NAME

UMA - general-purpose kernel object allocator

SYNOPSIS

```
#include <sys/param.h>
#include <sys/queue.h>
#include <vm/uma.h>

typedef int (*uma_ctor)(void *mem, int size, void *arg, int flags);
typedef void (*uma_dtor)(void *mem, int size, void *arg);
typedef int (*uma_init)(void *mem, int size, int flags);
typedef void (*uma_fini)(void *mem, int size);
typedef int (*uma_import)(void *arg, void **store, int count, int domain,
    int flags);
typedef void (*uma_release)(void *arg, void **store, int count);
typedef void *(*uma_alloc)(uma_zone_t zone, vm_size_t size, int domain,
    uint8_t *pflag, int wait);
typedef void (*uma_free)(void *item, vm_size_t size, uint8_t pflag);

uma_zone_t
uma_zcreate(char *name, size_t size, uma_ctor ctor, uma_dtor dtor, uma_init zinit, uma_fini zfini,
    int align, uint16_t flags);

uma_zone_t
uma_zcache_create(char *name, int size, uma_ctor ctor, uma_dtor dtor, uma_init zinit, uma_fini zfini,
    uma_import zimport, uma_release zrelease, void *arg, int flags);

uma_zone_t
uma_zsecond_create(char *name, uma_ctor ctor, uma_dtor dtor, uma_init zinit, uma_fini zfini,
    uma_zone_t master);

void
uma_zdestroy(uma_zone_t zone);

void *
uma_zalloc(uma_zone_t zone, int flags);

void *
uma_zalloc_arg(uma_zone_t zone, void *arg, int flags);
```

*void **
uma_zalloc_domain(*uma_zone_t zone*, *void *arg*, *int domain*, *int flags*);

*void **
uma_zalloc_pcpu(*uma_zone_t zone*, *int flags*);

*void **
uma_zalloc_pcpu_arg(*uma_zone_t zone*, *void *arg*, *int flags*);

*void **
uma_zalloc_smr(*uma_zone_t zone*, *int flags*);

void
uma_zfree(*uma_zone_t zone*, *void *item*);

void
uma_zfree_arg(*uma_zone_t zone*, *void *item*, *void *arg*);

void
uma_zfree_pcpu(*uma_zone_t zone*, *void *item*);

void
uma_zfree_pcpu_arg(*uma_zone_t zone*, *void *item*, *void *arg*);

void
uma_zfree_smr(*uma_zone_t zone*, *void *item*);

void
uma_prealloc(*uma_zone_t zone*, *int nitems*);

void
uma_zone_reserve(*uma_zone_t zone*, *int nitems*);

void
uma_zone_reserve_kva(*uma_zone_t zone*, *int nitems*);

void
uma_reclaim(*int req*);

void

uma_reclaim_domain(*int req*, *int domain*);

void
uma_zone_reclaim(*uma_zone_t zone*, *int req*);

void
uma_zone_reclaim_domain(*uma_zone_t zone*, *int req*, *int domain*);

void
uma_zone_set_allocf(*uma_zone_t zone*, *uma_alloc allocf*);

void
uma_zone_set_freef(*uma_zone_t zone*, *uma_free freef*);

int
uma_zone_set_max(*uma_zone_t zone*, *int nitems*);

void
uma_zone_set_maxcache(*uma_zone_t zone*, *int nitems*);

int
uma_zone_get_max(*uma_zone_t zone*);

int
uma_zone_get_cur(*uma_zone_t zone*);

void
uma_zone_set_warning(*uma_zone_t zone*, *const char *warning*);

void
uma_zone_set_maxaction(*uma_zone_t zone*, *void (*maxaction)(uma_zone_t)*);

smr_t
uma_zone_get_smr(*uma_zone_t zone*);

void
uma_zone_set_smr(*uma_zone_t zone*, *smr_t smr*);

#include <sys/sysctl.h>

SYSCTL_UMA_MAX(*parent*, *nbr*, *name*, *access*, *zone*, *descr*);

SYSCTL_ADD_UMA_MAX(*ctx*, *parent*, *nbr*, *name*, *access*, *zone*, *descr*);

SYSCTL_UMA_CUR(*parent*, *nbr*, *name*, *access*, *zone*, *descr*);

SYSCTL_ADD_UMA_CUR(*ctx*, *parent*, *nbr*, *name*, *access*, *zone*, *descr*);

DESCRIPTION

UMA (Universal Memory Allocator) provides an efficient interface for managing dynamically-sized collections of items of identical size, referred to as zones. Zones keep track of which items are in use and which are not, and UMA provides functions for allocating items from a zone and for releasing them back, making them available for subsequent allocation requests. Zones maintain per-CPU caches with linear scalability on SMP systems as well as round-robin and first-touch policies for NUMA systems. The number of items cached per CPU is bounded, and each zone additionally maintains an unbounded cache of items that is used to quickly satisfy per-CPU cache allocation misses.

