Jinue

Generated by Doxygen 1.5.5

Sat Jul 18 14:18:05 2009

Contents

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 boot_t Struct Reference

```
#include <boot.h>
```

3.1.1 Detailed Description

Definition at line 26 of file boot.h.

Data Fields

- unsigned long **magic**
- unsigned char **setup_sects**
- unsigned short **root_flags**
- unsigned long **sysize**
- unsigned short **ram_size**
- unsigned short **vid_mode**
- unsigned short **root_dev**
- unsigned short **signature**

3.1.2 Field Documentation

3.1.2.1 unsigned long boot_t::magic

Definition at line 27 of file boot.h.

Referenced by get_boot_data().

3.1.2.2 unsigned char boot_t::setup_sects

Definition at line 28 of file boot.h.

3.1.2.3 unsigned short boot_t::root_flags

Definition at line 29 of file boot.h.

3.1.2.4 unsigned long boot_t::sysize

Definition at line 30 of file boot.h.

Referenced by kinit().

3.1.2.5 unsigned short boot_t::ram_size

Definition at line 31 of file boot.h.

3.1.2.6 unsigned short boot_t::vid_mode

Definition at line 32 of file boot.h.

3.1.2.7 unsigned short boot_t::root_dev

Definition at line 33 of file boot.h.

3.1.2.8 unsigned short boot_t::signature

Definition at line 34 of file boot.h.

Referenced by get_boot_data().

The documentation for this struct was generated from the following file:

- include/**boot.h**

3.2 e820_t Struct Reference

`#include <boot.h>`

3.2.1 Detailed Description

Definition at line 19 of file boot.h.

Data Fields

- **e820_addr_t addr**
- **e820_size_t size**
- **e820_type_t type**

3.2.2 Field Documentation

3.2.2.1 e820_addr_t e820_t::addr

Definition at line 20 of file boot.h.

3.2.2.2 e820_size_t e820_t::size

Definition at line 21 of file boot.h.

Referenced by e820_get_size().

3.2.2.3 e820_type_t e820_t::type

Definition at line 22 of file boot.h.

Referenced by e820_get_type().

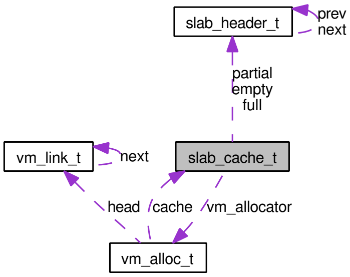
The documentation for this struct was generated from the following file:

- include/**boot.h**

3.3 slab_cache_t Struct Reference

`#include <slab.h>`

Collaboration diagram for slab_cache_t:



3.3.1 Detailed Description

data structure describing a cache

Definition at line 24 of file slab.h.

Data Fields

- **size_t obj_size**

  size of objects to allocate

- **count_t per_slab**

  number of objects per slab

- **slab_header_t * empty**

  head of list of empty slabs

- **slab_header_t * partial**

  head of list of partial slabs

- **slab_header_t * full**

  head of list of full slabs

- unsigned long **vm_flags**

  flags for mapping slabs in virtual memory

- struct **vm_alloc_t * vm_allocator**

virtual address space allocator for new slabs

3.3.2 Field Documentation

3.3.2.1 size_t slab_cache_t::obj_size

size of objects to allocate

Definition at line 26 of file slab.h.

Referenced by slab_prepare().

3.3.2.2 count_t slab_cache_t::per_slab

number of objects per slab

Definition at line 29 of file slab.h.

Referenced by slab_prepare().

3.3.2.3 slab_header_t∗ slab_cache_t::empty

head of list of empty slabs

Definition at line 32 of file slab.h.

Referenced by slab_alloc(), and vm_vfree_block().

3.3.2.4 slab_header_t∗ slab_cache_t::partial

head of list of partial slabs

Definition at line 35 of file slab.h.

Referenced by slab_alloc(), and vm_vfree_block().

3.3.2.5 slab_header_t∗ slab_cache_t::full

head of list of full slabs

Definition at line 38 of file slab.h.

Referenced by slab_alloc().

3.3.2.6 unsigned long slab_cache_t::vm_flags

flags for mapping slabs in virtual memory

Definition at line 41 of file slab.h.

Referenced by slab_alloc().

3.3.2.7 struct vm_alloc_t* slab_cache_t::vm_allocator [read]

virtual address space allocator for new slabs

Definition at line 44 of file slab.h.

Referenced by slab_alloc(), and vm_vfree_block().

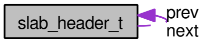
The documentation for this struct was generated from the following file:

- include/**slab.h**

3.4 slab_header_t Struct Reference

`#include <slab.h>`

Collaboration diagram for slab_header_t:



3.4.1 Detailed Description

header of a slab

Definition at line 7 of file slab.h.

Data Fields

- **count_t available**
  number of available objects in free list
- **addr_t free_list**
  head of the free list
- struct **slab_header_t** ∗ **next**
  pointer to next slab in linked list
- struct **slab_header_t** ∗ **prev**
  pointer to previous slab in linked list

3.4.2 Field Documentation

3.4.2.1 count_t slab_header_t::available

number of available objects in free list

Definition at line 9 of file slab.h.

Referenced by slab_alloc(), and slab_prepare().

3.4.2.2 addr_t slab_header_t::free_list

head of the free list

Definition at line 12 of file slab.h.

Referenced by slab_alloc(), and slab_prepare().

3.4.2.3 struct slab_header_t* slab_header_t::next `[read]`

pointer to next slab in linked list

Definition at line 15 of file slab.h.

Referenced by slab_add(), and slab_remove().

3.4.2.4 struct slab_header_t* slab_header_t::prev `[read]`

pointer to previous slab in linked list

Definition at line 18 of file slab.h.

Referenced by slab_add(), and slab_remove().

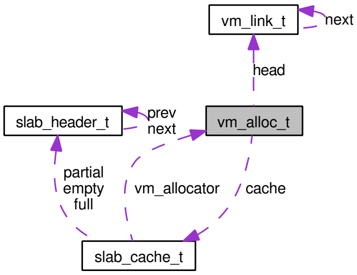
The documentation for this struct was generated from the following file:

- include/**slab.h**

3.5 vm_alloc_t Struct Reference

`#include <vm_alloc.h>`

Collaboration diagram for vm_alloc_t:



3.5.1 Detailed Description

data structure which keep tracks of free pages in a region of virtual memory

Definition at line 23 of file vm_alloc.h.

Data Fields

- **size_t size**

  total amount of memory available

- **vm_link_t** ∗ **head**

  head of the free list

- struct **slab_cache_t** ∗ **cache**

  slab cache on which to allocate the links of the free list

3.5.2 Field Documentation

3.5.2.1 size_t vm_alloc_t::size

total amount of memory available

Definition at line 25 of file vm_alloc.h.

Referenced by alloc_init(), e820_is_valid(), printk(), and vm_valloc().

3.5.2.2 vm_link_t* vm_alloc_t::head

head of the free list

Definition at line 28 of file vm_alloc.h.

Referenced by vm_valloc(), and vm_vfree_block().

3.5.2.3 struct slab_cache_t* vm_alloc_t::cache `[read]`

slab cache on which to allocate the links of the free list

Definition at line 31 of file vm_alloc.h.

Referenced by vm_valloc(), and vm_vfree_block().

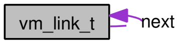
The documentation for this struct was generated from the following file:

- include/**vm_alloc.h**

3.6 vm_link_t Struct Reference

`#include <vm_alloc.h>`

Collaboration diagram for vm_link_t:



3.6.1 Detailed Description

links forming the linked lists of free virtual memory pages

Definition at line 8 of file vm_alloc.h.

Data Fields

- struct **vm_link_t** ∗ **next**

  next link in list

- **size_t size**

  size of current virtual memory block

- **addr_t addr**

  starting address of current block

3.6.2 Field Documentation

3.6.2.1 struct vm_link_t∗ vm_link_t::next `[read]`

next link in list

Definition at line 10 of file vm_alloc.h.

Referenced by vm_valloc(), and vm_vfree_block().

3.6.2.2 size_t vm_link_t::size

size of current virtual memory block

Definition at line 13 of file vm_alloc.h.

Referenced by vm_valloc(), and vm_vfree_block().

3.6.2.3 addr_t vm_link_t::addr

starting address of current block

Definition at line 16 of file vm_alloc.h.

Referenced by vm_valloc(), and vm_vfree_block().

