NAME

jemalloc - general purpose memory allocation functions

LIBRARY

This manual describes jemalloc 5.2.1-0-gea6b3e973b477b8061e0076bb257dbd7f3faa756. More information can be found at the **jemalloc website**[1].

The following configuration options are enabled in libc's built-in jemalloc: **--enable-fill**, **--enable-lazy-lock**, **--enable-stats**, **--enable-utrace**, **--enable-xmalloc**, and **--with-malloc-conf=abort_conf:false**. Additionally, **--enable-debug** is enabled in development versions of FreeBSD (controlled by the **MK_MALLOC_PRODUCTION** make variable).

SYNOPSIS

#include <stdlib.h>
#include <malloc_np.h>

Standard API

void *malloc(size_t *size***);**

void *calloc(size_t *number***, size_t** *size***);**

int posix_memalign(void ** *ptr***, size_t** *alignment***, size_t** *size***);**

void *aligned_alloc(size_t *alignment***, size_t** *size***);**

void *realloc(void * *ptr***, size_t** *size***);**

void free(void * *ptr***);**

Non-standard API

void *mallocx(size_t *size***, int** *flags***);**

void *rallocx(void * *ptr***, size_t** *size***, int** *flags***);**

size_t xallocx(void * *ptr***, size_t** *size***, size_t** *extra***, int** *flags***);**

size_t sallocx(void * *ptr***, int** *flags***);**

void dallocx(void * *ptr***, int** *flags***);**

void sdallocx(void ****ptr*, size_t** *size***, int** *flags***);**

size_t nallocx(size_t *size***, int** *flags***);**

int mallctl(const char ****name*, void ****oldp***, size_t ****oldlenp***, void ****newp***, size_t** *newlen***);**

int mallctlnametomib(const char ****name*, size_t ****mibp***, size_t ****miblenp***);**

int mallctlbymib(const size_t ****mib*, size_t** *miblen***, void ****oldp***, size_t ****oldlenp***, void ****newp***,**
size_t *newlen***);**

void malloc_stats_print(void *(*write_cb)* **(void *, const char *), void ****cbopaque***, const char ****opts***);**

size_t malloc_usable_size(const void ****ptr*);**

void (*malloc_message)(void ****cbopaque*, const char ****s***);**

const char ****malloc_conf***;

DESCRIPTION

Standard API

The malloc() function allocates *size* bytes of uninitialized memory. The allocated space is suitably aligned (after possible pointer coercion) for storage of any type of object.

The calloc() function allocates space for *number* objects, each *size* bytes in length. The result is identical to calling malloc() with an argument of *number* * *size*, with the exception that the allocated memory is explicitly initialized to zero bytes.

The posix_memalign() function allocates *size* bytes of memory such that the allocation's base address is a multiple of *alignment*, and returns the allocation in the value pointed to by *ptr*. The requested *alignment* must be a power of 2 at least as large as sizeof(**void ***).

The aligned_alloc() function allocates *size* bytes of memory such that the allocation's base address is a multiple of *alignment*. The requested *alignment* must be a power of 2. Behavior is undefined if *size* is not an integral multiple of *alignment*.

The realloc() function changes the size of the previously allocated memory referenced by *ptr* to *size* bytes. The contents of the memory are unchanged up to the lesser of the new and old sizes. If the new size is larger, the contents of the newly allocated portion of the memory are undefined. Upon success, the memory referenced by *ptr* is freed and a pointer to the newly allocated memory is returned. Note

that realloc() may move the memory allocation, resulting in a different return value than *ptr*. If *ptr* is **NULL**, the realloc() function behaves identically to malloc() for the specified size.

The free() function causes the allocated memory referenced by *ptr* to be made available for future allocations. If *ptr* is **NULL**, no action occurs.

Non-standard API

The mallocx(), rallocx(), xallocx(), sallocx(), dallocx(), sdallocx(), and nallocx() functions all have a *flags* argument that can be used to specify options. The functions only check the options that are contextually relevant. Use bitwise or (|) operations to specify one or more of the following:

MALLOCX_LG_ALIGN(*la*)

Align the memory allocation to start at an address that is a multiple of (1 << *la*). This macro does not validate that *la* is within the valid range.

MALLOCX_ALIGN(*a*)

Align the memory allocation to start at an address that is a multiple of *a*, where *a* is a power of two. This macro does not validate that *a* is a power of 2.

MALLOCX_ZERO

Initialize newly allocated memory to contain zero bytes. In the growing reallocation case, the real size prior to reallocation defines the boundary between untouched bytes and those that are initialized to contain zero bytes. If this macro is absent, newly allocated memory is uninitialized.

MALLOCX_TCACHE(*tc*)

Use the thread-specific cache (tcache) specified by the identifier *tc*, which must have been acquired via the tcache.create mallctl. This macro does not validate that *tc* specifies a valid identifier.

MALLOCX_TCACHE_NONE

Do not use a thread-specific cache (tcache). Unless **MALLOCX_TCACHE(*tc*)** or **MALLOCX_TCACHE_NONE** is specified, an automatically managed tcache will be used under many circumstances. This macro cannot be used in the same *flags* argument as **MALLOCX_TCACHE(*tc*)**.

MALLOCX_ARENA(*a*)

Use the arena specified by the index *a*. This macro has no effect for regions that were allocated via an arena other than the one specified. This macro does not validate that *a* specifies an arena index in the valid range.

The mallocx() function allocates at least *size* bytes of memory, and returns a pointer to the base address of the allocation. Behavior is undefined if *size* is **0**.

The rallocx() function resizes the allocation at *ptr* to be at least *size* bytes, and returns a pointer to the base address of the resulting allocation, which may or may not have moved from its original location. Behavior is undefined if *size* is **0**.

The xallocx() function resizes the allocation at *ptr* in place to be at least *size* bytes, and returns the real size of the allocation. If *extra* is non-zero, an attempt is made to resize the allocation to be at least (*size* + *extra*) bytes, though inability to allocate the extra byte(s) will not by itself result in failure to resize. Behavior is undefined if *size* is **0**, or if (*size* + *extra* > **SIZE_T_MAX**).

The sallocx() function returns the real size of the allocation at *ptr*.

The dallocx() function causes the memory referenced by *ptr* to be made available for future allocations.

The sdallocx() function is an extension of dallocx() with a *size* parameter to allow the caller to pass in the allocation size as an optimization. The minimum valid input size is the original requested size of the allocation, and the maximum valid input size is the corresponding value returned by nallocx() or sallocx().

The nallocx() function allocates no memory, but it performs the same size computation as the mallocx() function, and returns the real size of the allocation that would result from the equivalent mallocx() function call, or **0** if the inputs exceed the maximum supported size class and/or alignment. Behavior is undefined if *size* is **0**.

The mallctl() function provides a general interface for introspecting the memory allocator, as well as setting modifiable parameters and triggering actions. The period-separated *name* argument specifies a location in a tree-structured namespace; see the MALLCTL NAMESPACE section for documentation on the tree contents. To read a value, pass a pointer via *oldp* to adequate space to contain the value, and a pointer to its length via *oldlenp*; otherwise pass **NULL** and **NULL**. Similarly, to write a value, pass a pointer to the value via *newp*, and its length via *newlen*; otherwise pass **NULL** and **0**.

The mallctlnametomib() function provides a way to avoid repeated name lookups for applications that repeatedly query the same portion of the namespace, by translating a name to a "Management Information Base" (MIB) that can be passed repeatedly to mallctlbymib(). Upon successful return from mallctlnametomib(), *mibp* contains an array of **miblenp* integers, where **miblenp* is the lesser of the number of components in *name* and the input value of **miblenp*. Thus it is possible to pass a **miblenp* that is smaller than the number of period-separated name components, which results in a partial MIB that can be used as the basis for constructing a complete MIB. For name components that are integers

(e.g. the 2 in arenas.bin.2.size), the corresponding MIB component will always be that integer. Therefore, it is legitimate to construct code like the following:

```
unsigned nbins, i;
size_t mib[4];
size_t len, miblen;

len = sizeof(nbins);
mallctl("arenas.nbins", &nbins, &len, NULL, 0);

miblen = 4;
mallctlnametomib("arenas.bin.0.size", mib, &miblen);
for (i = 0; i < nbins; i++) {
        size_t bin_size;

        mib[2] = i;
        len = sizeof(bin_size);
        mallctlbymib(mib, miblen, (void *)&bin_size, &len, NULL, 0);
        /* Do something with bin_size... */
}
```

The malloc_stats_print() function writes summary statistics via the *write_cb* callback function pointer and *cbopaque* data passed to *write_cb*, or malloc_message() if *write_cb* is **NULL**. The statistics are presented in human-readable form unless "J" is specified as a character within the *opts* string, in which case the statistics are presented in **JSON format**[2]. This function can be called repeatedly. General information that never changes during execution can be omitted by specifying "g" as a character within the *opts* string. Note that malloc_stats_print() uses the mallctl*() functions internally, so inconsistent statistics can be reported if multiple threads use these functions simultaneously. If **--enable-stats** is specified during configuration, "m", "d", and "a" can be specified to omit merged arena, destroyed merged arena, and per arena statistics, respectively; "b" and "l" can be specified to omit per size class statistics for bins and large objects, respectively; "x" can be specified to omit all mutex statistics; "e" can be used to omit extent statistics. Unrecognized characters are silently ignored. Note that thread caching may prevent some statistics from being completely up to date, since extra locking would be required to merge counters that track thread cache operations.

