Is malloc thread-safe?

Is the `malloc()` function re-entrant?

c multithreading thread-safety malloc

edited Jul 5 '13 at 12:53
einpoklum
3,404 2 19 49

asked May 13 '09 at 2:17
Alphaneo
3,520 8 43 70

3 related: stackoverflow.com/questions/147298/… – lothar May 13 '09 at 2:20

18 Your title and body ask two different things. Re-entrant normally means "can be safely used in a signal handler", while thread-safe normally means "can be safely used in threads". It's easier to have thread-safe than re-entrant. – David Thornley Sep 20 '10 at 15:11

7 Answers

I read somewhere that if you compile with -pthreads, malloc becomes thread safe. I´m pretty sure its implementation dependant though, since malloc is ANSI C and threads are not.

If we are talking gcc:

> Compile and link with -pthreads and malloc() will be thread-safe, on x86 and AMD64.

http://groups.google.com/group/comp.lang.c.moderated/browse_thread/thread/2431a99b9bdcef11/ea8
79e40f7fa4

Another opinion, more insightful

> {malloc, calloc, realloc, free, posix_memalign} of glibc-2.2+ are thread safe

http://linux.derkeiler.com/Newsgroups/comp.os.linux.development.apps/2005-07/0323.html

edited May 13 '09 at 3:00

answered May 13 '09 at 2:26
Tom
18k 12 74 130

6 "I read somewhere..." is really not the sort of quality befitting an accepted answer.. ;-) – R.. Oct 21 '11 at 11:51

1 Whatever he provides the reference links. – chain ro Dec 3 '14 at 7:02

Question: "is malloc reentrant"?
Answer: no, it is not. Here is one definition of what makes a routine reentrant.

None of the common versions of malloc allow you to reenter it (e.g. from signal handler). Note that a reentrant routine may not use locks, and almost all malloc versions in existence do use locks (which makes them thread-safe), or global/static variables (which makes them thread-unsafe *and* non-reentrant).

All the answers so far answer "is malloc thread-safe?", which is entirely different question. To *that* question the answer is *it depends* on your runtime library, and possibly on the compiler flags you use. On any modern UNIX you'll get a thread-safe malloc by default. On Windows, use `/MT`, `/MTd`, `/MD` or `/MDd` flags to get thread-safe runtime library.

answered May 13 '09 at 5:15
Employed Russian
57.8k 7 61 116

8 Reentrant routines may use locks as long as the lock is reentrant; this necessitates the lock being roughly equivalent to a recursive or error-checking mutex, but with the added stipulation that the single atomic operation that takes the lock must result in the lock structure being in a consistent state for a newly-held lock with a single reference. Of course then you also have to deal with the case where the state protected by the lock has been partially-modified by the interrupted code in the same thread, but nonetheless reentrant locks can be a building block in handling that... – R.. Oct 21 '11 at 11:53

Here is an excerpt from malloc.c of glibc :

Thread-safety: thread-safe unless NO_THREADS is defined

assuming NO_THREADS is not defined by default, malloc is thread safe at least on linux.

answered Sep 20 '10 at 14:50
shahkhas
61 1 1

It depends on which implementation of the C runtime library you're using. If you're using MSVC for example then there's a compiler option which lets you specify which version of the library you want to build with (i.e. a run-time library that supports multi-threading by being tread-safe, or not).

answered May 13 '09 at 2:32
ChrisW
38.3k 5 58 134

No, it is not thread-safe. There may actually be a `malloc_lock()` and `malloc_unlock()` function available in your C library. I know that these exist for the Newlib library. I had to use this to implement a mutex for my processor, which is multi-threaded in hardware.

answered May 13 '09 at 3:02
sybreon
2,354 7 15

2 Some implementations of malloc are thread-safe. – ChrisW May 13 '09 at 4:14

3 Any implementation of malloc on any system conforming to any major threads standard (such as POSIX) is thread-safe. – R.. Oct 21 '11 at 11:54

No, it is not.

http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2007-07/msg02658.html

answered May 13 '09 at 2:25
Paul Sonier
26.3k 1 47 91

1 As stated at groups.google.com/group/comp.lang.c.moderated/browse_thread/…, this is not strictly correct. malloc is neither guaranteed to be, nor guaranteed not be, thread-safe. It is implementation-dependent. – Matthew Flaschen May 13 '09 at 2:30

2 Safety is not partial; if an implementation may not be safe, my interpretation is that it is not safe. Poster did not specify a platform; therefore, no guarantee of safety can be given. – Paul Sonier May 13 '09 at 2:54

2 Any given implementation either is or isn't safe. To say that an unknown implementation is unsafe is no more true than saying that it is safe. The correct answer is that it's implementation-dependent. – ChrisW May 13 '09 at 5:03

3 @ChrisW: since the OP did not specify the implementation, and since safety is an absolute condition (it can be broken with only one exception; only one unsafe action can make something "unsafe"), I decided to answer that it was unsafe. The original poster can determine for him/her self what is the correct answer. – Paul Sonier May 13 '09 at 14:21

2 @ChrisW - Russian roulette is safe. – Oktalist Jan 15 '13 at 21:05

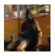
This is quite old question and I want to bring freshness according current state of things.

Yes, currently `malloc()` is thread-safe.

From `GNU C Library Reference Manual` of `glibc-2.20 [released 2014-09-07]` :

```
void * malloc (size_t size)
Preliminary: MT-Safe | ...
...
1.2.2.1 POSIX Safety Concepts
...
MT-Safe or Thread-Safe functions are safe to call in the presence of other threads. MT,
in MT-Safe, stands for Multi Thread.
Being MT-Safe does not imply a function is atomic, nor that it uses any of the memory
synchronization mechanisms POSIX exposes to users. It is even possible that calling
MT-Safe functions in sequence does not yield an MT-Safe combination. For example,
having a thread call two MT-Safe functions one right after the other does not guaran-
tee behavior equivalent to atomic execution of a combination of both functions, since
concurrent calls in other threads may interfere in a destructive way.
Whole-program optimizations that could inline functions across library interfaces may
expose unsafe reordering, and so performing inlining across the GNU C Library inter-
face is not recommended. The documented MT-Safety status is not guaranteed under
whole-program optimization. However, functions defined in user-visible headers are
designed to be safe for inlining.
```

answered Nov 20 '14 at 18:37



likern
356 5 17