Recommended practices for re-entrant code in C, C++

I was going through a re-entrancy guide on recommended practices when writing re-entrant code.

What other references and resources cover this topic?

What lint-like tools can be used to check for these issues?

c++ c reentrancy

edited Jul 14 '10 at 2:25
Ben Voigt
168k 15 175 342

asked Jul 13 '10 at 18:30
Fanatic23
1,610 9 29

That guide pretty much has everything you need to know. It's not a very subtle thing. Don't use data that you weren't passed in the function call, don't store any state in the function. – Stephen Canon Jul 14 '10 at 2:13

1 That guide has numerous errors in it, from a definition of reentrancy that's actually multithreading (though the signal processing examples are true reentrancy) to bad advice (mutexes... hello deadlock) to just plain bugginess (`sigsuspend(&zeromask)` .. you've just allowed processing interrupts that your caller disabled for a reason, try `sigsuspend(&oldmask)` instead). – Ben Voigt Jul 14 '10 at 2:22

4 Answers

The guide is sufficient.

My personal rule of thumbs are only 2 for re-reentering code:

1. take only pass by value parameters, used only value passed in as parameters in the function.
2. if I need to use any global parameters or pointer (for performance or storage sake), use a mutex or semaphore to control access to it.

answered Jul 14 '10 at 2:04
ttchong
214 2 9

If you need complex types, make them immutable if possible – fmark Jul 14 '10 at 2:06

Hi fmark: can you explain a further? Or point me to something related to this statement? – ttchong Jul 14 '10 at 2:09

7 no No NO! Reentrancy != Threading. Mutexes and semaphores will deadlock reentrant code, or else silently fail to do their job and leave data corruption. – Ben Voigt Jul 14 '10 at 2:16

2 Here's another example of code that is reentrant but not thread-safe. Let's suppose I have two functions that adjust the FPU control word (one needs exceptions enabled, the other needs them disabled). Both functions save the current state of the FPU control word on entry and restore it on exit. Now, these two functions can mutually recurse all day long, you can use them safely from signal handlers too -- they are reentrant. But they are definitely not thread-safe. – Ben Voigt Jul 14 '10 at 3:05

1 @Chethan: If you use incorrect definitions of standard terminology, you can make any statement correct. But also meaningless. – Ben Voigt Mar 21 '13 at 16:45

None really. Writting non-reentering code is usually more difficult than re-entring. Just follow those simple guidelines and don't try to do anything too waky and you'll be fine.

Non-reentering code is usually written for high-performance issues.

answered Jul 13 '10 at 18:44

Gianni
3,110 7 15

- Do use local variables.
- Don't use static locals or global variables, even TLS will not help you with recursion / reentrancy.
- Restore all your invariants before doing callbacks.
- Don't hold locks while you do callbacks. If you absolutely must (and I would still go looking for a way to avoid it) then make sure you know what happens if you try to re-enter your lock on the thread that already holds it. At a minimum you have to test for this, otherwise depending on the lock you'll get deadlocks or broken invariants (i.e. corruption).

answered Jul 14 '10 at 2:14

Ben Voigt
168k 15 175 342

---

1. A reentrant function may not use variables in a non-atomic way unless they are stored on the stack of the calling task or are the private variables of that task.
2. A reentrant function may not call other functions which are not reentrant.
3. A reentrant function may not use the hardware in a non-atomic way.

Ref: Page 462 [AN INTRODUCTION USING THE RENESAS RX62N MICROCONTROLLER] [James M. Conrad]

answered Oct 9 '14 at 15:20

Osaid
122 1 12