The malloc_usable_size() function returns the usable size of the allocation pointed to by *ptr*. The return value may be larger than the size that was requested during allocation. The malloc_usable_size() function is not a mechanism for in-place realloc(); rather it is provided solely as a tool for introspection purposes. Any discrepancy between the requested allocation size and the size reported by

malloc_usable_size() should not be depended on, since such behavior is entirely implementation-dependent.

TUNING

Once, when the first call is made to one of the memory allocation routines, the allocator initializes its internals based in part on various options that can be specified at compile- or run-time.

The string specified via **--with-malloc-conf**, the string pointed to by the global variable *malloc_conf*, the "name" of the file referenced by the symbolic link named /etc/malloc.conf, and the value of the environment variable **MALLOC_CONF**, will be interpreted, in that order, from left to right as options. Note that *malloc_conf* may be read before main() is entered, so the declaration of *malloc_conf* should specify an initializer that contains the final value to be read by jemalloc. **--with-malloc-conf** and *malloc_conf* are compile-time mechanisms, whereas /etc/malloc.conf and **MALLOC_CONF** can be safely set any time prior to program invocation.

An options string is a comma-separated list of option:value pairs. There is one key corresponding to each opt.* mallctl (see the MALLCTL NAMESPACE section for options documentation). For example, abort:true,narenas:1 sets the opt.abort and opt.narenas options. Some options have boolean values (true/false), others have integer values (base 8, 10, or 16, depending on prefix), and yet others have raw string values.

IMPLEMENTATION NOTES

Traditionally, allocators have used **sbrk**(2) to obtain memory, which is suboptimal for several reasons, including race conditions, increased fragmentation, and artificial limitations on maximum usable memory. If **sbrk**(2) is supported by the operating system, this allocator uses both **mmap**(2) and **sbrk**(2), in that order of preference; otherwise only **mmap**(2) is used.

This allocator uses multiple arenas in order to reduce lock contention for threaded programs on multi-processor systems. This works well with regard to threading scalability, but incurs some costs. There is a small fixed per-arena overhead, and additionally, arenas manage memory completely independently of each other, which means a small fixed increase in overall memory fragmentation. These overheads are not generally an issue, given the number of arenas normally used. Note that using substantially more arenas than the default is not likely to improve performance, mainly due to reduced cache performance. However, it may make sense to reduce the number of arenas if an application does not make much use of the allocation functions.

In addition to multiple arenas, this allocator supports thread-specific caching, in order to make it possible to completely avoid synchronization for most allocation requests. Such caching allows very fast allocation in the common case, but it increases memory usage and fragmentation, since a bounded number of objects can remain allocated in each thread cache.

Memory is conceptually broken into extents. Extents are always aligned to multiples of the page size. This alignment makes it possible to find metadata for user objects quickly. User objects are broken into two categories according to size: small and large. Contiguous small objects comprise a slab, which resides within a single extent, whereas large objects each have their own extents backing them.

Small objects are managed in groups by slabs. Each slab maintains a bitmap to track which regions are in use. Allocation requests that are no more than half the quantum (8 or 16, depending on architecture) are rounded up to the nearest power of two that is at least sizeof(**double**). All other object size classes are multiples of the quantum, spaced such that there are four size classes for each doubling in size, which limits internal fragmentation to approximately 20% for all but the smallest size classes. Small size classes are smaller than four times the page size, and large size classes extend from four times the page size up to the largest size class that does not exceed **PTRDIFF_MAX**.

Allocations are packed tightly together, which can be an issue for multi-threaded applications. If you need to assure that allocations do not suffer from cacheline sharing, round your allocation requests up to the nearest multiple of the cacheline size, or specify cacheline alignment when allocating.

The realloc(), rallocx(), and xallocx() functions may resize allocations without moving them under limited circumstances. Unlike the *allocx() API, the standard API does not officially round up the usable size of an allocation to the nearest size class, so technically it is necessary to call realloc() to grow e.g. a 9-byte allocation to 16 bytes, or shrink a 16-byte allocation to 9 bytes. Growth and shrinkage trivially succeeds in place as long as the pre-size and post-size both round up to the same size class. No other API guarantees are made regarding in-place resizing, but the current implementation also tries to resize large allocations in place, as long as the pre-size and post-size are both large. For shrinkage to succeed, the extent allocator must support splitting (see arena.<i>.extent_hooks). Growth only succeeds if the trailing memory is currently available, and the extent allocator supports merging.

Assuming 4 KiB pages and a 16-byte quantum on a 64-bit system, the size classes in each category are as shown in Table 1.

Table 1. Size classes

Category	**Spacing**	**Size**
Small	lg	[8]
	16	[16, 32, 48, 64, 80, 96, 112, 128]

		32	[160, 192, 224, 256]
		64	[320, 384, 448, 512]
		128	[640, 768, 896, 1024]
		256	[1280, 1536, 1792, 2048]
		512	[2560, 3072, 3584, 4096]
	1 KiB		[5 KiB, 6 KiB, 7 KiB, 8 KiB]
	2 KiB		[10 KiB, 12 KiB, 14 KiB]
Large	2 KiB		[16 KiB]
	4 KiB		[20 KiB, 24 KiB, 28 KiB, 32 KiB]
	8 KiB		[40 KiB, 48 KiB, 54 KiB, 64 KiB]
	16 KiB		[80 KiB, 96 KiB, 112 KiB, 128 KiB]
	32 KiB		[160 KiB, 192 KiB, 224 KiB, 256 KiB]
	64 KiB		[320 KiB, 384 KiB, 448 KiB, 512 KiB]

	128 KiB	[640 KiB, 768 KiB, 896 KiB, 1 MiB]
	256 KiB	[1280 KiB, 1536 KiB, 1792 KiB, 2 MiB]
	512 KiB	[2560 KiB, 3 MiB, 3584 KiB, 4 MiB]
	1 MiB	[5 MiB, 6 MiB, 7 MiB, 8 MiB]
	2 MiB	[10 MiB, 12 MiB, 14 MiB, 16 MiB]
	4 MiB	[20 MiB, 24 MiB, 28 MiB, 32 MiB]
	8 MiB	[40 MiB, 48 MiB, 56 MiB, 64 MiB]
	...	...
	512 PiB	[2560 PiB, 3 EiB, 3584 PiB, 4 EiB]
	1 EiB	[5 EiB, 6 EiB, 7 EiB]

MALLCTL NAMESPACE

The following names are defined in the namespace accessible via the mallctl*() functions. Value types are specified in parentheses, their readable/writable statuses are encoded as rw, r-, -w, or --, and required build configuration flags follow, if any. A name element encoded as <i> or <j> indicates an integer component, where the integer varies from 0 to some upper value that must be determined via introspection. In the case of stats.arenas.<i>.* and arena.<i>.{initialized,purge,decay,dss}, <i> equal to **MALLCTL_ARENAS_ALL** can be used to operate on all arenas or access the summation of statistics from all arenas; similarly <i> equal to **MALLCTL_ARENAS_DESTROYED** can be used to access the summation of statistics from all destroyed arenas. These constants can be utilized either via mallctlnametomib() followed by

mallctlbymib(), or via code such as the following:

```
#define STRINGIFY_HELPER(x) #x
#define STRINGIFY(x) STRINGIFY_HELPER(x)

mallctl("arena." STRINGIFY(MALLCTL_ARENAS_ALL) ".decay",
  NULL, NULL, NULL, 0);
```

Take special note of the epoch mallctl, which controls refreshing of cached dynamic statistics.

version (**const char ***) r-
: Return the jemalloc version string.

epoch (**uint64_t**) rw
: If a value is passed in, refresh the data from which the mallctl*() functions report values, and increment the epoch. Return the current epoch. This is useful for detecting whether another thread caused a refresh.

background_thread (**bool**) rw
: Enable/disable internal background worker threads. When set to true, background threads are created on demand (the number of background threads will be no more than the number of CPUs or active arenas). Threads run periodically, and handle purging asynchronously. When switching off, background threads are terminated synchronously. Note that after **fork**(2) function, the state in the child process will be disabled regardless the state in parent process. See stats.background_thread for related stats. opt.background_thread can be used to set the default option. This option is only available on selected pthread-based platforms.

