

NAME

vm_page_alloc - allocate a page of memory

SYNOPSIS

```
#include <sys/param.h>
```

```
#include <vm/vm.h>
```

```
#include <vm/vm_page.h>
```

vm_page_t

```
vm_page_alloc(vm_object_t object, vm_pindex_t pindex, int req);
```

vm_page_t

```
vm_page_alloc_after(vm_object_t object, vm_pindex_t pindex, int req, vm_page_t mpred);
```

vm_page_t

```
vm_page_alloc_contig(vm_object_t object, vm_pindex_t pindex, int req, u_long npages,  
    vm_paddr_t low, vm_paddr_t high, u_long alignment, vm_paddr_t boundary,  
    vm_memattr_t memattr);
```

vm_page_t

```
vm_page_alloc_contig_domain(vm_object_t object, vm_pindex_t pindex, int req, u_long npages,  
    vm_paddr_t low, vm_paddr_t high, u_long alignment, vm_paddr_t boundary,  
    vm_memattr_t memattr);
```

vm_page_t

```
vm_page_alloc_domain(vm_object_t object, vm_pindex_t pindex, int domain, int req);
```

vm_page_t

```
vm_page_alloc_domain_after(vm_object_t object, vm_pindex_t pindex, int domain, int req,  
    vm_page_t mpred);
```

vm_page_t

```
vm_page_alloc_freelist(int freelist, int req);
```

vm_page_t

```
vm_page_alloc_freelist_domain(int domain, int freelist, int req);
```

vm_page_t

```
vm_page_alloc_noobj(int req);
```

vm_page_t

vm_page_alloc_noobj_contig(*int req, u_long npages, vm_paddr_t low, vm_paddr_t high, u_long alignment, vm_paddr_t boundary, vm_memattr_t memattr*);

vm_page_t

vm_page_alloc_noobj_contig_domain(*int domain, int req, u_long npages, vm_paddr_t low, vm_paddr_t high, u_long alignment, vm_paddr_t boundary, vm_memattr_t memattr*);

vm_page_t

vm_page_alloc_noobj_domain(*int domain, int req*);

DESCRIPTION

The **vm_page_alloc**() family of functions allocate one or more pages of physical memory. Most kernel code should not call these functions directly but should instead use a kernel memory allocator such as **malloc**(9) or **uma**(9), or should use a higher-level interface to the page cache, such as **vm_page_grab**(9).

All of the functions take a *req* parameter which encodes the allocation priority and optional modifier flags, described below. The functions whose names do not include "noobj" additionally insert the pages starting at index *pindex* in the VM object *object*. The object must be write-locked and not have a page already resident at the specified index. The functions whose names include "domain" support NUMA-aware allocation by returning pages from the **numa**(4) domain specified by *domain*.

The **vm_page_alloc_after**() and **vm_page_alloc_domain_after**() functions behave identically to **vm_page_alloc**() and **vm_page_alloc_domain**(), respectively, except that they take an additional parameter *mpred* which must be the page resident in *object* with largest index smaller than *pindex*, or NULL if no such page exists. These functions exist to optimize the common case of loops that allocate multiple pages at successive indices within an object.

The **vm_page_alloc_contig**() and **vm_page_alloc_noobj_contig**() functions and their NUMA-aware variants allocate a physically contiguous run of *npages* pages which satisfies the specified constraints. The *low* and *high* parameters specify a physical address range from which the run is to be allocated. The *alignment* parameter specifies the requested alignment of the first page in the run and must be a power of two. If the *boundary* parameter is non-zero, the pages constituting the run will not cross a physical address that is a multiple of the parameter value, which must be a power of two. If *memattr* is not equal to **VM_MEMATTR_DEFAULT**, then mappings of the returned pages created by, e.g., **pmap_enter**(9) or **pmap_qenter**(9), will carry the machine-dependent encoding of the memory attribute. Additionally, the direct mapping of the page, if any, will be updated to reflect the requested memory attribute.