Two types of zones exist: regular zones and cache zones. In a regular zone, items are allocated from a slab, which is one or more virtually contiguous memory pages that have been allocated from the kernel's page allocator. Internally, slabs are managed by a UMA keg, which is responsible for allocating slabs and keeping track of their usage by one or more zones. In typical usage, there is one keg per zone, so slabs are not shared among multiple zones.

Normal zones import items from a keg, and release items back to that keg if requested. Cache zones do not have a keg, and instead use custom import and release methods. For example, some collections of kernel objects are statically allocated at boot-time, and the size of the collection does not change. A cache zone can be used to implement an efficient allocator for the objects in such a collection.

The **uma_zcreate**() and **uma_zcache_create**() functions create a new regular zone and cache zone, respectively. The **uma_zsecond_create**() function creates a regular zone which shares the keg of the zone specified by the *master* argument. The *name* argument is a text name of the zone for debugging and stats; this memory should not be freed until the zone has been deallocated.

The *ctor* and *dtor* arguments are callback functions that are called by the UMA subsystem at the time of the call to **uma_zalloc**() and **uma_zfree**() respectively. Their purpose is to provide hooks for initializing or destroying things that need to be done at the time of the allocation or release of a resource. A good usage for the *ctor* and *dtor* callbacks might be to initialize a data structure embedded in the item, such as a queue(3) head.

The *zinit* and *zfini* arguments are used to optimize the allocation of items from the zone. They are called

by the UMA subsystem whenever it needs to allocate or free items to satisfy requests or memory pressure. A good use for the *zinit* and *zfini* callbacks might be to initialize and destroy a mutex contained within an item. This would allow one to avoid destroying and re-initializing the mutex each time the item is freed and re-allocated. They are not called on each call to **uma_zalloc**() and **uma_zfree**() but rather when an item is imported into a zone's cache, and when a zone releases an item to the slab allocator, typically as a response to memory pressure.

For **uma_zcache_create**(), the *zimport* and *zrelease* functions are called to import items into the zone and to release items from the zone, respectively. The *zimport* function should store pointers to items in the *store* array, which contains a maximum of *count* entries. The function must return the number of imported items, which may be less than the maximum. Similarly, the *store* parameter to the *zrelease* function contains an array of *count* pointers to items. The *arg* parameter passed to **uma_zcache_create**() is provided to the import and release functions. The *domain* parameter to *zimport* specifies the requested numa(4) domain for the allocation. It is either a NUMA domain number or the special value UMA_ANYDOMAIN.

The *flags* argument of **uma_zcreate**() and **uma_zcache_create**() is a subset of the following flags:

UMA_ZONE_NOFREE
: Slabs allocated to the zone's keg are never freed.

UMA_ZONE_NODUMP
: Pages belonging to the zone will not be included in minidumps.

UMA_ZONE_PCPU
: An allocation from zone would have *mp_ncpu* shadow copies, that are privately assigned to CPUs. A CPU can address its private copy using base the allocation address plus a multiple of the current CPU ID and **sizeof**(*struct pcpu*):

```
foo_zone = uma_zcreate(..., UMA_ZONE_PCPU);
 ...
foo_base = uma_zalloc(foo_zone, ...);
 ...
critical_enter();
foo_pcpu = (foo_t *)zpcpu_get(foo_base);
/* do something with foo_pcpu */
critical_exit();
```

Note that M_ZERO cannot be used when allocating items from a PCPU zone. To obtain zeroed memory from a PCPU zone, use the **uma_zalloc_pcpu**() function and its variants instead, and pass

M_ZERO.

UMA_ZONE_NOTOUCH
: The UMA subsystem may not directly touch (i.e. read or write) the slab memory. Otherwise, by default, book-keeping of items within a slab may be done in the slab page itself, and INVARIANTS kernels may also do use-after-free checking by accessing the slab memory.