The documentation for this struct was generated from the following file:

- include/**vm_alloc.h**

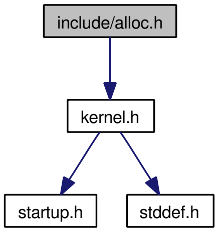
Chapter 4

File Documentation

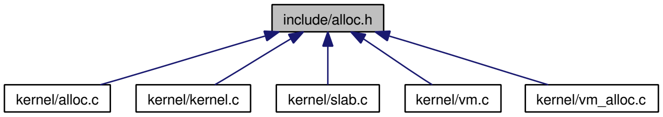
4.1 include/alloc.h File Reference

`#include <kernel.h>`

Include dependency graph for alloc.h:



This graph shows which files directly or indirectly include this file:



Functions

- void **alloc_init** (void)
- **addr_t alloc** (**size_t** size)
- void **free** (**addr_t** addr)

4.1.1 Function Documentation

4.1.1.1 addr_t alloc (size_t *size*)

ASSERTION: returned address should be aligned with a page boundary

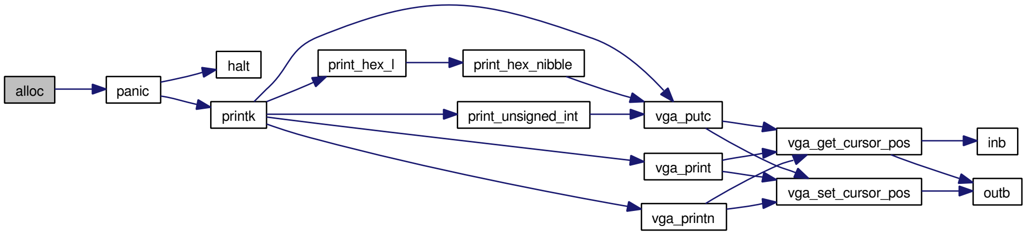
Definition at line 96 of file alloc.c.

References assert, PAGE_BITS, PAGE_MASK, PAGE_SIZE, and panic().

Referenced by vm_alloc(), vm_map(), and vm_vfree_block().

```
96                           {
97      addr_t addr;
98      size_t pages;
99
100      pages = size >> PAGE_BITS;
101
102      if( (size & PAGE_MASK) != 0 ) {
103          ++pages;
104      }
105
106      if(_alloc_size < pages) {
107          panic("out of memory.");
108      }
109
110      addr = _alloc_addr;
111      _alloc_addr += pages * PAGE_SIZE;
112      _alloc_size -= pages;
113
115      assert( ((unsigned long)addr & PAGE_MASK) == 0 );
116
117      return addr;
118 }
```

Here is the call graph for this function:



4.1.1.2 void alloc_init (void)

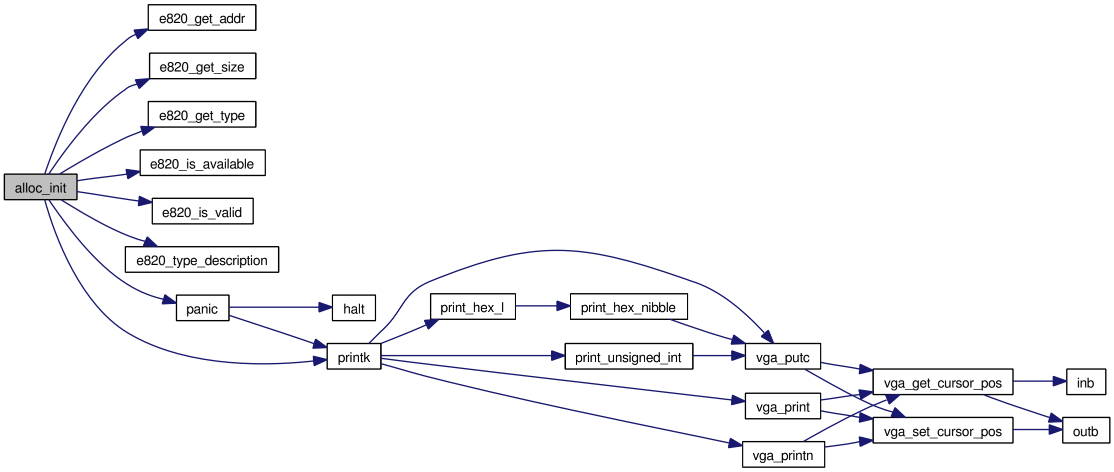
Definition at line 12 of file alloc.c.

References e820_get_addr(), e820_get_size(), e820_get_type(), e820_is_available(), e820_is_valid(), e820_type_description(), kernel_start, kernel_top, PAGE_SIZE, panic(), printk(), and vm_alloc_t::size.

Referenced by kinit().

```
                      {
    unsigned int idx;
    unsigned int remainder;
    bool avail;
    size_t size;
    e820_type_t type;
    addr_t addr, fixed_addr, best_addr;
    size_t fixed_size, best_size;

    idx = 0;
    best_size = 0;

    printk("Dump of the BIOS memory map:\n");
    printk("  address  size     type\n");
    while( e820_is_valid(idx) ) {
        addr = e820_get_addr(idx);
        size = e820_get_size(idx);
        type = e820_get_type(idx);
        avail = e820_is_available(idx);

        ++idx;

        printk("%c %x %x %s\n",
            avail?'*':' ',
            addr,
            size,
            e820_type_description(type) );

        if( !avail ) {
            continue;
        }

        fixed_addr = addr;
        fixed_size = size;

        /* is the region completely under the kernel ? */
        if(addr + size > kernel_start) {
            /* is the region completely above the kernel ? */
            if(addr < kernel_top) {
                /* if the region touches the kernel, we take only
                 * the part above the kernel, if there is one... */
                if(addr + size <= kernel_top) {
                    /* ... and apparently, there is none */
                    continue;
                }

                fixed_addr = kernel_top;
                fixed_size -= fixed_addr - addr;
            }
        }

        /* we must make sure the starting address is aligned on a
         * page boundary. The size will eventually be divided
         * by the page size, and thus need not be aligned. */
        remainder = (unsigned int)fixed_addr % PAGE_SIZE;
        if(remainder != 0) {
            remainder = PAGE_SIZE - remainder;
            if(fixed_size < remainder) {
                continue;
            }

            fixed_addr += remainder;
            fixed_size -= remainder;
        }

        if(fixed_size > best_size) {
            best_addr = fixed_addr;
            best_size = fixed_size;
        }
    }

    _alloc_addr = (addr_t)best_addr;
    _alloc_size = best_size / PAGE_SIZE;

    if(_alloc_size == 0) {
        panic("no memory to allocate.");
    }

    printk("%u kilobytes (%u pages) available starting at %xh.\n",
        _alloc_size * PAGE_SIZE / 1024,
        _alloc_size,
        _alloc_addr );
}
```

Here is the call graph for this function:



4.1.1.3 void free (addr_t *addr*)

ASSERTION: we assume starting address is aligned on a page boundary

Definition at line 120 of file alloc.c.

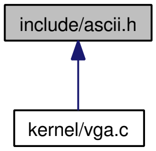
References assert, and PAGE_OFFSET_OF.

Referenced by vm_free().

```
120                        {
122     assert( PAGE_OFFSET_OF(addr) == 0 );
123 }
```

4.2 include/ascii.h File Reference

This graph shows which files directly or indirectly include this file:



Defines

- #define **CHAR_BS** 0x08
- #define **CHAR_HT** 0x09
- #define **CHAR_LF** 0x0a
- #define **CHAR_CR** 0x0d

4.2.1 Define Documentation

4.2.1.1 #define CHAR_BS 0x08

Definition at line 4 of file ascii.h.

4.2.1.2 #define CHAR_CR 0x0d

Definition at line 7 of file ascii.h.

4.2.1.3 #define CHAR_HT 0x09

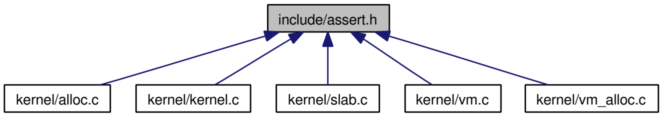
Definition at line 5 of file ascii.h.

4.2.1.4 #define CHAR_LF 0x0a

Definition at line 6 of file ascii.h.

4.3 include/assert.h File Reference

This graph shows which files directly or indirectly include this file:



Defines

- #define **assert**(expr)

Functions

- void **__assert_failed** (const char *expr, const char *file, unsigned int line, const char *func)

4.3.1 Define Documentation

4.3.1.1 #define assert(expr)

Value:

```
( \
        (expr)?(void)0:( __assert_failed(#expr, __FILE__, __LINE__, __func__) ) \
    )
```

Definition at line 12 of file assert.h.

Referenced by alloc(), free(), kinit(), slab_prepare(), vm_free(), vm_map(), vm_unmap(), vm_valloc(), and vm_vfree_block().