max_background_threads (**size_t**) rw
: Maximum number of background worker threads that will be created. This value is capped at opt.max_background_threads at startup.

config.cache_oblivious (**bool**) r-
: **--enable-cache-oblivious** was specified during build configuration.

config.debug (**bool**) r-
: **--enable-debug** was specified during build configuration.

config.fill (**bool**) r-
: **--enable-fill** was specified during build configuration.

config.lazy_lock (**bool**) r-
--enable-lazy-lock was specified during build configuration.

config.malloc_conf (**const char ***) r-
Embedded configure-time-specified run-time options string, empty unless **--with-malloc-conf** was specified during build configuration.

config.prof (**bool**) r-
--enable-prof was specified during build configuration.

config.prof_libgcc (**bool**) r-
--disable-prof-libgcc was not specified during build configuration.

config.prof_libunwind (**bool**) r-
--enable-prof-libunwind was specified during build configuration.

config.stats (**bool**) r-
--enable-stats was specified during build configuration.

config.utrace (**bool**) r-
--enable-utrace was specified during build configuration.

config.xmalloc (**bool**) r-
--enable-xmalloc was specified during build configuration.

opt.abort (**bool**) r-
Abort-on-warning enabled/disabled. If true, most warnings are fatal. Note that runtime option warnings are not included (see opt.abort_conf for that). The process will call **abort**(3) in these cases. This option is disabled by default unless **--enable-debug** is specified during configuration, in which case it is enabled by default.

opt.confirm_conf (**bool**) r-
Confirm-runtime-options-when-program-starts enabled/disabled. If true, the string specified via **--with-malloc-conf**, the string pointed to by the global variable *malloc_conf*, the "name" of the file referenced by the symbolic link named /etc/malloc.conf, and the value of the environment variable **MALLOC_CONF**, will be printed in order. Then, each option being set will be individually printed. This option is disabled by default.

opt.abort_conf (**bool**) r-
Abort-on-invalid-configuration enabled/disabled. If true, invalid runtime options are fatal. The

process will call **abort**(3) in these cases. This option is disabled by default unless **--enable-debug** is specified during configuration, in which case it is enabled by default.

opt.metadata_thp (**const char ***) r-
Controls whether to allow jemalloc to use transparent huge page (THP) for internal metadata (see stats.metadata). "always" allows such usage. "auto" uses no THP initially, but may begin to do so when metadata usage reaches certain level. The default is "disabled".

opt.retain (**bool**) r-
If true, retain unused virtual memory for later reuse rather than discarding it by calling **munmap**(2) or equivalent (see stats.retained for related details). It also makes jemalloc use **mmap**(2) or equivalent in a more greedy way, mapping larger chunks in one go. This option is disabled by default unless discarding virtual memory is known to trigger platform-specific performance problems, namely 1) for [64-bit] Linux, which has a quirk in its virtual memory allocation algorithm that causes semi-permanent VM map holes under normal jemalloc operation; and 2) for [64-bit] Windows, which disallows split / merged regions with **MEM_RELEASE**. Although the same issues may present on 32-bit platforms as well, retaining virtual memory for 32-bit Linux and Windows is disabled by default due to the practical possibility of address space exhaustion.

opt.dss (**const char ***) r-
dss (**sbrk**(2)) allocation precedence as related to **mmap**(2) allocation. The following settings are supported if **sbrk**(2) is supported by the operating system: "disabled", "primary", and "secondary"; otherwise only "disabled" is supported. The default is "secondary" if **sbrk**(2) is supported by the operating system; "disabled" otherwise.

opt.narenas (**unsigned**) r-
Maximum number of arenas to use for automatic multiplexing of threads and arenas. The default is four times the number of CPUs, or one if there is a single CPU.

opt.oversize_threshold (**size_t**) r-
The threshold in bytes of which requests are considered oversize. Allocation requests with greater sizes are fulfilled from a dedicated arena (automatically managed, however not within narenas), in order to reduce fragmentation by not mixing huge allocations with small ones. In addition, the decay API guarantees on the extents greater than the specified threshold may be overridden. Note that requests with arena index specified via **MALLOCX_ARENA**, or threads associated with explicit arenas will not be considered. The default threshold is 8MiB. Values not within large size classes disables this feature.

opt.percpu_arena (**const char ***) r-
Per CPU arena mode. Use the "percpu" setting to enable this feature, which uses number of CPUs

to determine number of arenas, and bind threads to arenas dynamically based on the CPU the thread runs on currently. "phycpu" setting uses one arena per physical CPU, which means the two hyper threads on the same CPU share one arena. Note that no runtime checking regarding the availability of hyper threading is done at the moment. When set to "disabled", narenas and thread to arena association will not be impacted by this option. The default is "disabled".

opt.background_thread (**bool**) r-

Internal background worker threads enabled/disabled. Because of potential circular dependencies, enabling background thread using this option may cause crash or deadlock during initialization. For a reliable way to use this feature, see background_thread for dynamic control options and details. This option is disabled by default.

opt.max_background_threads (**size_t**) r-

Maximum number of background threads that will be created if background_thread is set. Defaults to number of cpus.

opt.dirty_decay_ms (**ssize_t**) r-

Approximate time in milliseconds from the creation of a set of unused dirty pages until an equivalent set of unused dirty pages is purged (i.e. converted to muzzy via e.g. madvise(...**MADV_FREE**) if supported by the operating system, or converted to clean otherwise) and/or reused. Dirty pages are defined as previously having been potentially written to by the application, and therefore consuming physical memory, yet having no current use. The pages are incrementally purged according to a sigmoidal decay curve that starts and ends with zero purge rate. A decay time of 0 causes all unused dirty pages to be purged immediately upon creation. A decay time of -1 disables purging. The default decay time is 10 seconds. See arenas.dirty_decay_ms and arena.<i>.dirty_decay_ms for related dynamic control options. See opt.muzzy_decay_ms for a description of muzzy pages.for a description of muzzy pages. Note that when the oversize_threshold feature is enabled, the arenas reserved for oversize requests may have its own default decay settings.

opt.muzzy_decay_ms (**ssize_t**) r-

Approximate time in milliseconds from the creation of a set of unused muzzy pages until an equivalent set of unused muzzy pages is purged (i.e. converted to clean) and/or reused. Muzzy pages are defined as previously having been unused dirty pages that were subsequently purged in a manner that left them subject to the reclamation whims of the operating system (e.g. madvise(...**MADV_FREE**)), and therefore in an indeterminate state. The pages are incrementally purged according to a sigmoidal decay curve that starts and ends with zero purge rate. A decay time of 0 causes all unused muzzy pages to be purged immediately upon creation. A decay time of -1 disables purging. The default decay time is 10 seconds. See arenas.muzzy_decay_ms and arena.<i>.muzzy_decay_ms for related dynamic control options.

opt.lg_extent_max_active_fit (**size_t**) r-

When reusing dirty extents, this determines the (log base 2 of the) maximum ratio between the size of the active extent selected (to split off from) and the size of the requested allocation. This prevents the splitting of large active extents for smaller allocations, which can reduce fragmentation over the long run (especially for non-active extents). Lower value may reduce fragmentation, at the cost of extra active extents. The default value is 6, which gives a maximum ratio of 64 (2^6).

opt.stats_print (**bool**) r-

Enable/disable statistics printing at exit. If enabled, the malloc_stats_print() function is called at program exit via an **atexit**(3) function. opt.stats_print_opts can be combined to specify output options. If **--enable-stats** is specified during configuration, this has the potential to cause deadlock for a multi-threaded process that exits while one or more threads are executing in the memory allocation functions. Furthermore, atexit() may allocate memory during application initialization and then deadlock internally when jemalloc in turn calls atexit(), so this option is not universally usable (though the application can register its own atexit() function with equivalent functionality). Therefore, this option should only be used with care; it is primarily intended as a performance tuning aid during application development. This option is disabled by default.

opt.stats_print_opts (**const char ***) r-

Options (the *opts* string) to pass to the malloc_stats_print() at exit (enabled through opt.stats_print). See available options in malloc_stats_print(). Has no effect unless opt.stats_print is enabled. The default is "".

opt.junk (**const char ***) r- [**--enable-fill**]

Junk filling. If set to "alloc", each byte of uninitialized allocated memory will be initialized to 0xa5. If set to "free", all deallocated memory will be initialized to 0x5a. If set to "true", both allocated and deallocated memory will be initialized, and if set to "false", junk filling be disabled entirely. This is intended for debugging and will impact performance negatively. This option is "false" by default unless **--enable-debug** is specified during configuration, in which case it is "true" by default.

opt.zero (**bool**) r- [**--enable-fill**]

Zero filling enabled/disabled. If enabled, each byte of uninitialized allocated memory will be initialized to 0. Note that this initialization only happens once for each byte, so realloc() and rallocx() calls do not zero memory that was previously allocated. This is intended for debugging and will impact performance negatively. This option is disabled by default.