UMA_ZONE_ZINIT
: The zone will have its *uma_init* method set to internal method that initializes a new allocated slab to all zeros. Do not mistake *uma_init* method with *uma_ctor*. A zone with UMA_ZONE_ZINIT flag would not return zeroed memory on every **uma_zalloc**().

UMA_ZONE_NOTPAGE
: An allocator function will be supplied with **uma_zone_set_allocf**() and the memory that it returns may not be kernel virtual memory backed by VM pages in the page array.

UMA_ZONE_MALLOC
: The zone is for the malloc(9) subsystem.

UMA_ZONE_VM
: The zone is for the VM subsystem.

UMA_ZONE_CONTIG
: Items in this zone must be contiguous in physical address space. Items will follow normal alignment constraints and may span page boundaries between pages with contiguous physical addresses.

UMA_ZONE_UNMANAGED
: By default, UMA zone caches are shrunk to help resolve free page shortages. Cached items that have not been used for a long period may also be freed from zone. When this flag is set, the system will not reclaim memory from the zone's caches.

UMA_ZONE_SMR
: Create a zone whose items will be synchronized using the smr(9) mechanism. Upon creation the zone will have an associated structure which can be fetched using **uma_zone_get_smr**().

Zones can be destroyed using **uma_zdestroy**(), freeing all memory that is cached in the zone. All items allocated from the zone must be freed to the zone before the zone may be safely destroyed.

To allocate an item from a zone, simply call **uma_zalloc**() with a pointer to that zone and set the *flags*

argument to selected flags as documented in malloc(9). It will return a pointer to an item if successful, or NULL in the rare case where all items in the zone are in use and the allocator is unable to grow the zone and M_NOWAIT is specified.

Items are released back to the zone from which they were allocated by calling **uma_zfree**() with a pointer to the zone and a pointer to the item. If *item* is NULL, then **uma_zfree**() does nothing.

The variants **uma_zalloc_arg**() and **uma_zfree_arg**() allow callers to specify an argument for the ctor and dtor functions of the zone, respectively. The variants **uma_zalloc_pcpu**() and **uma_zfree_pcpu**() allocate and free *mp_ncpu* shadow copies as described for UMA_ZONE_PCPU. If *item* is NULL, then **uma_zfree_pcpu**() does nothing.

The **uma_zalloc_smr**() and **uma_zfree_smr**() functions allocate and free items from an SMR-enabled zone, that is, a zone created with UMA_ZONE_SMR or a zone that has had **uma_zone_set_smr**() called.

The **uma_zalloc_domain**() function allows callers to specify a fixed numa(4) domain to allocate from. This uses a guaranteed but slow path in the allocator which reduces concurrency.

The **uma_prealloc**() function allocates slabs for the requested number of items, typically following the initial creation of a zone. Subsequent allocations from the zone will be satisfied using the pre-allocated slabs. Note that slab allocation is performed with the M_WAITOK flag, so **uma_prealloc**() may sleep.

The **uma_zone_reserve**() function sets the number of reserved items for the zone. **uma_zalloc**() and variants will ensure that the zone contains at least the reserved number of free items. Reserved items may be allocated by specifying M_USE_RESERVE in the allocation request flags. **uma_zone_reserve**() does not perform any pre-allocation by itself.

The **uma_zone_reserve_kva**() function pre-allocates kernel virtual address space for the requested number of items. Subsequent allocations from the zone will be satisfied using the pre-allocated address space. Note that unlike **uma_zone_reserve**(), **uma_zone_reserve_kva**() does not restrict the use of the pre-allocation to M_USE_RESERVE requests.

The **uma_reclaim**() and **uma_zone_reclaim**() functions reclaim cached items from UMA zones, releasing unused memory. The **uma_reclaim**() function reclaims items from all regular zones, while **uma_zone_reclaim**() reclaims items only from the specified zone. The *req* parameter must be one of three values which specify how aggressively items are to be reclaimed:

UMA_RECLAIM_TRIM
: Reclaim items only in excess of the zone's estimated working set size. The working set size is periodically updated and tracks the recent history of the zone's usage.

UMA_RECLAIM_DRAIN
Reclaim all items from the unbounded cache. Free items in the per-CPU caches are left alone.

UMA_RECLAIM_DRAIN_CPU
Reclaim all cached items.