4.3.2 Function Documentation

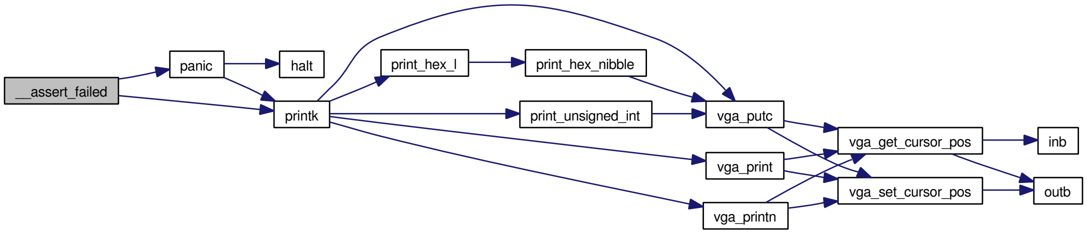
4.3.2.1 void __assert_failed (const char * *expr*, const char * *file*, unsigned int *line*, const char * *func*)

Definition at line 5 of file assert.c.

References panic(), and printk().

```
                      {

    printk(
        "ASSERTION FAILED [%s]: %s at line %u in function %s.\n",
        expr, file, line, func );

    panic("Assertion failed.");
}
```

Here is the call graph for this function:

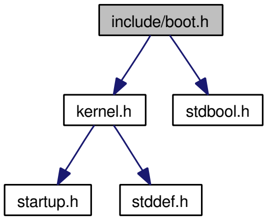


4.4 include/boot.h File Reference

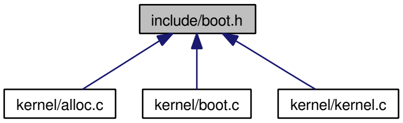
`#include <kernel.h>`

`#include <stdbool.h>`

Include dependency graph for boot.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct **e820_t**
- struct **boot_t**

Defines

- #define **BOOT_SIGNATURE** 0xaa55
- #define **BOOT_MAGIC** 0xcafef00d
- #define **SETUP_HEADER** 0x53726448
- #define **E820_RAM** 1
- #define **E820_RESERVED** 2
- #define **E820_ACPI** 3

Typedefs

- typedef unsigned long long **e820_addr_t**
- typedef unsigned long long **e820_size_t**
- typedef unsigned long **e820_type_t**

Functions

- **addr_t e820_get_addr** (unsigned int idx)
- **size_t e820_get_size** (unsigned int idx)
- **e820_type_t e820_get_type** (unsigned int idx)
- bool **e820_is_valid** (unsigned int idx)
- bool **e820_is_available** (unsigned int idx)
- const char ∗ **e820_type_description** (**e820_type_t** type)
- **boot_t** ∗ **get_boot_data** (void)

4.4.1 Define Documentation

4.4.1.1 #define BOOT_MAGIC 0xcafef00d

Definition at line 8 of file boot.h.

Referenced by get_boot_data().

4.4.1.2 #define BOOT_SIGNATURE 0xaa55

Definition at line 7 of file boot.h.

Referenced by get_boot_data().

4.4.1.3 #define E820_ACPI 3

Definition at line 13 of file boot.h.

Referenced by e820_type_description().

4.4.1.4 #define E820_RAM 1

Definition at line 11 of file boot.h.

Referenced by e820_is_available(), and e820_type_description().

4.4.1.5 #define E820_RESERVED 2

Definition at line 12 of file boot.h.

Referenced by e820_type_description().

4.4.1.6 #define SETUP_HEADER 0x53726448

Definition at line 9 of file boot.h.

4.4.2 Typedef Documentation

4.4.2.1 typedef unsigned long long e820_addr_t

Definition at line 15 of file boot.h.

4.4.2.2 typedef unsigned long long e820_size_t

Definition at line 16 of file boot.h.

4.4.2.3 typedef unsigned long e820_type_t

Definition at line 17 of file boot.h.

4.4.3 Function Documentation

4.4.3.1 addr_t e820_get_addr (unsigned int *idx*)

Definition at line 8 of file boot.c.

Referenced by alloc_init().

```
8                                          {
9     return (addr_t)(unsigned long)e820_map[idx].addr;
10 }
```

4.4.3.2 size_t e820_get_size (unsigned int *idx*)

Definition at line 12 of file boot.c.

References e820_t::size.

Referenced by alloc_init().

```
12                                          {
13     return (size_t)e820_map[idx].size;
14 }
```

4.4.3.3 e820_type_t e820_get_type (unsigned int *idx*)

Definition at line 16 of file boot.c.

References e820_t::type.

Referenced by alloc_init().

```
16                                              {
17     return e820_map[idx].type;
18 }
```

4.4.3.4 bool e820_is_available (unsigned int *idx*)

Definition at line 24 of file boot.c.

References E820_RAM.

Referenced by alloc_init().

```
24                                           {
25     return (e820_map[idx].type == E820_RAM);
26 }
```

4.4.3.5 bool e820_is_valid (unsigned int *idx*)

Definition at line 20 of file boot.c.

References vm_alloc_t::size.

Referenced by alloc_init().

```
20                                       {
21     return (e820_map[idx].size != 0);
22 }
```

4.4.3.6 const char* e820_type_description (e820_type_t *type*)

Definition at line 28 of file boot.c.

References E820_ACPI, E820_RAM, and E820_RESERVED.

Referenced by alloc_init().

```
28                                                      {
29     switch(type) {
30
31     case E820_RAM:
32         return "available";
33
34     case E820_RESERVED:
35         return "unavailable/reserved";
36
37     case E820_ACPI:
38         return "unavailable/acpi";
39
40     default:
41         return "unavailable/other";
42     }
43 }
```

4.4.3.7 boot_t∗ get_boot_data (void)

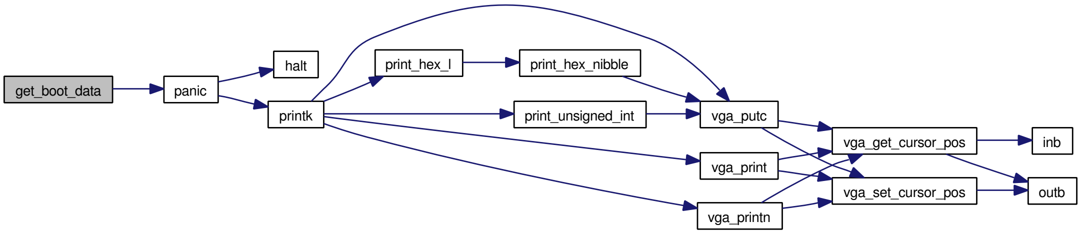
Definition at line 45 of file boot.c.

References BOOT_MAGIC, boot_setup_addr, BOOT_SIGNATURE, boot_t::magic, panic(), and boot_t::signature.

Referenced by kinit().

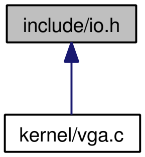
```
45                             {
46     boot_t *boot;
47 
48     boot = (boot_t *)( boot_setup_addr - sizeof(boot_t) );
49 
50     if(boot->signature != BOOT_SIGNATURE) {
51         panic("bad boot sector signature.");
52     }
53 
54     if(boot->magic != BOOT_MAGIC) {
55         panic("bad boot sector magic.");
56     }
57 
58     return boot;
59 }
```

Here is the call graph for this function:



4.5 include/io.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- unsigned char **inb** (unsigned short int port)
- unsigned short int **inw** (unsigned short int port)
- unsigned int **inl** (unsigned short int port)
- void **outb** (unsigned short int port, unsigned char value)
- void **outw** (unsigned short int port, unsigned short int value)
- void **outl** (unsigned short int port, unsigned int value)

4.5.1 Function Documentation

4.5.1.1 unsigned char inb (unsigned short int *port*)

Referenced by vga_get_cursor_pos(), and vga_init().

4.5.1.2 unsigned int inl (unsigned short int *port*)

4.5.1.3 unsigned short int inw (unsigned short int *port*)

4.5.1.4 void outb (unsigned short int *port*, unsigned char *value*)

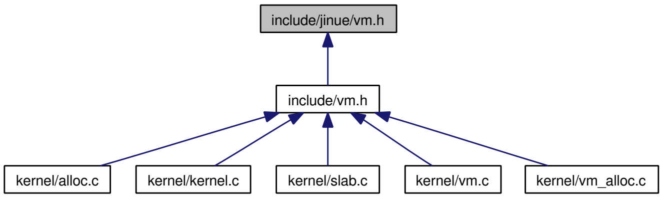
Referenced by vga_get_cursor_pos(), vga_init(), and vga_set_cursor_pos().

4.5.1.5 void outl (unsigned short int *port*, unsigned int *value*)

4.5.1.6 void outw (unsigned short int *port*, unsigned short int *value*)

4.6 include/jinue/vm.h File Reference

This graph shows which files directly or indirectly include this file:



Defines

- #define **PAGE_BITS** 12

  number of bits in virtual address for offset inside page

- #define **PAGE_SIZE** (1<<PAGE_BITS)

  size of page

- #define **PAGE_TABLE_BITS** 10

  number of bits in virtual address for page table entry

- #define **PAGE_TABLE_ENTRIES** (1<<PAGE_TABLE_BITS)

  number of entries in page table

- #define **PAGE_TABLE_SIZE** PAGE_SIZE

  size of a page table

- #define **PTE_SIZE** 4

  size of a page table entry, in bytes

- #define **KLIMIT** (1<<24)

  Virtual address range 0 to KLIMIT is reserved by kernel to store global data structures.