opt.utrace (**bool**) r- [**--enable-utrace**]

Allocation tracing based on **utrace**(2) enabled/disabled. This option is disabled by default.

opt.xmalloc (**bool**) r- [**--enable-xmalloc**]

Abort-on-out-of-memory enabled/disabled. If enabled, rather than returning failure for any allocation function, display a diagnostic message on **STDERR_FILENO** and cause the program to drop core (using **abort**(3)). If an application is designed to depend on this behavior, set the option at compile time by including the following in the source code:

```
malloc_conf = "xmalloc:true";
```

This option is disabled by default.

opt.tcache (**bool**) r-

Thread-specific caching (tcache) enabled/disabled. When there are multiple threads, each thread uses a tcache for objects up to a certain size. Thread-specific caching allows many allocations to be satisfied without performing any thread synchronization, at the cost of increased memory use. See the opt.lg_tcache_max option for related tuning information. This option is enabled by default.

opt.lg_tcache_max (**size_t**) r-

Maximum size class (log base 2) to cache in the thread-specific cache (tcache). At a minimum, all small size classes are cached, and at a maximum all large size classes are cached. The default maximum is 32 KiB (2^15).

opt.thp (**const char ***) r-

Transparent hugepage (THP) mode. Settings "always", "never" and "default" are available if THP is supported by the operating system. The "always" setting enables transparent hugepage for all user memory mappings with **MADV_HUGEPAGE**; "never" ensures no transparent hugepage with **MADV_NOHUGEPAGE**; the default setting "default" makes no changes. Note that: this option does not affect THP for jemalloc internal metadata (see opt.metadata_thp); in addition, for arenas with customized extent_hooks, this option is bypassed as it is implemented as part of the default extent hooks.

opt.prof (**bool**) r- [**--enable-prof**]

Memory profiling enabled/disabled. If enabled, profile memory allocation activity. See the opt.prof_active option for on-the-fly activation/deactivation. See the opt.lg_prof_sample option for probabilistic sampling control. See the opt.prof_accum option for control of cumulative sample reporting. See the opt.lg_prof_interval option for information on interval-triggered profile dumping, the opt.prof_gdump option for information on high-water-triggered profile dumping, and the opt.prof_final option for final profile dumping. Profile output is compatible with the **jeprof** command, which is based on the **pprof** that is developed as part of the **gperftools package**[3]. See HEAP PROFILE FORMAT for heap profile format documentation.

opt.prof_prefix (**const char ***) r- [**--enable-prof**]
Filename prefix for profile dumps. If the prefix is set to the empty string, no automatic dumps will occur; this is primarily useful for disabling the automatic final heap dump (which also disables leak reporting, if enabled). The default prefix is jeprof.

opt.prof_active (**bool**) r- [**--enable-prof**]
Profiling activated/deactivated. This is a secondary control mechanism that makes it possible to start the application with profiling enabled (see the opt.prof option) but inactive, then toggle profiling at any time during program execution with the prof.active mallctl. This option is enabled by default.

opt.prof_thread_active_init (**bool**) r- [**--enable-prof**]
Initial setting for thread.prof.active in newly created threads. The initial setting for newly created threads can also be changed during execution via the prof.thread_active_init mallctl. This option is enabled by default.

opt.lg_prof_sample (**size_t**) r- [**--enable-prof**]
Average interval (log base 2) between allocation samples, as measured in bytes of allocation activity. Increasing the sampling interval decreases profile fidelity, but also decreases the computational overhead. The default sample interval is 512 KiB (2^19 B).

opt.prof_accum (**bool**) r- [**--enable-prof**]
Reporting of cumulative object/byte counts in profile dumps enabled/disabled. If this option is enabled, every unique backtrace must be stored for the duration of execution. Depending on the application, this can impose a large memory overhead, and the cumulative counts are not always of interest. This option is disabled by default.

opt.lg_prof_interval (**ssize_t**) r- [**--enable-prof**]
Average interval (log base 2) between memory profile dumps, as measured in bytes of allocation activity. The actual interval between dumps may be sporadic because decentralized allocation counters are used to avoid synchronization bottlenecks. Profiles are dumped to files named according to the pattern <prefix>.<pid>.<seq>.i<iseq>.heap, where <prefix> is controlled by the opt.prof_prefix option. By default, interval-triggered profile dumping is disabled (encoded as -1).

opt.prof_gdump (**bool**) r- [**--enable-prof**]
Set the initial state of prof.gdump, which when enabled triggers a memory profile dump every time the total virtual memory exceeds the previous maximum. This option is disabled by default.

opt.prof_final (**bool**) r- [**--enable-prof**]
Use an **atexit**(3) function to dump final memory usage to a file named according to the pattern

<prefix>.<pid>.<seq>.f.heap, where <prefix> is controlled by the opt.prof_prefix option. Note that atexit() may allocate memory during application initialization and then deadlock internally when jemalloc in turn calls atexit(), so this option is not universally usable (though the application can register its own atexit() function with equivalent functionality). This option is disabled by default.

opt.prof_leak (**bool**) r- [**--enable-prof**]
Leak reporting enabled/disabled. If enabled, use an **atexit**(3) function to report memory leaks detected by allocation sampling. See the opt.prof option for information on analyzing heap profile output. This option is disabled by default.

thread.arena (**unsigned**) rw
Get or set the arena associated with the calling thread. If the specified arena was not initialized beforehand (see the arena.i.initialized mallctl), it will be automatically initialized as a side effect of calling this interface.

thread.allocated (**uint64_t**) r- [**--enable-stats**]
Get the total number of bytes ever allocated by the calling thread. This counter has the potential to wrap around; it is up to the application to appropriately interpret the counter in such cases.

thread.allocatedp (**uint64_t ***) r- [**--enable-stats**]
Get a pointer to the the value that is returned by the thread.allocated mallctl. This is useful for avoiding the overhead of repeated mallctl*() calls.

thread.deallocated (**uint64_t**) r- [**--enable-stats**]
Get the total number of bytes ever deallocated by the calling thread. This counter has the potential to wrap around; it is up to the application to appropriately interpret the counter in such cases.

thread.deallocatedp (**uint64_t ***) r- [**--enable-stats**]
Get a pointer to the the value that is returned by the thread.deallocated mallctl. This is useful for avoiding the overhead of repeated mallctl*() calls.

thread.tcache.enabled (**bool**) rw
Enable/disable calling thread’s tcache. The tcache is implicitly flushed as a side effect of becoming disabled (see thread.tcache.flush).

thread.tcache.flush (**void**) --
Flush calling thread’s thread-specific cache (tcache). This interface releases all cached objects and internal data structures associated with the calling thread’s tcache. Ordinarily, this interface need not be called, since automatic periodic incremental garbage collection occurs, and the thread cache

is automatically discarded when a thread exits. However, garbage collection is triggered by allocation activity, so it is possible for a thread that stops allocating/deallocating to retain its cache indefinitely, in which case the developer may find manual flushing useful.

thread.prof.name (**const char ***) r- or -w [**--enable-prof**]
Get/set the descriptive name associated with the calling thread in memory profile dumps. An internal copy of the name string is created, so the input string need not be maintained after this interface completes execution. The output string of this interface should be copied for non-ephemeral uses, because multiple implementation details can cause asynchronous string deallocation. Furthermore, each invocation of this interface can only read or write; simultaneous read/write is not supported due to string lifetime limitations. The name string must be nil-terminated and comprised only of characters in the sets recognized by **isgraph**(3) and **isblank**(3).

thread.prof.active (**bool**) rw [**--enable-prof**]
Control whether sampling is currently active for the calling thread. This is an activation mechanism in addition to prof.active; both must be active for the calling thread to sample. This flag is enabled by default.

tcache.create (**unsigned**) r-
Create an explicit thread-specific cache (tcache) and return an identifier that can be passed to the **MALLOCX_TCACHE(***tc***)** macro to explicitly use the specified cache rather than the automatically managed one that is used by default. Each explicit cache can be used by only one thread at a time; the application must assure that this constraint holds.

tcache.flush (**unsigned**) -w
Flush the specified thread-specific cache (tcache). The same considerations apply to this interface as to thread.tcache.flush, except that the tcache will never be automatically discarded.

tcache.destroy (**unsigned**) -w
Flush the specified thread-specific cache (tcache) and make the identifier available for use during a future tcache creation.

arena.<i>.initialized (**bool**) r-
Get whether the specified arena’s statistics are initialized (i.e. the arena was initialized prior to the current epoch). This interface can also be nominally used to query whether the merged statistics corresponding to **MALLCTL_ARENAS_ALL** are initialized (always true).

arena.<i>.decay (**void**) --
Trigger decay-based purging of unused dirty/muzzy pages for arena <i>, or for all arenas if <i>

equals **MALLCTL_ARENAS_ALL**. The proportion of unused dirty/muzzy pages to be purged depends on the current time; see opt.dirty_decay_ms and opt.muzy_decay_ms for details.