The **uma_reclaim_domain**() and **uma_zone_reclaim_domain**() functions apply only to items allocated from the specified domain. In the case of domains using a round-robin NUMA policy, cached items from all domains are freed to the keg, but only slabs from the specific domain will be freed.

The **uma_zone_set_allocf**() and **uma_zone_set_freef**() functions allow a zone's default slab allocation and free functions to be overridden. This is useful if memory with special constraints such as attributes, alignment, or address ranges must be used.

The **uma_zone_set_max**() function limits the number of items (and therefore memory) that can be allocated to *zone*. The *nitems* argument specifies the requested upper limit number of items. The effective limit is returned to the caller, as it may end up being higher than requested due to the implementation rounding up to ensure all memory pages allocated to the zone are utilised to capacity. The limit applies to the total number of items in the zone, which includes allocated items, free items and free items in the per-cpu caches. On systems with more than one CPU it may not be possible to allocate the specified number of items even when there is no shortage of memory, because all of the remaining free items may be in the caches of the other CPUs when the limit is hit.

The **uma_zone_set_maxcache**() function limits the number of free items which may be cached in the zone. This limit applies to both the per-CPU caches and the cache of free buckets.

The **uma_zone_get_max**() function returns the effective upper limit number of items for a zone.

The **uma_zone_get_cur**() function returns an approximation of the number of items currently allocated from the zone. The returned value is approximate because appropriate synchronisation to determine an exact value is not performed by the implementation. This ensures low overhead at the expense of potentially stale data being used in the calculation.

The **uma_zone_set_warning**() function sets a warning that will be printed on the system console when the given zone becomes full and fails to allocate an item. The warning will be printed no more often than every five minutes. Warnings can be turned off globally by setting the *vm.zone_warnings* sysctl tunable to *0*.

The **uma_zone_set_maxaction**() function sets a function that will be called when the given zone becomes full and fails to allocate an item. The function will be called with the zone locked. Also, the function that called the allocation function may have held additional locks. Therefore, this function

should do very little work (similar to a signal handler).

The **uma_zone_set_smr**() function associates an existing smr(9) structure with a UMA zone. The effect is similar to creating a zone with the UMA_ZONE_SMR flag, except that a new SMR structure is not created. This function must be called before any allocations from the zone are performed.

The **SYSCTL_UMA_MAX**(*parent*, *nbr*, *name*, *access*, *zone*, *descr*) macro declares a static sysctl(9) oid that exports the effective upper limit number of items for a zone. The *zone* argument should be a pointer to *uma_zone_t*. A read of the oid returns value obtained through **uma_zone_get_max**(). A write to the oid sets new value via **uma_zone_set_max**(). The **SYSCTL_ADD_UMA_MAX**(*ctx*, *parent*, *nbr*, *name*, *access*, *zone*, *descr*) macro is provided to create this type of oid dynamically.

The **SYSCTL_UMA_CUR**(*parent*, *nbr*, *name*, *access*, *zone*, *descr*) macro declares a static read-only sysctl(9) oid that exports the approximate current occupancy of the zone. The *zone* argument should be a pointer to *uma_zone_t*. A read of the oid returns value obtained through **uma_zone_get_cur**(). The **SYSCTL_ADD_UMA_CUR**(*ctx*, *parent*, *nbr*, *name*, *zone*, *descr*) macro is provided to create this type of oid dynamically.

IMPLEMENTATION NOTES

The memory that these allocation calls return is not executable. The **uma_zalloc**() function does not support the M_EXEC flag to allocate executable memory. Not all platforms enforce a distinction between executable and non-executable memory.

SEE ALSO

numa(4), vmstat(8), malloc(9), smr(9)

Jeff Bonwick, *The Slab Allocator: An Object-Caching Kernel Memory Allocator*, 1994.

HISTORY

The zone allocator first appeared in FreeBSD 3.0. It was radically changed in FreeBSD 5.0 to function as a slab allocator.

AUTHORS

The zone allocator was written by John S. Dyson. The zone allocator was rewritten in large parts by Jeff Roberson <*jeff@FreeBSD.org*> to function as a slab allocator.

This manual page was written by Dag-Erling Smørgrav <*des@FreeBSD.org*>. Changes for UMA by Jeroen Ruigrok van der Werven <*asmodai@FreeBSD.org*>.