NAME

malloc, free, calloc, realloc – allocate and free dynamic memory

SYNOPSIS

#include <stdlib.h>

void *malloc(size_t *size***);**
void free(void **ptr***);**
void *calloc(size_t *nmemb***, size_t** *size***);**
void *realloc(void **ptr***, size_t** *size***);**

DESCRIPTION

The **malloc**() function allocates *size* bytes and returns a pointer to the allocated memory. *The memory is not initialized*. If *size* is 0, then **malloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

The **free**() function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**(), **calloc**(), or **realloc**(). Otherwise, or if *free(ptr)* has already been called before, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

The **calloc**() function allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero. If *nmemb* or *size* is 0, then **calloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

The **realloc**() function changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will *not* be initialized. If *ptr* is NULL, then the call is equivalent to *malloc(size)*, for all values of *size*; if *size* is equal to zero, and *ptr* is not NULL, then the call is equivalent to *free(ptr)*. Unless *ptr* is NULL, it must have been returned by an earlier call to **malloc**(), **calloc**() or **realloc**(). If the area pointed to was moved, a *free(ptr)* is done.

RETURN VALUE

The **malloc**() and **calloc**() functions return a pointer to the allocated memory, which is suitably aligned for any built-in type. On error, these functions return NULL. NULL may also be returned by a successful call to **malloc**() with a *size* of zero, or by a successful call to **calloc**() with *nmemb* or *size* equal to zero.

The **free**() function returns no value.

The **realloc**() function returns a pointer to the newly allocated memory, which is suitably aligned for any built-in type and may be different from *ptr*, or NULL if the request fails. If *size* was equal to 0, either NULL or a pointer suitable to be passed to **free**() is returned. If **realloc**() fails, the original block is left untouched; it is not freed or moved.

CONFORMING TO

C89, C99.

NOTES

By default, Linux follows an optimistic memory allocation strategy. This means that when **malloc**() returns non-NULL there is no guarantee that the memory really is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer. For more information, see the description of */proc/sys/vm/overcommit_memory* and */proc/sys/vm/oom_adj* in **proc**(5), and the Linux kernel source file *Documentation/vm/overcommit-accounting*.

Normally, **malloc**() allocates memory from the heap, and adjusts the size of the heap as required, using **sbrk**(2). When allocating blocks of memory larger than **MMAP_THRESHOLD** bytes, the glibc **malloc**() implementation allocates the memory as a private anonymous mapping using **mmap**(2). **MMAP_THRESHOLD** is 128 kB by default, but is adjustable using **mallopt**(3). Allocations performed using **mmap**(2) are unaffected by the **RLIMIT_DATA** resource limit (see **getrlimit**(2)).

To avoid corruption in multithreaded applications, mutexes are used internally to protect the memory-

management data structures employed by these functions. In a multithreaded application in which threads simultaneously allocate and free memory, there could be contention for these mutexes. To scalably handle memory allocation in multithreaded applications, glibc creates additional *memory allocation arenas* if mutex contention is detected. Each arena is a large region of memory that is internally allocated by the system (using **brk**(2) or **mmap**(2)), and managed with its own mutexes.

The UNIX 98 standard requires **malloc**(), **calloc**(), and **realloc**() to set *errno* to **ENOMEM** upon failure. Glibc assumes that this is done (and the glibc versions of these routines do this); if you use a private malloc implementation that does not set *errno*, then certain library routines may fail without having a reason in *errno*.

Crashes in **malloc**(), **calloc**(), **realloc**(), or **free**() are almost always related to heap corruption, such as overflowing an allocated chunk or freeing the same pointer twice.

The **malloc**() implementation is tunable via environment variables; see **mallopt**(3) for details.

SEE ALSO

brk(2), **mmap**(2), **alloca**(3), **malloc_get_state**(3), **malloc_info**(3), **malloc_trim**(3), **malloc_usable_size**(3), **mallopt**(3), **mcheck**(3), **mtrace**(3), **posix_memalign**(3)

COLOPHON

This page is part of release 3.74 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at http://www.kernel.org/doc/man−pages/.