Code Re-entrancy vs. Thread Safety

What is the difference between the concepts of **"Code Re-entrancy"** and **"Thread Safety"**? As per the link mentioned below, a piece of code can be either of them, both of them or neither of them.

Reentrant and Thread safe code

I was not able to understand the explaination clearly. Help would be appreciated.

multithreading thread-safety reentrancy

edited Apr 30 '12 at 8:54
Nınɛfingɛrç
19.6k 6 57 112

asked Dec 9 '08 at 10:45
Codex
582 1 9 23

2 Answers

Re-entrant code has no state in a single point. You can call the code while something is executing in the code. If the code uses global state, one call can conceivably overwrite the global state, breaking the computation in the other call.

Thread safe code is code with no race conditions or other concurrency issues. A race condition is where the order in which two threads do something affects the computation. A typical concurrency issue is where a change to a shared data structure can be paritially completed and left in an incosistent state. In order to avoid this, you have to use concurrency control mechanisms such as semaphores of mutexes to ensure that nothing else can access the data structure until the operation is completed.

For example, a piece of code can be non re-entrant but thread-safe if it is guarded externally by a mutex but still has a global data structure where the state must be consistent for the entire duration of the call. In this case, the same thread could initiate a call-back into the procedure while still protected by an external coarse-grained mutex. If the call-back occured from within the non re-entrant procedure the call could leave the data structure in a state that could break the computation from the caller's point of view.

A piece of code can be re-entrant but non thread-safe if it can make a non-atomic change to a shared (and sharable) data structure that could be interrupted in the middle of the update leaving the data structure in an incosistent state. In this case another thread accessing the data structure could be affected by the half-changed data structure and either crash or perform an operation that corrupts the data.

edited Dec 9 '08 at 11:00

answered Dec 9 '08 at 10:54
ConcernedOfTunbridgeW
39.2k 10 93 161

3 Your second example doesn't sound re-entrant to me. If the change can be interrupted leaving an inconsistent state, and a signal occurs at that point, and the handler for that signal calls the function, then typically it goes boom. That's a re-entrancy issue, not a thread-safety issue. – Steve Jessop Dec 9 '08 at 12:22

1 You're right - as you say below you would also have to disable signals for the latter example to be effectively re-entrant. – ConcernedOfTunbridgeWells Dec 9 '08 at 14:16

@ConcernedOfTunbridgeWells, if a func uses heap inside, there is a good chance that this func is not re-entrant. Why? – Alcott Sep 21 '11 at 4:38

@Alcott - If a function solely uses variables allocated on the stack then each invocation will create a new stack frame, so there is no shared state. If the heap contains a shared data structure used by the function then colliding invocations could potentially corrupt the shared data structure unless some protection mechanism (e.g. a mutex) is placed around it. – ConcernedOfTunbridgeWells Sep 21 '11 at 12:29

That article says:

"a function can be either reentrant, thread-safe, both, or neither."

It also says:

"Non-reentrant functions are thread-unsafe".

I can see how this may cause a muddle. They mean that standard functions documented as not required to be re-entrant are also not required to be thread-safe, which is true of the POSIX libraries iirc (and POSIX declares it to be true of the ANSI/ISO libraries too, ISO having no concept of threads and hence no concept of thread-safety). In other words, "if a function says it is non-reentrant, then it is saying it's thread-unsafe too". That's not a logical necessity, it's just a convention.

Here's some pseudo-code which is thread-safe (well, there's plenty of opportunity for callbacks to create deadlocks due to locking inversion, but let's assume the documentation contains sufficient information for users to avoid that) but not re-entrant. It is supposed to increment the global counter, and perform the callback:

```
take_global_lock();
int i = get_global_counter();
do_callback(i);
set_global_counter(i+1);
release_global_lock();
```

If the callback calls this routine again, resulting in another callback, then both levels of callback will get the same parameter (which might be OK, depending on the API), but the counter will only be incremented once (which is almost certainly not the API you want, so it would have to be banned).

That's assuming the lock is recursive, of course. If the lock is non-recursive, then of course the code is non-reentrant anyway, since taking the lock the second time won't work.

Here's some pseudo-code which is "weakly re-entrant" but not thread-safe:

```
int i = get_global_counter();
do_callback(i);
set_global_counter(get_global_counter()+1);
```

Now it's fine to call the function from the callback, but it's not safe to call the function concurrently from different threads. It's also not safe to call it from a signal handler, because re-entrancy from a signal handler could likewise break the count if the signal happened to occur at the right time. So the code is non-re-entrant by the proper definition.

Here's some code which arguably is fully re-entrant (except I think the standard distinguishes between reentrant and 'non-interruptible by signals', and I'm not sure where this falls), but still isn't thread-safe:

```
int i = get_global_counter();
do_callback(i);
disable_signals(); // and any other kind of interrupts on your system
set_global_counter(get_global_counter()+1);
restore_signal_state();
```

On a single-threaded app, this is fine, assuming that the OS supports disabling everything that needs to be disabled. It prevents re-entrancy from occurring at the critical point. Depending how signals are disabled, it may be safe to call from a signal handler, although in this particular example there's still the issue of the parameter passed to the callback being the same for separate calls. It can still go wrong multi-threaded, though.

In practice, non-thread-safe often implies non-re-entrant, since (informally) anything that can go wrong due to the thread being interrupted by the scheduler, and the function called again from another thread, can also go wrong if the thread is interrupted by a signal, and the function is called again from the signal handler. But then the "fix" to prevent signals (disabling them) is different from the "fix" to prevent concurrency (locks, usually). This is at best a rule of thumb.

Note that I've implied globals here, but exactly the same considerations would apply if the function took as a parameter a pointer to the counter and the lock. It's just that the various cases would be thread-unsafe or non-re-entrant when called with the same parameter, rather than when called at all.

edited Dec 10 '08 at 3:10

answered Dec 9 '08 at 13:03

Steve Jessop
175k 15 235 488