- #define **PLIMIT** (KLIMIT + (1<<24))

  Virtual address range KLIMIT to PLIMIT is reserved by kernel to store data structures specific to the current process.

- #define **PAGE_TABLES_ADDR** KLIMIT

  This is where the page tables are mapped in every address space.

- #define **PAGE_DIRECTORY_ADDR** (KLIMIT + PAGE_TABLE_ENTRIES * PAGE_TABLE_SIZE)

  This is where the page directory is mapped in every address space.

4.6.1 Define Documentation

4.6.1.1 #define KLIMIT (1<<24)

Virtual address range 0 to KLIMIT is reserved by kernel to store global data structures.

Kernel image must be completely inside this region. This region has the same mapping in the address space of all processes. Size must be a multiple of the size described by a single page directory entry (PTE_SIZE * PAGE_SIZE).

Definition at line 28 of file vm.h.

4.6.1.2 #define PAGE_BITS 12

number of bits in virtual address for offset inside page

Definition at line 5 of file vm.h.

Referenced by alloc().

4.6.1.3 #define PAGE_DIRECTORY_ADDR (KLIMIT + PAGE_TABLE_ENTRIES * PAGE_TABLE_SIZE)

This is where the page directory is mapped in every address space.

It must reside in region spanning from KLIMIT to PLIMIT.

Definition at line 46 of file vm.h.

4.6.1.4 #define PAGE_SIZE (1<<PAGE_BITS)

size of page

Definition at line 8 of file vm.h.

Referenced by alloc(), alloc_init(), kinit(), vm_alloc(), vm_map(), vm_valloc(), vm_vfree(), and vm_vfree_block().

4.6.1.5 #define PAGE_TABLE_BITS 10

number of bits in virtual address for page table entry

Definition at line 11 of file vm.h.

4.6.1.6 #define PAGE_TABLE_ENTRIES (1<<PAGE_TABLE_BITS)

number of entries in page table

Definition at line 14 of file vm.h.

Referenced by vm_map().

4.6.1.7 #define PAGE_TABLE_SIZE PAGE_SIZE

size of a page table

Definition at line 17 of file vm.h.

4.6.1.8 #define PAGE_TABLES_ADDR KLIMIT

This is where the page tables are mapped in every address space.

This requires a virtual memory region of size 4M, which must reside completely inside region spanning from KLIMIT to PLIMIT. Must be aligned on a 4M boundary

Definition at line 42 of file vm.h.

4.6.1.9 #define PLIMIT (KLIMIT + (1<<24))

Virtual address range KLIMIT to PLIMIT is reserved by kernel to store data structures specific to the current process.

The mapping of this region changes from one address space to the next. Size must be a multiple of the size described by a single page directory entry (PTE_SIZE * PAGE_SIZE).

Definition at line 36 of file vm.h.

4.6.1.10 #define PTE_SIZE 4

size of a page table entry, in bytes

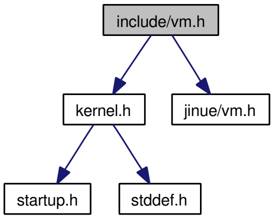
Definition at line 20 of file vm.h.

4.7 include/vm.h File Reference

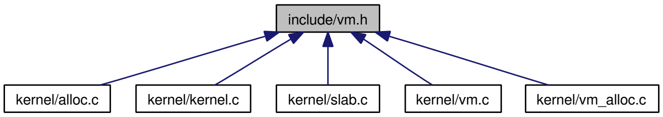
`#include <kernel.h>`

`#include <jinue/vm.h>`

Include dependency graph for vm.h:



This graph shows which files directly or indirectly include this file:



Defines

- #define **PAGE_MASK** (PAGE_SIZE - 1)

  bit mask for offset in page

- #define **PAGE_OFFSET_OF**(x) ((unsigned long)(x) & PAGE_MASK)

  offset in page of virtual address

- #define **PAGE_TABLE_MASK** (PAGE_TABLE_ENTRIES - 1)

  bit mask for page table entry

- #define **PAGE_TABLE_OFFSET_OF**(x) (((unsigned long)(x) >> PAGE_BITS) & PAGE_TABLE_MASK)

  page table entry offset of virtual (linear) address

- #define **PAGE_DIRECTORY_OFFSET_OF**(x) ((unsigned long)(x) >> (PAGE_BITS + PAGE_TABLE_BITS))

  page directory entry offset of virtual (linear address)

- #define **PMAPPING_START** (PAGE_DIRECTORY_ADDR + PAGE_TABLE_SIZE)

  low limit of region spanning from KLIMIT to PLIMIT actually available for mappings

- #define **PMAPPING_END** PLIMIT

  high limit of region spanning from KLIMIT to PLIMIT actually available for mappings

- #define **PAGE_DIRECTORY** ((**pte_t** ∗)PAGE_DIRECTORY_ADDR)

  page directory in virtual memory

- #define **PAGE_TABLES** ((**page_table_t** ∗)PAGE_TABLES_ADDR)

  page tables in virtual memory

- #define **PAGE_TABLE_OF**(x) (PAGE_TABLES[PAGE_DIRECTORY_OFFSET_OF(x)])

  page table in virtual memory

- #define **PDE_OF**(x) (&PAGE_DIRECTORY[PAGE_DIRECTORY_OFFSET_OF(x)])

  address of page directory entry in virtual memory

- #define **PTE_OF**(x) (&PAGE_TABLE_OF(x)[PAGE_TABLE_OFFSET_OF(x)])

  address of page table entry in virtual memory

- #define **PAGE_TABLES_TABLE** (PAGE_TABLE_OF(PAGE_TABLES_ADDR))

  page table which maps all page tables in memory

- #define **PAGE_TABLE_PTE_OF**(x) (&PAGE_TABLES_TABLE[PAGE_DIRECTORY_OFFSET_OF(x)])

  address of page entry in PAGE_ OF_ PAGE_ TABLES

- #define **VM_FLAG_PRESENT** (1<< 0)

  page is present in memory

- #define **VM_FLAG_READ_ONLY** (1<< 1)

  page is read only

- #define **VM_FLAG_KERNEL** 0

  kernel mode page (default)

- #define **VM_FLAG_USER** (1<< 2)

  user mode page

- #define **VM_FLAG_WRITE_THROUGH** (1<< 3)

  write-through cache policy for page

- #define **VM_FLAG_CACHE_DISABLE** (1<< 4)

  uncached page

- #define **VM_FLAG_ACCESSED** (1<< 5)

  page was accessed (read)

- #define **VM_FLAG_DIRTY** (1<< 6)

  page was written to

- #define **VM_FLAG_BIG_PAGE** (1<< 7)

  page directory entry describes a 4M page

- #define **VM_FLAG_GLOBAL** (1<< 8)

  page is global (mapped in every address space)

- #define **VM_FLAGS_PAGE_TABLE** (VM_FLAG_USER | VM_FLAG_READ_ONLY)

  set of flags for a page table (or page directory)

Typedefs

- typedef unsigned long **pte_t**

  type of a page table (or page directory) entry

- typedef **pte_t** **page_table_t** [PAGE_TABLE_ENTRIES]

  type of a page table

Functions

- void **vm_map** (**addr_t** vaddr, **addr_t** paddr, unsigned long flags)

  Map a page frame (physical page) to a virtual memory page.

- void **vm_unmap** (**addr_t** addr)

  Unmap a page from virtual memory.

4.7.1 Define Documentation

4.7.1.1 #define PAGE_DIRECTORY ((pte_t *)PAGE_DIRECTORY_ADDR)

page directory in virtual memory

Definition at line 49 of file vm.h.

4.7.1.2 #define PAGE_DIRECTORY_OFFSET_OF(x) ((unsigned long)(x) >> (PAGE_BITS + PAGE_TABLE_BITS))

page directory entry offset of virtual (linear address)

Definition at line 29 of file vm.h.

4.7.1.3 #define PAGE_MASK (PAGE_SIZE - 1)

bit mask for offset in page

Definition at line 11 of file vm.h.

Referenced by alloc(), slab_prepare(), and vm_free().

4.7.1.4 #define PAGE_OFFSET_OF(x) ((unsigned long)(x) & PAGE_MASK)

offset in page of virtual address

Definition at line 14 of file vm.h.

Referenced by free(), slab_prepare(), vm_map(), vm_unmap(), vm_valloc(), and vm_vfree_block().

