.

Preface

For more information about GCT, other products, or other services, contact:

Brian Marick
Testing Foundations
809 Balboa
Champaign, Illinois 61820

(217) 351-7228
Email: marick@cs.uiuc.edu, testing!marick@uunet.uu.net

You can join the GCT mailing list by sending mail to gct-request@cs.uiuc.edu.

Introduction

Race coverage concerns itself with *threads*. You have threads when two simultaneous executions of some program text can share data. Examples:

(1) POSIX threads.

(2) Multiple processors executing the UNIX kernel at the same time.

Since the threads share data, they must protect it from simultaneous access. The usual approach is locking. A lock is placed before each access to a datum or group of data. Only one thread at a time may pass through the lock. When the successful thread is finished with the datum, it unlocks it, allowing one of any waiting threads to proceed. (Other strategies for protecting data are possible; race coverage applies to them as well, since it measures whether there was an opportunity for simultaneous access.)

Incorrect data protection can be of various types:

(1) Failure to lock. Here, simultaneous access can lead to corruption, which is often quickly fatal.

(2) Failure to release a lock. Since the data remains locked, threads can wait forever. The system often quickly comes to a halt.

(3) Deadlock. Thread A has lock 1 and is waiting for thread B to release lock 2. Thread B is waiting for A to release lock 1. Neither will release their locks until the other does.

You can be more confident that a particular routine locks correctly if there have been several cases where several threads were in it simultaneously. (For shorthand, we say that the routine is then *racing*.) Race coverage measures how often a routine races.

If incorrect locking is a risk for your system, and you have routines that have never been raced, direct your testing effort at them. A good approach is to find groups of functionally related unraced routines (file manipulation routines, for example) and write stress tests for those groups. For example, you might write a test where many processes write simultaneously to many files. This will cause more routine entries, thus more simultaneous routine entries, thus perhaps more revealed locking problems.

Like all coverage, race coverage is an indirect measure of quality. Very high race coverage would not force the detection of deadlocks between the file system code and, say, networking code. You could achieve high race coverage by first stressing the file system, then stressing networking, but you'd never find failures that result from interactions between the two. In practice, race coverage often does a surprisingly good job of finding interaction bugs like deadlocks. You can increase the chance by running stress tests against a background load that moderately exercises all of the system.

It might be reasonable to extend race coverage so that it checks whether a thread has been in routine F while another thread was in routine G. Similarly, you might want to check whether a particular loop's body has been raced, rather than whether the loop's containing routine has. These would entail more overhead, and they have not been tried. To date, race coverage has been good enough at pointing to weaknesses in system stress tests.

The overhead of race coverage is low. For example, when measuring the race coverage of an operating system kernel, it is perfectly reasonable to instrument "development systems" -- the kernels running on the machines the developers use for editing and compiling. Comparison of the measurements for those systems to the measurements of specially-stressed systems can be illuminating.

Acknowledgement: The form of race coverage implemented by GCT was suggested by Gary Whisenhunt. He is not responsible for the name, which is not a very good one.

Using Race Coverage

1. Setup

(1) Add a line like

```
(coverage race)
```

to the control file. Race coverage cannot be combined with operator or operand coverage. It can be combined with all other kinds of coverage.

Like routine instrumentation, race instrumentation is immune to the effect of the **macros** option. As with routine instrumentation, empty functions are not instrumented.

(2) Edit the file gct-defs.h. You’ll see this code:

```
<<<TAG>>>:  You may need to declare externs here.

/* This yields a thread number from 0 to bits-in-word (for this simple
   implementation) */
#define GCT_THREAD       <<<you must define this>>>

/* The value of a group/thread - for debugging and testing */
#define GCT_RACE_GROUP_VALUE(group, thread) \
    (Gct_group_table[(group)] & (1 << (thread)))
```

You must define the GCT_THREAD macro to return a number between 0 and the number of bits in a word (less one). If you have more threads than will fit in a word, you’ll have to make more extensive changes. The next chapter will give you the background you need.

Often, the thread is identified by a variable. In a multiprocessing UNIX kernel, for example, the definition might be

```
#define GCT_THREAD    cpuid
```

In this case, you’ll need to declare **cpuid** in all files by changing the line beginning <<<TAG>>> to

```
extern int cpuid;
```

(3) When compiling the instrumented file, you may see warnings about unreachable statements. You can ignore them. (See below for the cause.)

2. Greport

The output identifies a routine that has never had two threads in it at once.

```
"example1.c", line 9: race in myfunc is never probed.
```

As with routine coverage, the line reported is the first line of a routine.

How Race Coverage Works

3. How Race Coverage Works.

The current implementation has some minor support for a future extension: measuring races in groups of routines, rather than in a single routine. For that reason, this discussion talks about "race groups". When you see that term, think "routine".

When any routine of a race group is entered, the condition count is incremented if another thread is currently in that race group. A thread leaves the race group when the routine returns or calls another routine. (In the case of a call, the thread reenters the race group when it returns from the call.) Thus, the count is incremented only if two threads are executing the immediate body of a race group routine at the same time.

