Thread safety vs Re entrancy

I know this has been discussed a lot in this forum. But one thing still confuses me. Wikipedia mentions that every re entrant code is thread safe. http://en.wikipedia.org/wiki/Reentrant_%28subroutine%29 And later on gives an example of a function which is re entrant but not thread safe.

```
int t;

void swap(int *x, int *y)
{
    int s;

    s = t;  // save global variable
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
    t = s;  // restore global variable
}

void isr()
{
    int x = 1, y = 2;
    swap(&x, &y);
}
```

This confuses me. Are all re entrant codes thread safe? Also, are all recursive function thread safe. I am not able to visualize the difference.

Thanks

multithreading operating-system

asked Feb 25 '12 at 3:48

Deepti Jain
408 2 6 18

The Wiki article has 'issues', IMHO. The example above that you pasted in is not really a reentrant function that is not thread-safe, it's an attempt at a reentrant function with a deliberately-introduced bug. swap() does not need the global 't' at all. Is's just silly. If a function can be interrupted, reentered, (from another stack), and then return the wrong results for one or both calls, it is not reentrant! A thread context-change is an interrupt and therefore a reentrant function must be able to return the correct results to all calling threads. – Martin James Feb 25 '12 at 12:07

@MartinJames Please see my answer for examples of a thread-safe non-reentrant and a reentrant non-thread-safe functions. Both are admittedly somewhat contrived, but I think that they illustrate the point. – dasblinkenlight Feb 25 '12 at 14:20

5 Answers

The concepts of re-entrancy and thread safety are related, but not equivalent. You can write a re-entrant function that is not thread-safe, and a thread-safe function that is not re-entrant. I will use C# for my examples:

Re-entrant Function that is not Thread-Safe

This function reverses the entries of an array:

```
void Reverse(int[] data) {
    if (data == null || data.Length < 2) return;
    for (var i = 0 ; i != data.Length/2 ; i++) {
        int tmp = data[i];
        data[i] = data[data.Length-i-1];
```

```
        data[data.Length-i-1] = tmp;
    }
}
```

This function is clearly re-entrant, because it does not reference outside resources. Its thread safety, however, is *conditional* upon not passing it the same `data` from multiple threads. If several threads concurrently pass `Reverse` the same instance of an array, an incorrect result may be produced. One way of making this function unconditionally thread-safe would be an addition of a lock:

```
void Reverse(int[] data) {
    if (data == null || data.Length < 2) return;
    lock (data) {
        for (var i = 0 ; i != data.Length/2 ; i++) {
            int tmp = data[i];
            data[i] = data[data.Length-i-1];
            data[data.Length-i-1] = tmp;
        }
    }
}
```

Thread-Safe Function that is not Re-Entrant

This function calls function `f()` `c` times, and returns the value that has been returned more times than other values.

```
static int[] counts = new int[65536];

unsigned short MaxCount(Func<unsigned short> f, int c) {
    lock(counts) {
        Array.Clear(counts, 0, counts.Length);
        for (var i = 0 ; i != c ; i++) {
            counts[f()]++;
        }
        unsigned short res = 0;
        for (var i = 1 ; i != counts.Length ; i++) {
            if (counts[i] > counts[res]) {
                res = i;
            }
        }
        return res;
    }
}
```

This function is thread-safe, because it locks the static array that it uses to do the counting. However, it is not re-entrant: for example, if the functor `f` passed in were to call `MaxCount`, wrong results would be returned.

answered Feb 25 '12 at 14:18

dasblinkenlight
310k 27 270 513

2 'This function is clearly re-entrant' - what happens if it is interrupted and the interrupt-handler passes it the same array? Adding a lock to such a routine would probably be disastrous as locks are typically not available to interrupt-handlers/drivers - the OS would probably double-fault halt or BSOD. The point is that, if you use the argument of 'passing in the same parameters by reference' to say that a function is no thread-safe, it also implies that the function is not reentrant. – Martin James Feb 25 '12 at 16:09

IMHO 'reentrant' should mean that you can call it from anywhere, with anything, and it will always work. – Martin James Feb 25 '12 at 16:14

1 @MartinJames My understanding is that code is reentrant when it *can* be used in a reentrant way, perhaps by following certain rules. For `Reverse(int[])` the rules are that you cannot pass the same array as its argument. In contrast, `strtok` cannot be made reentrant by following rules, no matter what rules you set. Even adding a lock around the access to its static variable would make `strtok` thread-safe without making it reentrant: you cannot use `strtok` in two nested loops parsing *different* strings even with a lock around calls to it. Therefore, `strtok` is not reentrant. – dasblinkenlight Feb 25 '12 at 17:36

> Are all re entrant codes thread safe?

No.

Later in the Wikipedia article: *…The key for avoiding confusion is that reentrant refers to only one thread executing. It is a concept from the time when no multitasking operating systems existed.*

In other words, the concept of reentrance is oblivious to multithreaded execution (as the most common example). Only if that is an actual constraint in your system, then yes, reentrant would qualify as thread safe (*but* there would also be only one thread in that environment).