4.7.1.5 #define PAGE_TABLE_MASK (PAGE_TABLE_ENTRIES - 1)

bit mask for page table entry

Definition at line 23 of file vm.h.

4.7.1.6 #define PAGE_TABLE_OF(x) (PAGE_TABLES[PAGE_DIRECTORY_OFFSET_OF(x)])

page table in virtual memory

Definition at line 55 of file vm.h.

Referenced by vm_map().

4.7.1.7 #define PAGE_TABLE_OFFSET_OF(x) (((unsigned long)(x) >> PAGE_BITS) & PAGE_TABLE_MASK)

page table entry offset of virtual (linear) address

Definition at line 26 of file vm.h.

4.7.1.8 #define PAGE_TABLE_PTE_OF(x) (&PAGE_TABLES_TABLE[PAGE_DIRECTORY_OFFSET_OF(x)])

address of page entry in PAGE_OF_PAGE_TABLES

Definition at line 67 of file vm.h.

Referenced by vm_map().

4.7.1.9 #define PAGE_TABLES ((page_table_t *)PAGE_TABLES_ADDR)

page tables in virtual memory

Definition at line 52 of file vm.h.

4.7.1.10 #define PAGE_TABLES_TABLE (PAGE_TABLE_OF(PAGE_TABLES_ADDR))

page table which maps all page tables in memory

Definition at line 64 of file vm.h.

4.7.1.11 #define PDE_OF(x) (&PAGE_DIRECTORY[PAGE_DIRECTORY_OFFSET_OF(x)])

address of page directory entry in virtual memory

Definition at line 58 of file vm.h.

Referenced by slab_prepare(), and vm_map().

4.7.1.12 #define PMAPPING_END PLIMIT

high limit of region spanning from KLIMIT to PLIMIT actually available for mappings

Definition at line 43 of file vm.h.

4.7.1.13 #define PMAPPING_START (PAGE_DIRECTORY_ADDR + PAGE_TABLE_SIZE)

low limit of region spanning from KLIMIT to PLIMIT actually available for mappings

Definition at line 39 of file vm.h.

4.7.1.14 #define PTE_OF(x) (&PAGE_TABLE_OF(x)[PAGE_TABLE_OFFSET_OF(x)])

address of page table entry in virtual memory

Definition at line 61 of file vm.h.

Referenced by slab_prepare(), vm_free(), vm_map(), and vm_unmap().

4.7.1.15 #define VM_FLAG_ACCESSED (1<< 5)

page was accessed (read)

Definition at line 91 of file vm.h.

4.7.1.16 #define VM_FLAG_BIG_PAGE (1<< 7)

page directory entry describes a 4M page

Definition at line 97 of file vm.h.

4.7.1.17 #define VM_FLAG_CACHE_DISABLE (1<< 4)

uncached page

Definition at line 88 of file vm.h.

4.7.1.18 #define VM_FLAG_DIRTY (1<< 6)

page was written to

Definition at line 94 of file vm.h.

4.7.1.19 #define VM_FLAG_GLOBAL (1<< 8)

page is global (mapped in every address space)

Definition at line 100 of file vm.h.

4.7.1.20 #define VM_FLAG_KERNEL 0

kernel mode page (default)

Definition at line 79 of file vm.h.

Referenced by vm_vfree_block().

4.7.1.21 #define VM_FLAG_PRESENT (1<< 0)

page is present in memory

Definition at line 73 of file vm.h.

Referenced by slab_prepare(), and vm_map().

4.7.1.22 #define VM_FLAG_READ_ONLY (1<< 1)

page is read only

Definition at line 76 of file vm.h.

4.7.1.23 #define VM_FLAG_USER (1<< 2)

user mode page

Definition at line 82 of file vm.h.

Referenced by vm_map().

4.7.1.24 #define VM_FLAG_WRITE_THROUGH (1<< 3)

write-through cache policy for page

Definition at line 85 of file vm.h.

4.7.1.25 #define VM_FLAGS_PAGE_TABLE (VM_FLAG_USER | VM_FLAG_READ_ONLY)

set of flags for a page table (or page directory)

Definition at line 103 of file vm.h.

Referenced by vm_map().

4.7.2 Typedef Documentation

4.7.2.1 typedef pte_t page_table_t[PAGE_TABLE_ENTRIES]

type of a page table

Definition at line 32 of file vm.h.

4.7.2.2 typedef unsigned long pte_t

type of a page table (or page directory) entry

Definition at line 20 of file vm.h.

4.7.3 Function Documentation

4.7.3.1 void vm_map (addr_t *vaddr*, addr_t *paddr*, unsigned long *flags*)

Map a page frame (physical page) to a virtual memory page.

Parameters:

vaddr virtual address of mapping

paddr address of page frame

flags flags used for mapping (see VM_FLAG_x constants in vm.h)

ASSERTION: we assume vaddr is aligned on a page boundary

ASSERTION: we assume paddr is aligned on a page boundary

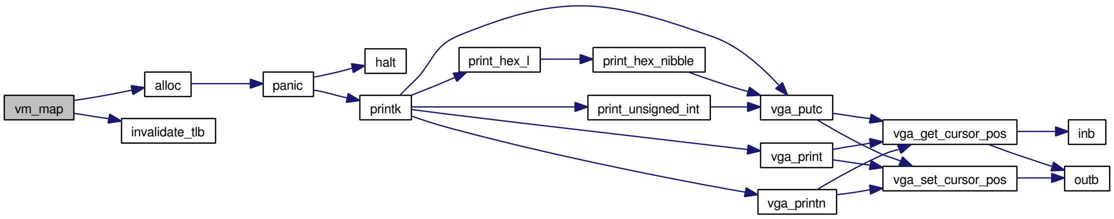
Definition at line 13 of file vm.c.

References alloc(), assert, invalidate_tlb(), PAGE_OFFSET_OF, PAGE_SIZE, PAGE_TABLE_ENTRIES, PAGE_TABLE_OF, PAGE_TABLE_PTE_OF, PDE_OF, PTE_OF, VM_FLAG_PRESENT, VM_FLAG_USER, and VM_FLAGS_PAGE_TABLE.

Referenced by vm_alloc(), and vm_vfree_block().

```
13                                                          {
14     pte_t *pte, *pde;
15     addr_t page_table;
16     int idx;
17 
19     assert( PAGE_OFFSET_OF(vaddr) == 0 );
20 
22     assert( PAGE_OFFSET_OF(paddr) == 0 );
23 
24     /* get page directory entry */
25     pde = PDE_OF(vaddr);
26 
27     /* check if page table must be created */
28     if( !(*pde & VM_FLAG_PRESENT) ) {
29         /* allocate a new page table */
30         page_table = alloc(PAGE_SIZE);
31 
32         /* map page table in the region of memory reserved for that purpose */
33         pte = PAGE_TABLE_PTE_OF(vaddr);
34         *pte = (pte_t)page_table | VM_FLAGS_PAGE_TABLE | VM_FLAG_PRESENT;
35 
36         /* obtain virtual address of new page table */
37         pte = PAGE_TABLE_OF(vaddr);
38 
39         /* invalidate TLB entry for new page table */
40         invalidate_tlb( (addr_t)pte );
41 
42         /* zero content of page table */
43         for(idx = 0; idx < PAGE_TABLE_ENTRIES; ++idx) {
44             pte[idx] = 0;
45         }
46 
47         /* link to page table from page directory */
48         *pde = (pte_t)page_table | VM_FLAG_USER | VM_FLAG_PRESENT;
49     }
50 
51     /* perform the actual mapping */
52     pte = PTE_OF(vaddr);
53     *pte = (pte_t)paddr | flags | VM_FLAG_PRESENT;
54 
55     /* invalidate TLB entry for newly mapped page */
56     invalidate_tlb(vaddr);
57 }
```

Here is the call graph for this function:



4.7.3.2 void vm_unmap (addr_t *addr*)

Unmap a page from virtual memory.

Parameters:

addr address of page to unmap

ASSERTION: we assume addr is aligned on a page boundary

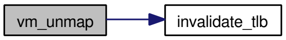
Definition at line 63 of file vm.c.

References assert, invalidate_tlb(), NULL, PAGE_OFFSET_OF, and PTE_OF.