Here's what GCT does, in more detail:

(1) GCT inserts the macro GCT_RACE_GROUP_CHECK just before the first statement of a routine. This might be the result of that instrumentation:

```
main()
{
  char myvar = complicated_routine();

  GCT_RACE_GROUP_CHECK(23, 1); myvar++;
  ...
}
```

Notice that the macro is placed after any variable declarations, even if those declarations call many functions. If thread 1 is executing the declarations and thread 2 is in the body of a function, a bug is less likely to be triggered than if both are in the body, so we count only the latter case.

GCT_RACE_GROUP_CHECK checks whether the race group (1, in this case) is racing. If so, it increments the appropriate condition count (23, in this case). There is one condition count for every race group.

The default definition of this GCT_RACE_GROUP_CHECK simply checks whether any bit is set:

```
/* Test whether another thread is in the same race group. */
#define GCT_RACING(group)    (Gct_group_table[(group)])

#define GCT_RACE_GROUP_CHECK(index, group)\
   (_G(index, GCT_RACING(group)))
```

_G is a standard GCT macro that increments the condition count if the second argument is non-zero. Gct_group_table is an array of words. Each thread is assigned one of the bits. When a thread enters a routine, its bit is set. This macro will have to be changed if there are more threads than bits in a word.

The GCT_RACING test works because the current thread does not record its own entry into the race group until after the check.

(2) Immediately after GCT_RACE_GROUP_CHECK, GCT inserts a call to GCT_RACE_GROUP_ENTER, like this:

```
main()
{
  char myvar = complicated_routine();

  GCT_RACE_GROUP_CHECK(23, 1); GCT_RACE_GROUP_ENTER(1); myvar++;
  ...
}
```

GCT_RACE_GROUP_ENTER takes a number which uniquely identifies the function. Its default definition (found in gct-defs.h) looks like this:

```
#define GCT_RACE_GROUP_ENTER(group) \
   (Gct_group_table[(group)] |= (1 << GCT_THREAD))
```

(3) Normally, we do not consider a routine to be racing if thread 1 is in its body and thread 2 is in some subfunction: in such a case, there's a much smaller chance that the two routines will actually have a locking conflict. Therefore, when a function is called, GCT surrounds its call with the macros GCT_RACE_GROUP_CALL and GCT_RACE_GROUP_REENTER:

```
i = find_something();
```

becomes

```
i = (GCT_RACE_GROUP_CALL(1), _G123 = find_something(),
GCT_RACE_GROUP_REENTER(1), _G123);
```

These routines turn the thread's bit off and then back on, respectively:

```
#define GCT_RACE_GROUP_CALL(group)
   (Gct_group_table[(group)] &= ˜(1 << GCT_THREAD))

#define GCT_RACE_GROUP_REENTER(group)
   (Gct_group_table[(group)] |= (1 << GCT_THREAD))
```

If you prefer to count threads in subfunctions, simply define these macros to do nothing:

```
#define GCT_NO_OP  49   /* random expression needed to avoid syntax error */
#define GCT_RACE_GROUP_CALL(group)    GCT_NO_OP
#define GCT_RACE_GROUP_REENTER(group)    GCT_NO_OP
```

If you define GCT_RACE_GROUP_CALL to do nothing, GCT_RACE_GROUP_CHECK has to be changed to avoid counting recursive calls as races. See *gct-defs.h* for the new definition.

Note that special non-returning "magic cookies" like longjmp() or the UNIX kernel's swtch() routines look like function calls; they will not fool GCT.

(4) When a routine returns (whether explicitly or by falling off the end), GCT_RACE_GROUP_EXIT is

called. If the original end of the routine looked like:

```
    if (errors > 0)
    return;
    clean_up_temporaries();
  }
```

GCT would rewrite it as

```
    if (errors > 0)
    { GCT_RACE_GROUP_EXIT(1); return;}
    clean_up_temporaries();
  GCT_RACE_GROUP_EXIT(1);}
```

The standard macro does the same thing as GCT_RACE_GROUP_CALL. GCT_RACE_GROUP_EXIT will not be added before the closing brace if a return statement directly precedes it. However, in code like

```
    if (test)
    return 5;
    else
    return 4;
    } /* End of routine. */
```

GCT will add a GCT_RACE_GROUP_EXIT before the closing brace. This may provoke warnings about unreached statements from your compiler; ignore them.

3.1. Miscellany

You might be tempted to worry about problems caused by the lack of locking in the GCT macros. Don't bother. The lack of locking may lead to a "missed decrement" if two threads are manipulating the Gct_group_table at the same time: in this case, the number of races will be inflated -- that's OK, since your goal was to find when a race occurred and one certainly did. The lack of locking may also cause an underestimate of the number of races. That's OK, too: you should aim to count many races, so mistakenly seeing no races when there was in fact exactly one will probably not change your interpretation of the results.

Note that GCT_RACE_GROUP_EXIT is placed before return statements. That means that the returned expression is "outside" the race group, exactly as declarations are. No GCT_RACE_GROUP_CALL or GCT_RACE_GROUP_REENTER macros are generated for function calls within returned expressions.