IMO, the term "reentrant" should be used only in the context of qualifying systems (does not apply to your desktop OS, but would apply to *some* embedded systems).

Now, if you really want to force use of the word 'reentrance' into a multithreaded environment: You could enforce reentrance in a multithreaded system only if you also guaranteed that it is also threadsafe. Perhaps a better question to ask would be "*In order to guarantee reentrance in a multithreaded context, would that imply the function and all data it refers to also needs to be threadsafe?*" - A: Yes, the function and everything it refers to would also need to be threadsafe and reentrant for the function to be reentrant in that context. Achieving this quickly becomes complex, and is a reason global variables are not a good idea.

> Also, are all recursive function thread safe.

No.

edited Feb 25 '12 at 6:03

answered Feb 25 '12 at 5:15

justin
77.6k 10 105 160

OK, I'll bite. What reentrant code/data is not thread-safe? A reentrant function can be interrupted, called again from another context and will return the correct result for both the interrupted and interrupting contexts. Preemptive multitaskers are interrupt-handlers, so where's the problem? – Martin James Feb 25 '12 at 11:03

@MartinJames I suppose the problem is that you don't agree with my answer, with several points/examples in the wikipedia article, and other answers. – justin Feb 26 '12 at 1:01

The function "swap" is not reentrant, since it uses global state (to whit, the global variable "t").

answered Feb 25 '12 at 3:50

Perry
2,387 4 13

If your subroutine doesn't affect external variables (no side effects), you can say it **is re-entrant**. While this is not a rule of thumb, it is a good guideline.
If the subroutine uses external variables and saves state of external variables at the beginning and restores that state at the end, then the subroutine **is re-entrant**.

If a subroutine modifies external variables and it doesn't save their state first, and it is interrupted, the state of those external variables can be changed, thus when the call returns to the original location in the subroutine, the external variable is out of sync, which results in an inconsistent state for the subroutine.

Re-entrant functions save their state (on their local stack, within the thread's stack not using globals).

In your case, you access an external variable, `t`, but it is re-entrant because the variable is saved and restored at the end of the subroutine.

Normally **thread safe** implies re-entrant, but again not a rule more of a guideline. **Note:** Some locks in java are re-entrant so you can recursively call the method and not be blocked by previous calls. (re-entrance in this context means that a threads can access any section locked with the same lock - the thread can reenter any block of code for which it already holds the lock)

Solution/Answer:
If you protect `t` with atomics you should get a thread safe subroutine. Also, if you place `t` on each thread's stack (make it local) then the subroutine becomes thread safe since there is no global data.

Also, reentrant != recursive; you can just as well execute an ISR within a recursive subroutine. Thread-wise, if you call a recursive subroutine from 2 threads, you will get garbage. To make is thread safe, protect the recursive subroutine with re-entrant locks (other non re-entrant locks will result in deadlock/blocking).

edited Feb 25 '12 at 4:57

answered Feb 25 '12 at 4:10

Adrian

2,515 5 18 41

'Thread-wise, if you call a recursive subroutine from 2 threads, you will get garbage'. Why? A lot of recursive code uses only stack-frames for data - just locals and parameters, no statics/globals. – Martin James Feb 25 '12 at 9:56

@MartinJames The example I was using (in my head) was that recursive code could use some external variables. Normally, they wouldn't use that, but it is possible so they wouldn't be thread safe. I guess I should edit and clarify – Adrian Feb 25 '12 at 17:31

That function is not thread safe because it accesses a resource `t` that is outside of function scope (not allocated on the stack) without any protection mechanisms (e.g. a lock) to ensure that access to `t` is atomic.

The Wikipedia page actually states:

> A reentrant subroutine **can** achieve thread-safety, but this condition alone might **not be sufficient** in all situations.

(My emphasis).

UPDATE:

Reentrant (as defined in the article) simply means that a single thread of execution (think DOS) can "rip the instruction pointer" out of the middle of the executing routine, put it somewhere else, and continue a linear flow through new code (e.g. an interrupt subroutine in the DOS days), and that same linear flow can go back into the function a second time. In this limited scenario, the second invocation of the function MUST complete before control is transferred away from the interrupt subroutine back to "regular" program execution, which would resume at the same point in the routine where the instruction pointer was originally ripped away. This scenario does not allow for arbitrary thread scheduling, but rather for an interruption that switches control to a new point, which then completes, and resumes back at the point it was interrupted.

Note that in real life, things may not be so simple. I'm not entirely sure anymore (been... 20 years?), but I think one interrupt routine can interrupt and interrupt routine already in progress (e.g. a soft debugger interrupt can interrupt the timer interrupt, etc).

edited Feb 26 '12 at 2:43

answered Feb 25 '12 at 3:50

Eric J.

80.1k 21 158 320

1 So does that mean that every re entrant code is not necessarily thread safe. Right? – Deepti Jain Feb 25 '12 at 3:56

@DeeptiJain - this is the statement that I have a problem with. I'm having trouble thinking of an example of a reentrant function that is not thread-safe. – Martin James Feb 25 '12 at 11:41

Updated with thoughts on the difference between a threaded environment and one that allows for interrupts. – Eric J. Feb 26 '12 at 2:44