Referenced by vm_free().

```
63                              {
64     pte_t *pte;
65 
67     assert( PAGE_OFFSET_OF(addr) == 0 );
68 
69     pte = PTE_OF(addr);
70     *pte = NULL;
71 
72     invalidate_tlb(addr);
73 }
```

Here is the call graph for this function:

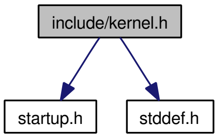


4.8 include/kernel.h File Reference

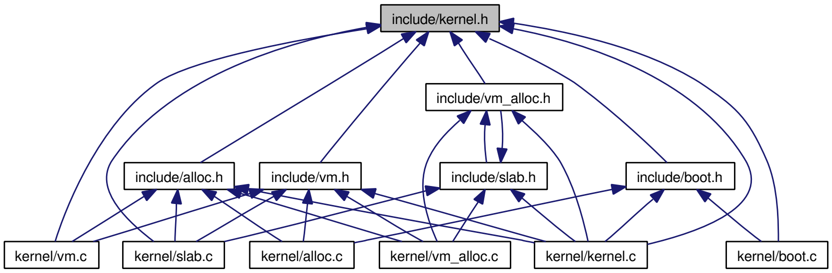
`#include <startup.h>`

`#include <stddef.h>`

Include dependency graph for kernel.h:



This graph shows which files directly or indirectly include this file:



Defines

- #define **kernel_start** ((**addr_t**)start)

Typedefs

- typedef void ∗ **addr_t**
- typedef unsigned long **count_t**

Functions

- void **kernel** (void)
- void **kinit** (void)
- void **idle** (void)

Variables

- **addr_t kernel_top**
- **size_t kernel_size**

4.8.1 Define Documentation

4.8.1.1 #define kernel_start ((addr_t)start)

Definition at line 10 of file kernel.h.

Referenced by alloc_init(), and kinit().

4.8.2 Typedef Documentation

4.8.2.1 typedef void∗ addr_t

Definition at line 7 of file kernel.h.

4.8.2.2 typedef unsigned long count_t

Definition at line 8 of file kernel.h.

4.8.3 Function Documentation

4.8.3.1 void idle (void)

Definition at line 52 of file kernel.c.

Referenced by kernel().

```
52                 {
53     while(1) {}
54 }
```

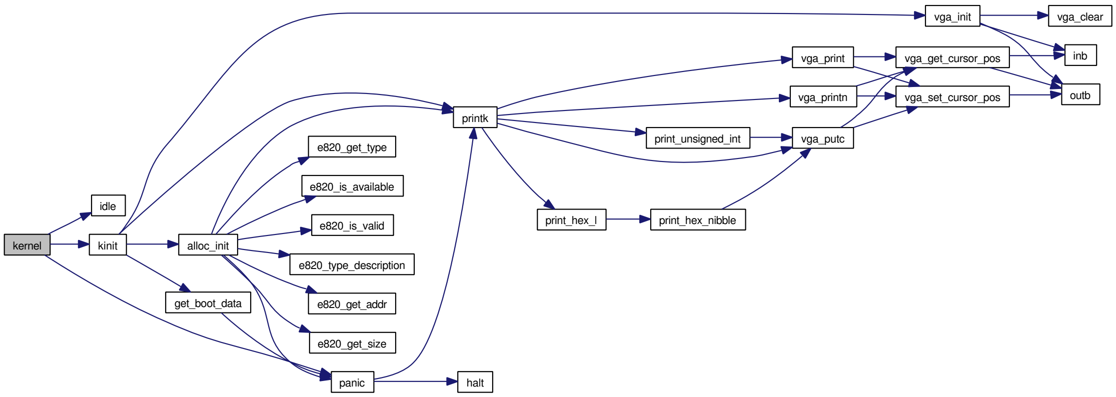
4.8.3.2 void kernel (void)

Definition at line 16 of file kernel.c.

References idle(), kinit(), and panic().

```
16                   {
17     kinit();
18     idle();
19 
20     panic("idle() returned.");
21 }
```

Here is the call graph for this function:



4.8.3.3 void kinit (void)

ASSERTION: we assume the kernel starts on a page boundary

Definition at line 23 of file kernel.c.

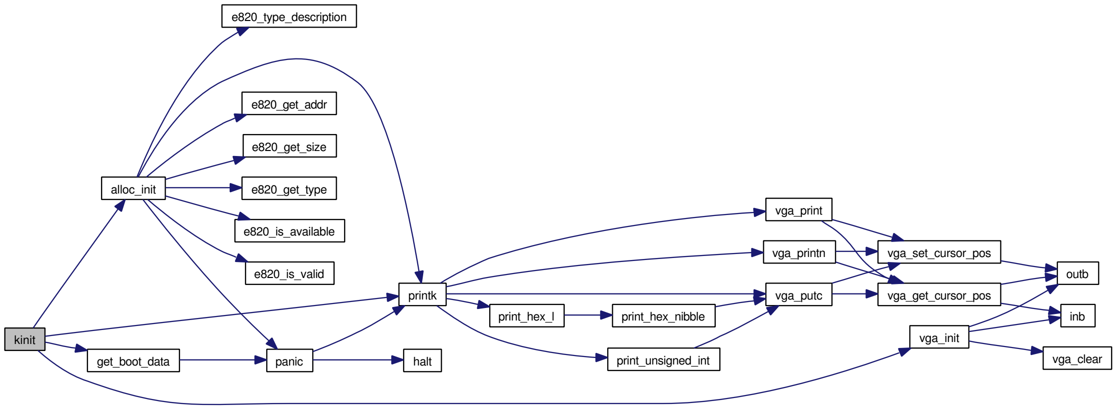
References alloc_init(), assert, get_boot_data(), kernel_size, kernel_start, kernel_top, PAGE_SIZE, printk(), boot_t::sysize, and vga_init().

Referenced by kernel().

```
23                 {
24     boot_t *boot;
25     unsigned int remainder;
26 
27     /* say hello */
28     vga_init();
29     printk("Kernel started.\n");
30 
32     assert((unsigned int)kernel_start % PAGE_SIZE == 0);
33 
34     /* find out kernel size and set kernel_top
35      * (top of kernel, aligned to page boundary) */
36     boot = get_boot_data();
37 
38     kernel_size = boot->sysize * 16;
39     remainder   = kernel_size % PAGE_SIZE;
40 
```

```
    printk("Kernel size is %u (+%u) bytes.\n", kernel_size, PAGE_SIZE - remainder);

    if(remainder != 0) {
        kernel_size += PAGE_SIZE - remainder;
    }
    kernel_top  = kernel_start + kernel_size;

    /* initialize allocator */
    alloc_init();
}
```

Here is the call graph for this function:



4.8.4 Variable Documentation

4.8.4.1 size_t kernel_size

Definition at line 14 of file kernel.c.

Referenced by kinit().

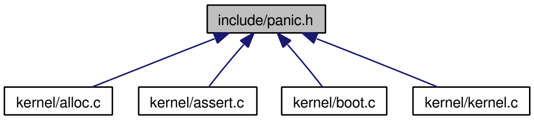
4.8.4.2 addr_t kernel_top

Definition at line 13 of file kernel.c.

Referenced by alloc_init(), and kinit().

4.9 include/panic.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void **panic** (const char ∗message)

4.9.1 Function Documentation

4.9.1.1 void panic (const char ∗ *message*)

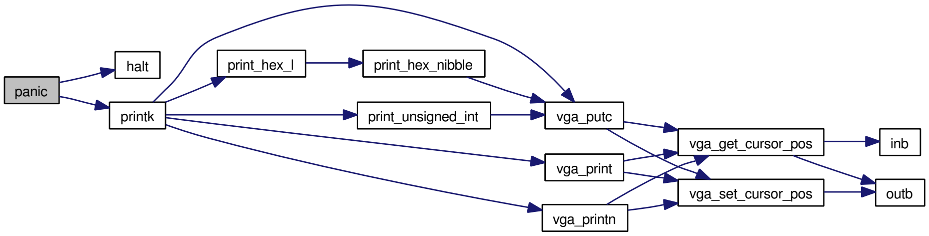
Definition at line 4 of file panic.c.

References halt(), and printk().

Referenced by __assert_failed(), alloc(), alloc_init(), get_boot_data(), and kernel().

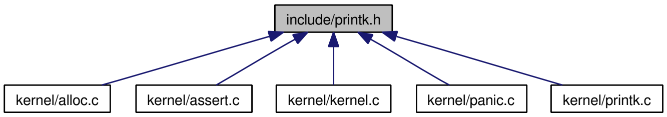
```
4                                {
5     printk("KERNEL PANIC: %s\n", message);
6     halt();
7 }
```

Here is the call graph for this function:



4.10 include/printk.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void **printk** (const char ∗format,...)
- void **print_unsigned_int** (unsigned int n)
- void **print_hex_nibble** (unsigned char byte)
- void **print_hex_b** (unsigned char byte)
- void **print_hex_w** (unsigned short word)
- void **print_hex_l** (unsigned long dword)
- void **print_hex_q** (unsigned long long qword)

4.10.1 Function Documentation

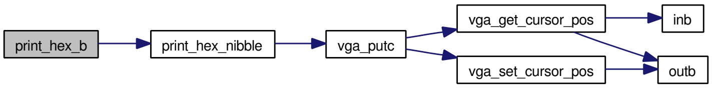
4.10.1.1 void print_hex_b (unsigned char *byte*)

Definition at line 105 of file printk.c.