arena.<i>.purge (**void**) --
Purge all unused dirty pages for arena <i>, or for all arenas if <i> equals **MALLCTL_ARENAS_ALL**.

arena.<i>.reset (**void**) --
Discard all of the arena's extant allocations. This interface can only be used with arenas explicitly created via arenas.create. None of the arena's discarded/cached allocations may accessed afterward. As part of this requirement, all thread caches which were used to allocate/deallocate in conjunction with the arena must be flushed beforehand.

arena.<i>.destroy (**void**) --
Destroy the arena. Discard all of the arena's extant allocations using the same mechanism as for arena.<i>.reset (with all the same constraints and side effects), merge the arena stats into those accessible at arena index **MALLCTL_ARENAS_DESTROYED**, and then completely discard all metadata associated with the arena. Future calls to arenas.create may recycle the arena index. Destruction will fail if any threads are currently associated with the arena as a result of calls to thread.arena.

arena.<i>.dss (**const char ***) rw
Set the precedence of dss allocation as related to mmap allocation for arena <i>, or for all arenas if <i> equals **MALLCTL_ARENAS_ALL**. See opt.dss for supported settings.

arena.<i>.dirty_decay_ms (**ssize_t**) rw
Current per-arena approximate time in milliseconds from the creation of a set of unused dirty pages until an equivalent set of unused dirty pages is purged and/or reused. Each time this interface is set, all currently unused dirty pages are considered to have fully decayed, which causes immediate purging of all unused dirty pages unless the decay time is set to -1 (i.e. purging disabled). See opt.dirty_decay_ms for additional information.

arena.<i>.muzzy_decay_ms (**ssize_t**) rw
Current per-arena approximate time in milliseconds from the creation of a set of unused muzzy pages until an equivalent set of unused muzzy pages is purged and/or reused. Each time this interface is set, all currently unused muzzy pages are considered to have fully decayed, which causes immediate purging of all unused muzzy pages unless the decay time is set to -1 (i.e. purging disabled). See opt.muzzy_decay_ms for additional information.

arena.<i>.retain_grow_limit (**size_t**) rw

Maximum size to grow retained region (only relevant when opt.retain is enabled). This controls the maximum increment to expand virtual memory, or allocation through arena.<i>extent_hooks. In particular, if customized extent hooks reserve physical memory (e.g. 1G huge pages), this is useful to control the allocation hook's input size. The default is no limit.

arena.<i>.extent_hooks (**extent_hooks_t ***) rw

Get or set the extent management hook functions for arena <i>. The functions must be capable of operating on all extant extents associated with arena <i>, usually by passing unknown extents to the replaced functions. In practice, it is feasible to control allocation for arenas explicitly created via arenas.create such that all extents originate from an application-supplied extent allocator (by specifying the custom extent hook functions during arena creation). However, the API guarantees for the automatically created arenas may be relaxed -- hooks set there may be called in a "best effort" fashion; in addition there may be extents created prior to the application having an opportunity to take over extent allocation.

```
typedef extent_hooks_s extent_hooks_t;
struct extent_hooks_s {
        extent_alloc_t          *alloc;
        extent_dalloc_t         *dalloc;
        extent_destroy_t        *destroy;
        extent_commit_t                 *commit;
        extent_decommit_t       *decommit;
        extent_purge_t          *purge_lazy;
        extent_purge_t          *purge_forced;
        extent_split_t          *split;
        extent_merge_t          *merge;
};
```

The **extent_hooks_t** structure comprises function pointers which are described individually below. jemalloc uses these functions to manage extent lifetime, which starts off with allocation of mapped committed memory, in the simplest case followed by deallocation. However, there are performance and platform reasons to retain extents for later reuse. Cleanup attempts cascade from deallocation to decommit to forced purging to lazy purging, which gives the extent management functions opportunities to reject the most permanent cleanup operations in favor of less permanent (and often less costly) operations. All operations except allocation can be universally opted out of by setting the hook pointers to **NULL**, or selectively opted out of by returning failure. Note that once the extent hook is set, the structure is accessed directly by the associated arenas, so it must remain valid for the entire lifetime of the arenas.

typedef void *(extent_alloc_t)(extent_hooks_t ***extent_hooks*, void ***new_addr***, size_t** *size***,**

size_t *alignment*, **bool ***zero*, **bool ***commit*,
unsigned *arena_ind*);

An extent allocation function conforms to the **extent_alloc_t** type and upon success returns a pointer to *size* bytes of mapped memory on behalf of arena *arena_ind* such that the extent's base address is a multiple of *alignment*, as well as setting **zero* to indicate whether the extent is zeroed and **commit* to indicate whether the extent is committed. Upon error the function returns **NULL** and leaves **zero* and **commit* unmodified. The *size* parameter is always a multiple of the page size. The *alignment* parameter is always a power of two at least as large as the page size. Zeroing is mandatory if **zero* is true upon function entry. Committing is mandatory if **commit* is true upon function entry. If *new_addr* is not **NULL**, the returned pointer must be *new_addr* on success or **NULL** on error. Committed memory may be committed in absolute terms as on a system that does not overcommit, or in implicit terms as on a system that overcommits and satisfies physical memory needs on demand via soft page faults. Note that replacing the default extent allocation function makes the arena's arena.<i>.dss setting irrelevant.

typedef bool (extent_dalloc_t)(extent_hooks_t ***extent_hooks*, **void ***addr*, **size_t *size*,
bool *committed*, **unsigned** *arena_ind*);

An extent deallocation function conforms to the **extent_dalloc_t** type and deallocates an extent at given *addr* and *size* with *committed*/decommited memory as indicated, on behalf of arena *arena_ind*, returning false upon success. If the function returns true, this indicates opt-out from deallocation; the virtual memory mapping associated with the extent remains mapped, in the same commit state, and available for future use, in which case it will be automatically retained for later reuse.

typedef void (extent_destroy_t)(extent_hooks_t ***extent_hooks*, **void ***addr*, **size_t *size*,
bool *committed*, **unsigned** *arena_ind*);

An extent destruction function conforms to the **extent_destroy_t** type and unconditionally destroys an extent at given *addr* and *size* with *committed*/decommited memory as indicated, on behalf of arena *arena_ind*. This function may be called to destroy retained extents during arena destruction (see arena.<i>.destroy).

typedef bool (extent_commit_t)(extent_hooks_t ***extent_hooks*, **void ***addr*, **size_t *size*,
size_t *offset*, **size_t** *length*,
unsigned *arena_ind*);

An extent commit function conforms to the **extent_commit_t** type and commits zeroed physical memory to back pages within an extent at given *addr* and *size* at *offset* bytes, extending for *length* on behalf of arena *arena_ind*, returning false upon success. Committed memory may be committed in absolute terms as on a system that does not overcommit, or in implicit terms as on a system that overcommits and satisfies physical memory needs on demand via soft page faults. If the function returns true, this indicates insufficient physical memory to satisfy the request.

typedef bool (extent_decommit_t)(extent_hooks_t **extent_hooks*, void **addr*, size_t *size*, size_t *offset*, size_t *length*, unsigned *arena_ind*);

An extent decommit function conforms to the **extent_decommit_t** type and decommits any physical memory that is backing pages within an extent at given *addr* and *size* at *offset* bytes, extending for *length* on behalf of arena *arena_ind*, returning false upon success, in which case the pages will be committed via the extent commit function before being reused. If the function returns true, this indicates opt-out from decommit; the memory remains committed and available for future use, in which case it will be automatically retained for later reuse.

typedef bool (extent_purge_t)(extent_hooks_t **extent_hooks*, void **addr*, size_t *size*, size_t *offset*, size_t *length*, unsigned *arena_ind*);

An extent purge function conforms to the **extent_purge_t** type and discards physical pages within the virtual memory mapping associated with an extent at given *addr* and *size* at *offset* bytes, extending for *length* on behalf of arena *arena_ind*. A lazy extent purge function (e.g. implemented via madvise(...**MADV_FREE**)) can delay purging indefinitely and leave the pages within the purged virtual memory range in an indeterminite state, whereas a forced extent purge function immediately purges, and the pages within the virtual memory range will be zero-filled the next time they are accessed. If the function returns true, this indicates failure to purge.

typedef bool (extent_split_t)(extent_hooks_t **extent_hooks*, void **addr*, size_t *size*, size_t *size_a*, size_t *size_b*, bool *committed*, unsigned *arena_ind*);

An extent split function conforms to the **extent_split_t** type and optionally splits an extent at given *addr* and *size* into two adjacent extents, the first of *size_a* bytes, and the second of *size_b* bytes, operating on *committed*/decommitted memory as indicated, on behalf of arena *arena_ind*, returning false upon success. If the function returns true, this indicates that the extent remains

unsplit and therefore should continue to be operated on as a whole.

