NAME

attributes − POSIX safety concepts

DESCRIPTION

Note: the text of this man page is based on the material taken from the "POSIX Safety Concepts" section of the GNU C Library manual. Further details on the topics described here can be found in that manual.

Various function manual pages include a section ATTRIBUTES that describes the safety of calling the function in various contexts. This section annotates functions with the following safety markings:

MT-Safe

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 guarantee 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 interface 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.

MT-Unsafe

MT-Unsafe functions are not safe to call in a multithreaded programs.

Other keywords that appear in safety notes are defined in subsequent sections.

Conditionally safe features

For some features that make functions unsafe to call in certain contexts, there are known ways to avoid the safety problem other than refraining from calling the function altogether. The keywords that follow refer to such features, and each of their definitions indicates how the whole program needs to be constrained in order to remove the safety problem indicated by the keyword. Only when all the reasons that make a function unsafe are observed and addressed, by applying the documented constraints, does the function become safe to call in a context.

init Functions marked with *init* as an MT-Unsafe feature perform MT-Unsafe initialization when they are first called.

Calling such a function at least once in single-threaded mode removes this specific cause for the function to be regarded as MT-Unsafe. If no other cause for that remains, the function can then be safely called after other threads are started.

race Functions annotated with *race* as an MT-Safety issue operate on objects in ways that may cause data races or similar forms of destructive interference out of concurrent execution. In some cases, the objects are passed to the functions by users; in others, they are used by the functions to return values to users; in others, they are not even exposed to users.

const Functions marked with *const* as an MT-Safety issue non-atomically modify internal objects that are better regarded as constant, because a substantial portion of the GNU C Library accesses them without synchronization. Unlike *race*, which causes both readers and writers of internal objects to be regarded as MT-Unsafe, this mark is applied to writers only. Writers remain MT-Unsafe to call, but the then-mandatory constness of objects they modify enables readers to be regarded as MT-Safe (as long as no other reasons for them to be unsafe remain), since the lack of synchronization is not a problem when the objects are effectively constant.

The identifier that follows the *const* mark will appear by itself as a safety note in readers. Programs that wish to work around this safety issue, so as to call writers, may use a non-recursive read-write lock associated with the identifier, and guard *all* calls to functions marked with *const*

followed by the identifier with a write lock, and *all* calls to functions marked with the identifier by itself with a read lock.

sig Functions marked with *sig* as a MT-Safety issue may temporarily install a signal handler for internal purposes, which may interfere with other uses of the signal, identified after a colon.

This safety problem can be worked around by ensuring that no other uses of the signal will take place for the duration of the call. Holding a non-recursive mutex while calling all functions that use the same temporary signal; blocking that signal before the call and resetting its handler afterwards is recommended.

term Functions marked with *term* as an MT-Safety issue may change the terminal settings in the recommended way, namely: call **tcgetattr**(3), modify some flags, and then call **tcsetattr**(3), this creates a window in which changes made by other threads are lost. Thus, functions marked with *term* are MT-Unsafe.

It is thus advisable for applications using the terminal to avoid concurrent and reentrant interactions with it, by not using it in signal handlers or blocking signals that might use it, and holding a lock while calling these functions and interacting with the terminal. This lock should also be used for mutual exclusion with functions marked with *race:tcattr(fd)*, where *fd* is a file descriptor for the controlling terminal. The caller may use a single mutex for simplicity, or use one mutex per terminal, even if referenced by different file descriptors.

Other safety remarks

Additional keywords may be attached to functions, indicating features that do not make a function unsafe to call, but that may need to be taken into account in certain classes of programs:

locale Functions annotated with *locale* as an MT-Safety issue read from the locale object without any form of synchronization. Functions annotated with *locale* called concurrently with locale changes may behave in ways that do not correspond to any of the locales active during their execution, but an unpredictable mix thereof.

We do not mark these functions as MT-Unsafe, however, because functions that modify the locale object are marked with *const:locale* and regarded as unsafe. Being unsafe, the latter are not to be called when multiple threads are running or asynchronous signals are enabled, and so the locale can be considered effectively constant in these contexts, which makes the former safe.

env Functions marked with *env* as an MT-Safety issue access the environment with **getenv**(3) or similar, without any guards to ensure safety in the presence of concurrent modifications.

We do not mark these functions as MT-Unsafe, however, because functions that modify the environment are all marked with *const:env* and regarded as unsafe. Being unsafe, the latter are not to be called when multiple threads are running or asynchronous signals are enabled, and so the environment can be considered effectively constant in these contexts, which makes the former safe.

hostid The function marked with *hostid* as an MT-Safety issue reads from the system-wide data structures that hold the "host ID" of the machine. These data structures cannot generally be modified atomically. Since it is expected that the "host ID" will not normally change, the function that reads from it (**gethostid**(3)) is regarded as safe, whereas the function that modifies it (**sethostid**(3)) is marked with *const:hostid*, indicating it may require special care if it is to be called. In this specific case, the special care amounts to system-wide (not merely intra-process) coordination.

sigintr Functions marked with *sigintr* as an MT-Safety issue access the GNU C Library *_sigintr* internal data structure without any guards to ensure safety in the presence of concurrent modifications.

We do not mark these functions as MT-Unsafe, however, because functions that modify this data structure are all marked with *const:sigintr* and regarded as unsafe. Being unsafe, the latter are not to be called when multiple threads are running or asynchronous signals are enabled, and so the data structure can be considered effectively constant in these contexts, which makes the former safe.

cwd Functions marked with *cwd* as an MT-Safety issue may temporarily change the current working directory during their execution, which may cause relative pathnames to be resolved in unexpected ways in other threads or within asynchronous signal or cancelation handlers.

This is not enough of a reason to mark so-marked functions as MT-Unsafe, but when this behavior is optional (e.g., **nftw**(3) with **FTW_CHDIR**), avoiding the option may be a good alternative to using full pathnames or file descriptor-relative (e.g., **openat**(2)) system calls.

:identifier

Annotations may sometimes be followed by identifiers, intended to group several functions that, for example, access the data structures in an unsafe way, as in *race* and *const*, or to provide more specific information, such as naming a signal in a function marked with *sig*. It is envisioned that it may be applied to *lock* and *corrupt* as well in the future.

In most cases, the identifier will name a set of functions, but it may name global objects or function arguments, or identifiable properties or logical components associated with them, with a notation such as, for example, *:buf(arg)* to denote a buffer associated with the argument *arg*, or *:tcattr(fd)* to denote the terminal attributes of a file descriptor *fd*.

The most common use for identifiers is to provide logical groups of functions and arguments that need to be protected by the same synchronization primitive in order to ensure safe operation in a given context.

/condition

Some safety annotations may be conditional, in that they only apply if a boolean expression involving arguments, global variables or even the underlying kernel evaluates to true. For example, */!ps* and */one_per_line* indicate the preceding marker only applies when argument *ps* is NULL, or global variable *one_per_line* is nonzero.

When all marks that render a function unsafe are adorned with such conditions, and none of the named conditions hold, then the function can be regarded as safe.

SEE ALSO

pthreads(7), **signal−safety**(7)