References print_hex_nibble().

```
105                                    {
106     print_hex_nibble( (char)byte );
107     print_hex_nibble( (char)(byte>>4) );
108 }
```

Here is the call graph for this function:



4.10.1.2 void print_hex_l (unsigned long *dword*)

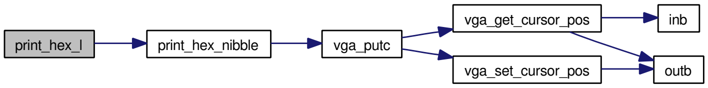
Definition at line 118 of file printk.c.

References print_hex_nibble().

Referenced by printk().

```
118                                       {
119     int off;
120 
121     for(off=32-4; off>=0; off-=4) {
122         print_hex_nibble( (char)(dword>>off) );
123     }
124 }
```

Here is the call graph for this function:



4.10.1.3 void print_hex_nibble (unsigned char *byte*)

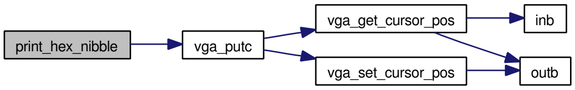
Definition at line 91 of file printk.c.

References vga_putc().

Referenced by print_hex_b(), print_hex_l(), print_hex_q(), and print_hex_w().

```
91                                         {
92     char c;
93 
94     c = byte & 0xf;
95     if(c < 10) {
96         c += '0';
97     }
98     else {
99         c+= ('a' - 10);
100     }
101 
102     vga_putc(c);
103 }
```

Here is the call graph for this function:



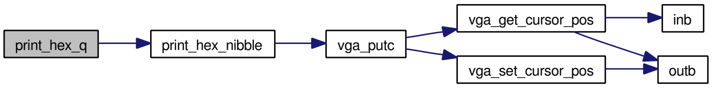
4.10.1.4 void print_hex_q (unsigned long long *qword*)

Definition at line 126 of file printk.c.

References print_hex_nibble().

```
126                                           {
127     int off;
128 
129     for(off=64-4; off>=0; off-=4) {
130         print_hex_nibble( (char)(qword>>off) );
131     }
132 }
```

Here is the call graph for this function:



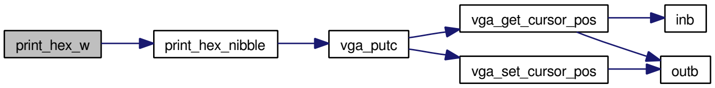
4.10.1.5 void print_hex_w (unsigned short *word*)

Definition at line 110 of file printk.c.

References print_hex_nibble().

```
110                                      {
111     int off;
112 
113     for(off=16-4; off>=0; off-=4) {
114         print_hex_nibble( (char)(word>>off) );
115     }
116 }
```

Here is the call graph for this function:



4.10.1.6 void print_unsigned_int (unsigned int *n*)

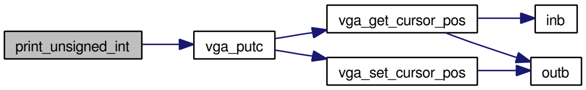
Definition at line 67 of file printk.c.

References vga_putc().

Referenced by printk().

```
67                                         {
68     unsigned int flag = 0;
69     unsigned int pwr;
70     unsigned int digit;
71     char c;
72 
73     if(n == 0) {
74         vga_putc('0');
75         return;
76     }
77 
78     for(pwr = 1000 * 1000 * 1000; pwr > 0; pwr /= 10) {
79         digit = n / pwr;
80 
81         if(digit != 0 || flag) {
82             c = (char)digit + '0';
83             vga_putc(c);
84 
85             flag = 1;
86             n -= digit * pwr;
87         }
88     }
89 }
```

Here is the call graph for this function:



4.10.1.7 void printk (const char * *format*, ...)

Definition at line 6 of file printk.c.

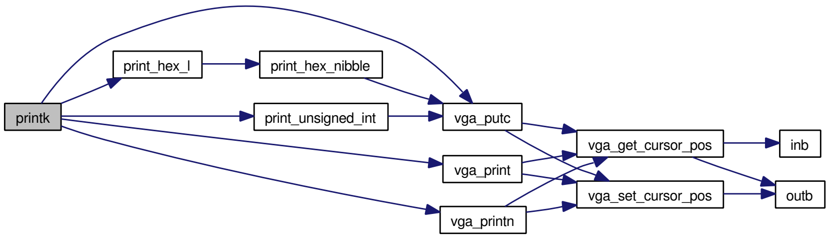
References print_hex_l(), print_unsigned_int(), vm_alloc_t::size, va_arg, va_end, va_start, vga_print(), vga_printn(), and vga_putc().

Referenced by __assert_failed(), alloc_init(), kinit(), and panic().

```
6                                        {
7     va_list ap;
8     const char *idx, *anchor;
9     ptrdiff_t size;
10 
11     va_start(ap, format);
12 
```

```
    idx = format;

    while(1) {
        anchor = idx;

        while( *idx != 0 && *idx != ’%’ ) {
            ++idx;
        }

        size = idx - anchor;

        if(size > 0) {
            vga_printn(anchor, size);
        }

        if(*idx == 0 || *(idx+1) == 0) {
            break;
        }

        ++idx;

        switch( *idx ) {
        case ’%’:
            vga_putc(’%’);
            break;

        case ’c’:
            /* promotion, promotion */
            vga_putc( (char)va_arg(ap, int) );
            break;

        case ’s’:
            vga_print( va_arg(ap, const char *) );
            break;

        case ’u’:
            print_unsigned_int( va_arg(ap, unsigned int) );
            break;

        case ’x’:
            print_hex_l( va_arg(ap, unsigned long) );
            break;

        default:
            va_end(ap);
            return;
        }

        ++idx;
    }

    va_end(ap);
}
```

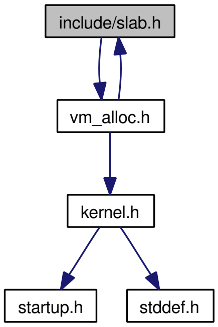
Here is the call graph for this function:



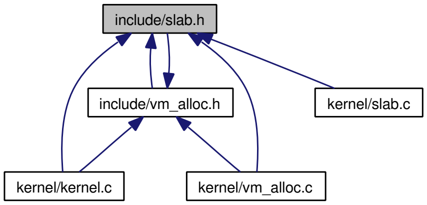
4.11 include/slab.h File Reference

`#include <vm_alloc.h>`

Include dependency graph for slab.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct **slab_header_t**

  header of a slab

- struct **slab_cache_t**

  data structure describing a cache

Typedefs

- typedef struct **slab_header_t slab_header_t**
- typedef struct **slab_cache_t slab_cache_t**

Functions

- void **slab_create** (**slab_cache_t** ∗cache, unsigned long flags)
- void **slab_destroy** (**slab_cache_t** ∗cache)
- **addr_t** **slab_alloc** (**slab_cache_t** ∗cache)
- void **slab_free** (**slab_cache_t** ∗cache, **addr_t** obj)
- void **slab_prepare** (**slab_cache_t** ∗cache, **addr_t** page)

  Prepare a memory page for use as a slab.

- void **slab_add** (**slab_header_t** ∗∗head, **slab_header_t** ∗slab)

  Add a slab to a linked list of slabs.

- void **slab_remove** (**slab_header_t** ∗∗head, **slab_header_t** ∗slab)

  Remove a slab from a linked list of slab.

4.11.1 Typedef Documentation

4.11.1.1 typedef struct slab_cache_t slab_cache_t

Definition at line 47 of file slab.h.

4.11.1.2 typedef struct slab_header_t slab_header_t

Definition at line 21 of file slab.h.

4.11.2 Function Documentation

4.11.2.1 void slab_add (slab_header_t ∗∗ *head*, slab_header_t ∗ *slab*)

Add a slab to a linked list of slabs.

Parameters:

head of list (typically &C->empty, &C->partial or &C->full of some cache C)

slab to add to list

Definition at line 122 of file slab.c.

References slab_header_t::next, NULL, and slab_header_t::prev.

Referenced by slab_alloc(), and vm_vfree_block().

```
122                                                          {
123     slab->next = *head;
124     slab->prev = NULL;
125
126     (*head)->prev = slab;
127     *head = slab;
128 }
```

4.11.2.2 addr_t slab_alloc (slab_cache_t * *cache*)

TODO: handle the NULL pointer

Definition at line 13 of file slab.c.