The **vm_page_alloc_freelist**() and **vm_page_alloc_freelist_domain**() functions behave identically to

vm_page_alloc_noobj() and **vm_page_alloc_noobj_domain()**, respectively, except that a successful allocation will return a page from the specified physical memory freelist. These functions are not intended for use outside of the virtual memory subsystem and exist only to support the requirements of certain platforms.

REQUEST FLAGS

All page allocator functions accept a *req* parameter that governs certain aspects of the function's behavior.

The `VM_ALLOC_WAITOK`, `VM_ALLOC_WAITFAIL`, and `VM_ALLOC_NOWAIT` flags specify the behavior of the allocator if free pages could not be immediately allocated. The `VM_ALLOC_WAITOK` flag can only be used with the "noobj" variants. If `VM_ALLOC_NOWAIT` is specified, then the allocator gives up and returns `NULL`. `VM_ALLOC_NOWAIT` is specified implicitly if none of the flags are present in the request. If either `VM_ALLOC_WAITOK` or `VM_ALLOC_WAITFAIL` is specified, the allocator will put the calling thread to sleep until sufficient free pages become available. At this point, if `VM_ALLOC_WAITFAIL` is specified the allocator will return `NULL`, and if `VM_ALLOC_WAITOK` is specified the allocator will retry the allocation. After a failed `VM_ALLOC_WAITFAIL` allocation returns, the VM object, if any, will have been unlocked while the thread was sleeping. In this case the VM object write lock will be re-acquired before the function call returns.

req also encodes the allocation request priority. By default the page(s) are allocated with no special treatment. If the number of available free pages is below a certain watermark, the allocation will fail or the allocating thread will sleep, depending on the specified wait flag. The watermark is computed at boot time and corresponds to a small (less than one percent) fraction of the system's total physical memory. To allocate memory more aggressively, one of following flags may be specified.

`VM_ALLOC_SYSTEM` The page can be allocated if the free page count is above the interrupt reserved water mark. This flag should be used only when the system really needs the page.

`VM_ALLOC_INTERRUPT` The allocation will fail only if zero free pages are available. This flag should be used only if the consequences of an allocation failure are worse than leaving the system without free memory. For example, this flag is used when allocating kernel page table pages, where allocation failures trigger a kernel panic.

The following optional flags can further modify allocator behavior:

`VM_ALLOC_SBUSY` The returned page will be shared-busy. This flag may only be specified when

allocating pages in a VM object.

VM_ALLOC_NOBUSY The returned page will not be busy. This flag is implicit when allocating pages without a VM object. When allocating pages in a VM object, and neither **VM_ALLOC_SBUSY** nor **VM_ALLOC_NOBUSY** are specified, the returned pages will be exclusively busied.

VM_ALLOC_NODUMP

The returned page will not be included in any kernel core dumps regardless of whether or not it is mapped in to KVA.

VM_ALLOC_WIRED The returned page will be wired.

VM_ALLOC_ZERO If this flag is specified, the "noobj" variants will return zeroed pages. The other allocator interfaces ignore this flag.

VM_ALLOC_NORECLAIM

If this flag is specified and the request can not be immediately satisfied, the allocator will not attempt to break superpage reservations to satisfy the allocation. This may be useful when the overhead of scanning the reservation queue outweighs the cost of a failed allocation. This flag may be used only with the "contig" variants, and must not be specified in combination with **VM_ALLOC_WAITOK**.

VM_ALLOC_COUNT(n)

Hint that at least *n* pages will be allocated by the caller in the near future. *n* must be no larger than 65535. If the system is short of free pages, this hint may cause the kernel to reclaim memory more aggressively than it would otherwise.

RETURN VALUES

If the allocation was successful, a pointer to the *struct vm_page* corresponding to the allocated page is returned. If the allocation request specified multiple pages, the returned pointer points to an array of *struct vm_page* constituting the run. Upon failure, NULL is returned. Regardless of whether the allocation succeeds or fails, the VM object *object* will be write-locked upon return.

SEE ALSO

numa(4), malloc(9), uma(9), vm_page_grab(9), vm_page_sbusy(9)

AUTHORS

This manual page was written by Chad David <*davidc@acns.ab.ca*>.