typedef bool (extent_merge_t)(extent_hooks_t ****extent_hooks*, void ****addr_a***, size_t** *size_a***,**
void ****addr_b*, size_t** *size_b***, bool** *committed***,**
unsigned *arena_ind***);**

An extent merge function conforms to the **extent_merge_t** type and optionally merges adjacent extents, at given *addr_a* and *size_a* with given *addr_b* and *size_b* into one contiguous extent, operating on *committed*/decommitted memory as indicated, on behalf of arena *arena_ind*, returning false upon success. If the function returns true, this indicates that the extents remain distinct mappings and therefore should continue to be operated on independently.

arenas.narenas (**unsigned**) r-
Current limit on number of arenas.

arenas.dirty_decay_ms (**ssize_t**) rw
Current default per-arena approximate time in milliseconds from the creation of a set of unused dirty pages until an equivalent set of unused dirty pages is purged and/or reused, used to initialize arena.<i>.dirty_decay_ms during arena creation. See opt.dirty_decay_ms for additional information.

arenas.muzzy_decay_ms (**ssize_t**) rw
Current default per-arena approximate time in milliseconds from the creation of a set of unused muzzy pages until an equivalent set of unused muzzy pages is purged and/or reused, used to initialize arena.<i>.muzzy_decay_ms during arena creation. See opt.muzzy_decay_ms for additional information.

arenas.quantum (**size_t**) r-
Quantum size.

arenas.page (**size_t**) r-
Page size.

arenas.tcache_max (**size_t**) r-
Maximum thread-cached size class.

arenas.nbins (**unsigned**) r-
Number of bin size classes.

arenas.nhbins (**unsigned**) r-
Total number of thread cache bin size classes.

arenas.bin.<i>.size (**size_t**) r-
Maximum size supported by size class.

arenas.bin.<i>.nregs (**uint32_t**) r-
Number of regions per slab.

arenas.bin.<i>.slab_size (**size_t**) r-
Number of bytes per slab.

arenas.nlextents (**unsigned**) r-
Total number of large size classes.

arenas.lextent.<i>.size (**size_t**) r-
Maximum size supported by this large size class.

arenas.create (**unsigned**, **extent_hooks_t ***) rw
Explicitly create a new arena outside the range of automatically managed arenas, with optionally specified extent hooks, and return the new arena index.

arenas.lookup (**unsigned**, **void***) rw
Index of the arena to which an allocation belongs to.

prof.thread_active_init (**bool**) rw [**--enable-prof**]
Control the initial setting for thread.prof.active in newly created threads. See the opt.prof_thread_active_init option for additional information.

prof.active (**bool**) rw [**--enable-prof**]
Control whether sampling is currently active. See the opt.prof_active option for additional information, as well as the interrelated thread.prof.active mallctl.

prof.dump (**const char ***) -w [**--enable-prof**]
Dump a memory profile to the specified file, or if NULL is specified, to a file according to the pattern <prefix>.<pid>.<seq>.m<mseq>.heap, where <prefix> is controlled by the opt.prof_prefix option.

prof.gdump (**bool**) rw [**--enable-prof**]
When enabled, trigger a memory profile dump every time the total virtual memory exceeds the

previous maximum. Profiles are dumped to files named according to the pattern <prefix>.<pid>.<seq>.u<useq>.heap, where <prefix> is controlled by the opt.prof_prefix option.

prof.reset (**size_t**) -w [**--enable-prof**]
Reset all memory profile statistics, and optionally update the sample rate (see opt.lg_prof_sample and prof.lg_sample).

prof.lg_sample (**size_t**) r- [**--enable-prof**]
Get the current sample rate (see opt.lg_prof_sample).

prof.interval (**uint64_t**) r- [**--enable-prof**]
Average number of bytes allocated between interval-based profile dumps. See the opt.lg_prof_interval option for additional information.

stats.allocated (**size_t**) r- [**--enable-stats**]
Total number of bytes allocated by the application.

stats.active (**size_t**) r- [**--enable-stats**]
Total number of bytes in active pages allocated by the application. This is a multiple of the page size, and greater than or equal to stats.allocated. This does not include stats.arenas.<i>.pdirty, stats.arenas.<i>.pmuzzy, nor pages entirely devoted to allocator metadata.

stats.metadata (**size_t**) r- [**--enable-stats**]
Total number of bytes dedicated to metadata, which comprise base allocations used for bootstrap-sensitive allocator metadata structures (see stats.arenas.<i>.base) and internal allocations (see stats.arenas.<i>.internal). Transparent huge page (enabled with opt.metadata_thp) usage is not considered.

stats.metadata_thp (**size_t**) r- [**--enable-stats**]
Number of transparent huge pages (THP) used for metadata. See stats.metadata and opt.metadata_thp) for details.

stats.resident (**size_t**) r- [**--enable-stats**]
Maximum number of bytes in physically resident data pages mapped by the allocator, comprising all pages dedicated to allocator metadata, pages backing active allocations, and unused dirty pages. This is a maximum rather than precise because pages may not actually be physically resident if they correspond to demand-zeroed virtual memory that has not yet been touched. This is a multiple of the page size, and is larger than stats.active.

stats.mapped (**size_t**) r- [**--enable-stats**]

Total number of bytes in active extents mapped by the allocator. This is larger than stats.active. This does not include inactive extents, even those that contain unused dirty pages, which means that there is no strict ordering between this and stats.resident.

stats.retained (**size_t**) r- [**--enable-stats**]
Total number of bytes in virtual memory mappings that were retained rather than being returned to the operating system via e.g. **munmap**(2) or similar. Retained virtual memory is typically untouched, decommitted, or purged, so it has no strongly associated physical memory (see extent hooks for details). Retained memory is excluded from mapped memory statistics, e.g. stats.mapped.

stats.background_thread.num_threads (**size_t**) r- [**--enable-stats**]
Number of background threads running currently.

stats.background_thread.num_runs (**uint64_t**) r- [**--enable-stats**]
Total number of runs from all background threads.

stats.background_thread.run_interval (**uint64_t**) r- [**--enable-stats**]
Average run interval in nanoseconds of background threads.

stats.mutexes.ctl.{counter}; (**counter specific type**) r- [**--enable-stats**]
Statistics on *ctl* mutex (global scope; mallctl related). {counter} is one of the counters below:

num_ops (**uint64_t**): Total number of lock acquisition operations on this mutex.

num_spin_acq (**uint64_t**): Number of times the mutex was spin-acquired. When the mutex is currently locked and cannot be acquired immediately, a short period of spin-retry within jemalloc will be performed. Acquired through spin generally means the contention was lightweight and not causing context switches.

num_wait (**uint64_t**): Number of times the mutex was wait-acquired, which means the mutex contention was not solved by spin-retry, and blocking operation was likely involved in order to acquire the mutex. This event generally implies higher cost / longer delay, and should be investigated if it happens often.

max_wait_time (**uint64_t**): Maximum length of time in nanoseconds spent on a single wait-acquired lock operation. Note that to avoid profiling overhead on the common path, this does not consider spin-acquired cases.

total_wait_time (**uint64_t**): Cumulative time in nanoseconds spent on wait-acquired lock

operations. Similarly, spin-acquired cases are not considered.

max_num_thds (**uint32_t**): Maximum number of threads waiting on this mutex simultaneously. Similarly, spin-acquired cases are not considered.

num_owner_switch (**uint64_t**): Number of times the current mutex owner is different from the previous one. This event does not generally imply an issue; rather it is an indicator of how often the protected data are accessed by different threads.

stats.mutexes.background_thread.{counter} (**counter specific type**) r- [**--enable-stats**]
Statistics on *background_thread* mutex (global scope; background_thread related). {counter} is one of the counters in mutex profiling counters.

stats.mutexes.prof.{counter} (**counter specific type**) r- [**--enable-stats**]
Statistics on *prof* mutex (global scope; profiling related). {counter} is one of the counters in mutex profiling counters.

stats.mutexes.reset (**void**) -- [**--enable-stats**]
Reset all mutex profile statistics, including global mutexes, arena mutexes and bin mutexes.

stats.arenas.<i>.dss (**const char ***) r-
dss (**sbrk**(2)) allocation precedence as related to **mmap**(2) allocation. See opt.dss for details.

stats.arenas.<i>.dirty_decay_ms (**ssize_t**) r-
Approximate time in milliseconds from the creation of a set of unused dirty pages until an equivalent set of unused dirty pages is purged and/or reused. See opt.dirty_decay_ms for details.

stats.arenas.<i>.muzzy_decay_ms (**ssize_t**) r-
Approximate time in milliseconds from the creation of a set of unused muzzy pages until an equivalent set of unused muzzy pages is purged and/or reused. See opt.muzzy_decay_ms for details.

stats.arenas.<i>.nthreads (**unsigned**) r-
Number of threads currently assigned to arena.

stats.arenas.<i>.uptime (**uint64_t**) r-
Time elapsed (in nanoseconds) since the arena was created. If <i> equals **0** or **MALLCTL_ARENAS_ALL**, this is the uptime since malloc initialization.

stats.arenas.<i>.pactive (**size_t**) r-

Number of pages in active extents.

stats.arenas.<i>.pdirty (**size_t**) r-
Number of pages within unused extents that are potentially dirty, and for which madvise() or similar has not been called. See opt.dirty_decay_ms for a description of dirty pages.

stats.arenas.<i>.pmuzzy (**size_t**) r-
Number of pages within unused extents that are muzzy. See opt.muzzy_decay_ms for a description of muzzy pages.

stats.arenas.<i>.mapped (**size_t**) r- [**--enable-stats**]
Number of mapped bytes.

stats.arenas.<i>.retained (**size_t**) r- [**--enable-stats**]
Number of retained bytes. See stats.retained for details.

stats.arenas.<i>.extent_avail (**size_t**) r- [**--enable-stats**]
Number of allocated (but unused) extent structs in this arena.

stats.arenas.<i>.base (**size_t**) r- [**--enable-stats**]
Number of bytes dedicated to bootstrap-sensitive allocator metadata structures.

stats.arenas.<i>.internal (**size_t**) r- [**--enable-stats**]
Number of bytes dedicated to internal allocations. Internal allocations differ from application-originated allocations in that they are for internal use, and that they are omitted from heap profiles.

stats.arenas.<i>.metadata_thp (**size_t**) r- [**--enable-stats**]
Number of transparent huge pages (THP) used for metadata. See opt.metadata_thp for details.

stats.arenas.<i>.resident (**size_t**) r- [**--enable-stats**]
Maximum number of bytes in physically resident data pages mapped by the arena, comprising all pages dedicated to allocator metadata, pages backing active allocations, and unused dirty pages. This is a maximum rather than precise because pages may not actually be physically resident if they correspond to demand-zeroed virtual memory that has not yet been touched. This is a multiple of the page size.

stats.arenas.<i>.dirty_npurge (**uint64_t**) r- [**--enable-stats**]
Number of dirty page purge sweeps performed.

stats.arenas.<i>.dirty_nmadvise (**uint64_t**) r- [**--enable-stats**]
Number of madvise() or similar calls made to purge dirty pages.

stats.arenas.<i>.dirty_purged (**uint64_t**) r- [**--enable-stats**]
Number of dirty pages purged.

stats.arenas.<i>.muzzy_npurge (**uint64_t**) r- [**--enable-stats**]
Number of muzzy page purge sweeps performed.

stats.arenas.<i>.muzzy_nmadvise (**uint64_t**) r- [**--enable-stats**]
Number of madvise() or similar calls made to purge muzzy pages.

stats.arenas.<i>.muzzy_purged (**uint64_t**) r- [**--enable-stats**]
Number of muzzy pages purged.

stats.arenas.<i>.small.allocated (**size_t**) r- [**--enable-stats**]
Number of bytes currently allocated by small objects.

stats.arenas.<i>.small.nmalloc (**uint64_t**) r- [**--enable-stats**]
Cumulative number of times a small allocation was requested from the arena’s bins, whether to fill the relevant tcache if opt.tcache is enabled, or to directly satisfy an allocation request otherwise.

stats.arenas.<i>.small.ndalloc (**uint64_t**) r- [**--enable-stats**]
Cumulative number of times a small allocation was returned to the arena’s bins, whether to flush the relevant tcache if opt.tcache is enabled, or to directly deallocate an allocation otherwise.

stats.arenas.<i>.small.nrequests (**uint64_t**) r- [**--enable-stats**]
Cumulative number of allocation requests satisfied by all bin size classes.

stats.arenas.<i>.small.nfills (**uint64_t**) r- [**--enable-stats**]
Cumulative number of tcache fills by all small size classes.

stats.arenas.<i>.small.nflushes (**uint64_t**) r- [**--enable-stats**]
Cumulative number of tcache flushes by all small size classes.

stats.arenas.<i>.large.allocated (**size_t**) r- [**--enable-stats**]
Number of bytes currently allocated by large objects.

stats.arenas.<i>.large.nmalloc (**uint64_t**) r- [**--enable-stats**]

Cumulative number of times a large extent was allocated from the arena, whether to fill the relevant tcache if opt.tcache is enabled and the size class is within the range being cached, or to directly satisfy an allocation request otherwise.

stats.arenas.<i>.large.ndalloc (**uint64_t**) r- [**--enable-stats**]
Cumulative number of times a large extent was returned to the arena, whether to flush the relevant tcache if opt.tcache is enabled and the size class is within the range being cached, or to directly deallocate an allocation otherwise.

stats.arenas.<i>.large.nrequests (**uint64_t**) r- [**--enable-stats**]
Cumulative number of allocation requests satisfied by all large size classes.

stats.arenas.<i>.large.nfills (**uint64_t**) r- [**--enable-stats**]
Cumulative number of tcache fills by all large size classes.

stats.arenas.<i>.large.nflushes (**uint64_t**) r- [**--enable-stats**]
Cumulative number of tcache flushes by all large size classes.

stats.arenas.<i>.bins.<j>.nmalloc (**uint64_t**) r- [**--enable-stats**]
Cumulative number of times a bin region of the corresponding size class was allocated from the arena, whether to fill the relevant tcache if opt.tcache is enabled, or to directly satisfy an allocation request otherwise.

stats.arenas.<i>.bins.<j>.ndalloc (**uint64_t**) r- [**--enable-stats**]
Cumulative number of times a bin region of the corresponding size class was returned to the arena, whether to flush the relevant tcache if opt.tcache is enabled, or to directly deallocate an allocation otherwise.

stats.arenas.<i>.bins.<j>.nrequests (**uint64_t**) r- [**--enable-stats**]
Cumulative number of allocation requests satisfied by bin regions of the corresponding size class.

stats.arenas.<i>.bins.<j>.curregs (**size_t**) r- [**--enable-stats**]
Current number of regions for this size class.

stats.arenas.<i>.bins.<j>.nfills (**uint64_t**) r-
Cumulative number of tcache fills.

stats.arenas.<i>.bins.<j>.nflushes (**uint64_t**) r-
Cumulative number of tcache flushes.

stats.arenas.<i>.bins.<j>.nslabs (**uint64_t**) r- [**--enable-stats**]
Cumulative number of slabs created.

stats.arenas.<i>.bins.<j>.nreslabs (**uint64_t**) r- [**--enable-stats**]
Cumulative number of times the current slab from which to allocate changed.

stats.arenas.<i>.bins.<j>.curslabs (**size_t**) r- [**--enable-stats**]
Current number of slabs.

stats.arenas.<i>.bins.<j>.nonfull_slabs (**size_t**) r- [**--enable-stats**]
Current number of nonfull slabs.

stats.arenas.<i>.bins.<j>.mutex.{counter} (**counter specific type**) r- [**--enable-stats**]
Statistics on *arena.<i>.bins.<j>* mutex (arena bin scope; bin operation related). {counter} is one of the counters in mutex profiling counters.

stats.arenas.<i>.extents.<j>.n{extent_type} (**size_t**) r- [**--enable-stats**]
Number of extents of the given type in this arena in the bucket corresponding to page size index <j>. The extent type is one of dirty, muzzy, or retained.

stats.arenas.<i>.extents.<j>.{extent_type}_bytes (**size_t**) r- [**--enable-stats**]
Sum of the bytes managed by extents of the given type in this arena in the bucket corresponding to page size index <j>. The extent type is one of dirty, muzzy, or retained.

stats.arenas.<i>.lextents.<j>.nmalloc (**uint64_t**) r- [**--enable-stats**]
Cumulative number of times a large extent of the corresponding size class was allocated from the arena, whether to fill the relevant tcache if opt.tcache is enabled and the size class is within the range being cached, or to directly satisfy an allocation request otherwise.

stats.arenas.<i>.lextents.<j>.ndalloc (**uint64_t**) r- [**--enable-stats**]
Cumulative number of times a large extent of the corresponding size class was returned to the arena, whether to flush the relevant tcache if opt.tcache is enabled and the size class is within the range being cached, or to directly deallocate an allocation otherwise.

stats.arenas.<i>.lextents.<j>.nrequests (**uint64_t**) r- [**--enable-stats**]
Cumulative number of allocation requests satisfied by large extents of the corresponding size class.

stats.arenas.<i>.lextents.<j>.curlextents (**size_t**) r- [**--enable-stats**]
Current number of large allocations for this size class.

stats.arenas.<i>.mutexes.large.{counter} (**counter specific type**) r- [**--enable-stats**]
Statistics on *arena.<i>.large* mutex (arena scope; large allocation related). {counter} is one of the counters in mutex profiling counters.

stats.arenas.<i>.mutexes.extent_avail.{counter} (**counter specific type**) r- [**--enable-stats**]
Statistics on *arena.<i>.extent_avail* mutex (arena scope; extent avail related). {counter} is one of the counters in mutex profiling counters.

stats.arenas.<i>.mutexes.extents_dirty.{counter} (**counter specific type**) r- [**--enable-stats**]
Statistics on *arena.<i>.extents_dirty* mutex (arena scope; dirty extents related). {counter} is one of the counters in mutex profiling counters.

stats.arenas.<i>.mutexes.extents_muzzy.{counter} (**counter specific type**) r- [**--enable-stats**]
Statistics on *arena.<i>.extents_muzzy* mutex (arena scope; muzzy extents related). {counter} is one of the counters in mutex profiling counters.

stats.arenas.<i>.mutexes.extents_retained.{counter} (**counter specific type**) r- [**--enable-stats**]
Statistics on *arena.<i>.extents_retained* mutex (arena scope; retained extents related). {counter} is one of the counters in mutex profiling counters.

stats.arenas.<i>.mutexes.decay_dirty.{counter} (**counter specific type**) r- [**--enable-stats**]
Statistics on *arena.<i>.decay_dirty* mutex (arena scope; decay for dirty pages related). {counter} is one of the counters in mutex profiling counters.

stats.arenas.<i>.mutexes.decay_muzzy.{counter} (**counter specific type**) r- [**--enable-stats**]
Statistics on *arena.<i>.decay_muzzy* mutex (arena scope; decay for muzzy pages related). {counter} is one of the counters in mutex profiling counters.

stats.arenas.<i>.mutexes.base.{counter} (**counter specific type**) r- [**--enable-stats**]
Statistics on *arena.<i>.base* mutex (arena scope; base allocator related). {counter} is one of the counters in mutex profiling counters.

stats.arenas.<i>.mutexes.tcache_list.{counter} (**counter specific type**) r- [**--enable-stats**]
Statistics on *arena.<i>.tcache_list* mutex (arena scope; tcache to arena association related). This mutex is expected to be accessed less often. {counter} is one of the counters in mutex profiling counters.

HEAP PROFILE FORMAT

Although the heap profiling functionality was originally designed to be compatible with the **pprof** command that is developed as part of the **gperftools package**[3], the addition of per thread heap

profiling functionality required a different heap profile format. The **jeprof** command is derived from **pprof**, with enhancements to support the heap profile format described here.

In the following hypothetical heap profile, **[...]** indicates elision for the sake of compactness.

```
heap_v2/524288
  t*: 28106: 56637512 [0: 0]
  [...]
  t3: 352: 16777344 [0: 0]
  [...]
  t99: 17754: 29341640 [0: 0]
  [...]
@ 0x5f86da8 0x5f5a1dc [...] 0x29e4d4e 0xa200316 0xabb2988 [...]
  t*: 13: 6688 [0: 0]
  t3: 12: 6496 [0: ]
  t99: 1: 192 [0: 0]
[...]

MAPPED_LIBRARIES:
[...]
```

The following matches the above heap profile, but most tokens are replaced with **<description>** to indicate descriptions of the corresponding fields.

```
<heap_profile_format_version>/<mean_sample_interval>
  <aggregate>: <curobjs>: <curbytes> [<cumobjs>: <cumbytes>]
  [...]
  <thread_3_aggregate>: <curobjs>: <curbytes>[<cumobjs>: <cumbytes>]
  [...]
  <thread_99_aggregate>: <curobjs>: <curbytes>[<cumobjs>: <cumbytes>]
  [...]
@ <top_frame> <frame> [...] <frame> <frame> <frame> [...]
  <backtrace_aggregate>: <curobjs>: <curbytes> [<cumobjs>: <cumbytes>]
  <backtrace_thread_3>: <curobjs>: <curbytes> [<cumobjs>: <cumbytes>]
  <backtrace_thread_99>: <curobjs>: <curbytes> [<cumobjs>: <cumbytes>]
[...]

MAPPED_LIBRARIES:
</proc/<pid>/maps>
```

DEBUGGING MALLOC PROBLEMS

When debugging, it is a good idea to configure/build jemalloc with the **--enable-debug** and **--enable-fill** options, and recompile the program with suitable options and symbols for debugger support. When so configured, jemalloc incorporates a wide variety of run-time assertions that catch application errors such as double-free, write-after-free, etc.

Programs often accidentally depend on "uninitialized" memory actually being filled with zero bytes. Junk filling (see the opt.junk option) tends to expose such bugs in the form of obviously incorrect results and/or coredumps. Conversely, zero filling (see the opt.zero option) eliminates the symptoms of such bugs. Between these two options, it is usually possible to quickly detect, diagnose, and eliminate such bugs.

This implementation does not provide much detail about the problems it detects, because the performance impact for storing such information would be prohibitive.

DIAGNOSTIC MESSAGES

If any of the memory allocation/deallocation functions detect an error or warning condition, a message will be printed to file descriptor **STDERR_FILENO**. Errors will result in the process dumping core. If the opt.abort option is set, most warnings are treated as errors.

The *malloc_message* variable allows the programmer to override the function which emits the text strings forming the errors and warnings if for some reason the **STDERR_FILENO** file descriptor is not suitable for this. malloc_message() takes the *cbopaque* pointer argument that is **NULL** unless overridden by the arguments in a call to malloc_stats_print(), followed by a string pointer. Please note that doing anything which tries to allocate memory in this function is likely to result in a crash or deadlock.

All messages are prefixed by "<jemalloc>: ".

RETURN VALUES

Standard API

The malloc() and calloc() functions return a pointer to the allocated memory if successful; otherwise a **NULL** pointer is returned and *errno* is set to ENOMEM.

The posix_memalign() function returns the value 0 if successful; otherwise it returns an error value. The posix_memalign() function will fail if:

EINVAL

The *alignment* parameter is not a power of 2 at least as large as sizeof(**void ***).

ENOMEM
Memory allocation error.

The aligned_alloc() function returns a pointer to the allocated memory if successful; otherwise a **NULL** pointer is returned and *errno* is set. The aligned_alloc() function will fail if:

EINVAL
The *alignment* parameter is not a power of 2.

ENOMEM
Memory allocation error.

The realloc() function returns a pointer, possibly identical to *ptr*, to the allocated memory if successful; otherwise a **NULL** pointer is returned, and *errno* is set to ENOMEM if the error was the result of an allocation failure. The realloc() function always leaves the original buffer intact when an error occurs.

The free() function returns no value.

Non-standard API

The mallocx() and rallocx() functions return a pointer to the allocated memory if successful; otherwise a **NULL** pointer is returned to indicate insufficient contiguous memory was available to service the allocation request.

The xallocx() function returns the real size of the resulting resized allocation pointed to by *ptr*, which is a value less than *size* if the allocation could not be adequately grown in place.

The sallocx() function returns the real size of the allocation pointed to by *ptr*.

The nallocx() returns the real size that would result from a successful equivalent mallocx() function call, or zero if insufficient memory is available to perform the size computation.

The mallctl(), mallctlnametomib(), and mallctlbymib() functions return 0 on success; otherwise they return an error value. The functions will fail if:

EINVAL
newp is not **NULL**, and *newlen* is too large or too small. Alternatively, **oldlenp* is too large or too small; in this case as much data as possible are read despite the error.

ENOENT
name or *mib* specifies an unknown/invalid value.

EPERM
: Attempt to read or write void value, or attempt to write read-only value.

EAGAIN
: A memory allocation failure occurred.

EFAULT
: An interface with side effects failed in some way not directly related to mallctl*() read/write processing.

The malloc_usable_size() function returns the usable size of the allocation pointed to by *ptr*.

ENVIRONMENT

The following environment variable affects the execution of the allocation functions:

MALLOC_CONF
: If the environment variable **MALLOC_CONF** is set, the characters it contains will be interpreted as options.

EXAMPLES

To dump core whenever a problem occurs:

```
ln -s 'abort:true' /etc/malloc.conf
```

To specify in the source that only one arena should be automatically created:

```
malloc_conf = "narenas:1";
```

SEE ALSO

madvise(2), **mmap**(2), **sbrk**(2), **utrace**(2), **alloca**(3), **atexit**(3), **getpagesize**(3)

STANDARDS

The malloc(), calloc(), realloc(), and free() functions conform to ISO/IEC 9899:1990 ("ISO C90").

The posix_memalign() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The malloc_usable_size() and posix_memalign() functions first appeared in FreeBSD 7.0.

The aligned_alloc(), malloc_stats_print(), and mallctl*() functions first appeared in FreeBSD 10.0.

The *allocx() functions first appeared in FreeBSD 11.0.

AUTHOR

Jason Evans

NOTES

1. jemalloc website
   http://jemalloc.net/

2. JSON format
   http://www.json.org/

3. gperftools package
   http://code.google.com/p/gperftools/