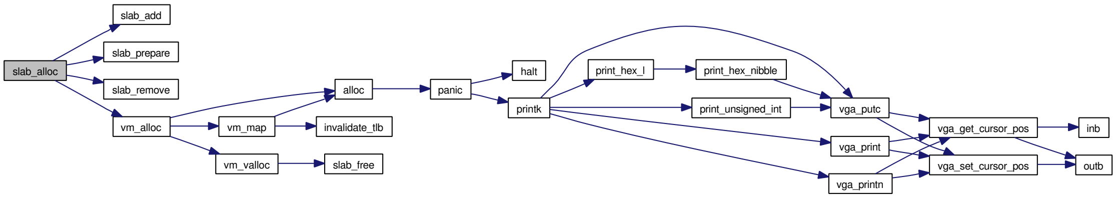
References slab_header_t::available, slab_cache_t::empty, slab_header_t::free_list, slab_cache_t::full, NULL, slab_cache_t::partial, slab_add(), slab_prepare(), slab_remove(), vm_alloc(), slab_cache_t::vm_allocator, and slab_cache_t::vm_flags.

Referenced by vm_vfree_block().

```
13                                      {
14     slab_header_t *slab;
15     addr_t addr;
16
17     /* use a partial slab if one is available... */
18     slab = cache->partial;
19     if(slab != NULL) {
20         addr = slab->free_list;
21         slab->free_list = *(addr_t *)addr;
22
23         /* maybe the slab is now full */
24         if(--slab->available == 0) {
25             slab_remove(&cache->partial, slab);
26             slab_add(&cache->full, slab);
27         }
28
29         return addr;
30     }
31
32     /* ... otherwise, use an empty slab ... */
33     slab = cache->empty;
34     if(slab != NULL) {
35         /* the slab is no longer empty */
36         slab_remove(&cache->empty, slab);
37         slab_add(&cache->partial, slab);
38
39         addr = slab->free_list;
40         slab->free_list = *(addr_t *)addr;
41
42         /* maybe the slab is now full */
43         if(--slab->available == 0) {
44             slab_remove(&cache->partial, slab);
45             slab_add(&cache->full, slab);
```

```
        }

        return addr;
    }

    /* ... and, as last resort, allocate a slab */
    slab = (slab_header_t *)vm_alloc(cache->vm_allocator, cache->vm_flags);
    slab_prepare(cache, (addr_t)slab);

    /* this slab is not empty since we are allocating an object from it */
    slab_add(&cache->partial, slab);

    addr = slab->free_list;
    slab->free_list = *(addr_t *)addr;

    /* maybe the slab is now full */
    if(--slab->available == 0) {
        slab_remove(&cache->partial, slab);
        slab_add(&cache->full, slab);
    }

    return addr;
}
```

Here is the call graph for this function:



4.11.2.3 void slab_create (slab_cache_t * *cache*, unsigned long *flags*)

Definition at line 7 of file slab.c.

```
7                                                           {
8 }
```

4.11.2.4 void slab_destroy (slab_cache_t * *cache*)

Definition at line 10 of file slab.c.

```
10                                          {
11 }
```

4.11.2.5 void slab_free (slab_cache_t * *cache*, addr_t *obj*)

Definition at line 71 of file slab.c.

Referenced by vm_valloc().

```
71                                             {
72 }
```

4.11.2.6 void slab_prepare (slab_cache_t * *cache*, addr_t *page*)

Prepare a memory page for use as a slab.

Initialize fields of the slab header and create the free list.

Parameters:

cache slab cache to which the slab is to be added

page memory page from which to create a slab

ASSERTION: we assume "page" is the starting address of a page

ASSERTION: we assume at least one object can be allocated on slab

ASSERTION: we assume a physical memory page is mapped at "page"

Definition at line 79 of file slab.c.

References assert, slab_header_t::available, slab_header_t::free_list, NULL, slab_cache_t::obj_size, PAGE_MASK, PAGE_OFFSET_OF, PDE_OF, slab_cache_t::per_slab, PTE_OF, and VM_FLAG_PRESENT.

Referenced by slab_alloc(), and vm_vfree_block().

```
79                                                {
80     unsigned int cx;
81     size_t obj_size;
82     count_t per_slab;
83     slab_header_t *slab;
84     addr_t *ptr;
85     addr_t next;
86 
88     assert( PAGE_OFFSET_OF(page) == 0 );
89 
91     assert( cache->per_slab > 0 );
92 
94     assert( (*PDE_OF(page) & ~PAGE_MASK) != NULL &&  (*PDE_OF(page) & VM_FLAG_PRESENT) != 0 );
95     assert( (*PTE_OF(page) & ~PAGE_MASK) != NULL &&  (*PTE_OF(page) & VM_FLAG_PRESENT) != 0 );
96 
97     obj_size = cache->obj_size;
98     per_slab = cache->per_slab;
99 
100     /* initialize slab header */
101     slab = (slab_header_t *)page;
102     slab->available = per_slab;
103     slab->free_list = page + sizeof(slab_header_t);
104 
105     /* create free list */
106     ptr = (addr_t *)slab->free_list;
107 
108     for(cx = 0; cx < per_slab - 1; ++cx) {
109         next = ptr + obj_size;
110         *ptr = next;
111         ptr = (addr_t *)next;
112     }
113 
114     *ptr = NULL;
115 }
```

4.11.2.7 void slab_remove (slab_header_t ** *head*, slab_header_t * *slab*)

Remove a slab from a linked list of slab.

Parameters:

head of list (typically &C->empty, &C->partial or &C->full of some cache C)

slab to remove from list

Definition at line 135 of file slab.c.

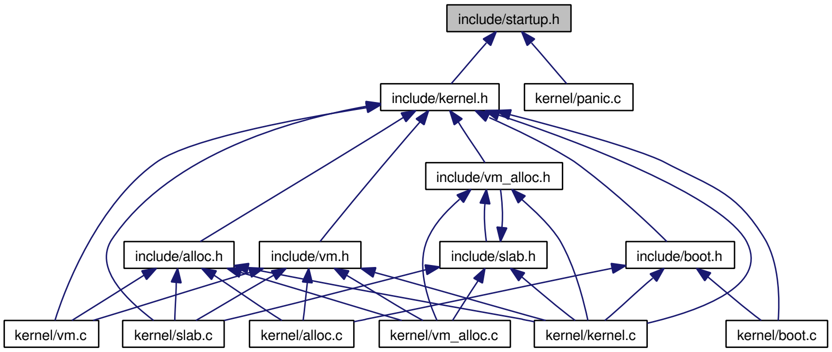
References slab_header_t::next, NULL, and slab_header_t::prev.

Referenced by slab_alloc().

```
135                                                        {
136     if(slab->next != NULL) {
137         slab->next->prev = slab->prev;
138     }
139 
140     if(slab->prev != NULL) {
141         slab->prev->next = slab->next;
142     }
143     else {
144         *head = slab->next;
145     }
146 }
```

4.12 include/startup.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void **start** (void)
- void **halt** (void)

4.12.1 Function Documentation

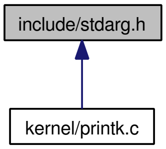
4.12.1.1 void halt (void)

Referenced by panic().

4.12.1.2 void start (void)

4.13 include/stdarg.h File Reference

This graph shows which files directly or indirectly include this file:



Defines

- #define **va_start**(ap, parmN) __builtin_stdarg_start((ap), (parmN))
- #define **va_arg** __builtin_va_arg
- #define **va_end** __builtin_va_end
- #define **va_copy**(dest, src) __builtin_va_copy((dest), (src))

Typedefs

- typedef __builtin_va_list **va_list**

4.13.1 Define Documentation

4.13.1.1 #define va_arg __builtin_va_arg

Definition at line 7 of file stdarg.h.

Referenced by printk().

4.13.1.2 #define va_copy(dest, src) __builtin_va_copy((dest), (src))

Definition at line 9 of file stdarg.h.

4.13.1.3 #define va_end __builtin_va_end

Definition at line 8 of file stdarg.h.

Referenced by printk().

4.13.1.4 #define va_start(ap, parmN) __builtin_stdarg_start((ap), (parmN))

Definition at line 6 of file stdarg.h.

Referenced by printk().

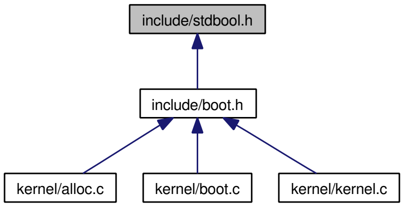
4.13.2 Typedef Documentation

4.13.2.1 typedef __builtin_va_list va_list

Definition at line 4 of file stdarg.h.

4.14 include/stdbool.h File Reference

This graph shows which files directly or indirectly include this file:



Defines

- #define **bool** _Bool
- #define **true** 1
- #define **false** 0
- #define **__bool_true_false_are_defined** 1

4.14.1 Define Documentation

4.14.1.1 #define __bool_true_false_are_defined 1

Definition at line 8 of file stdbool.h.

4.14.1.2 #define bool _Bool

Definition at line 4 of file stdbool.h.

4.14.1.3 #define false 0

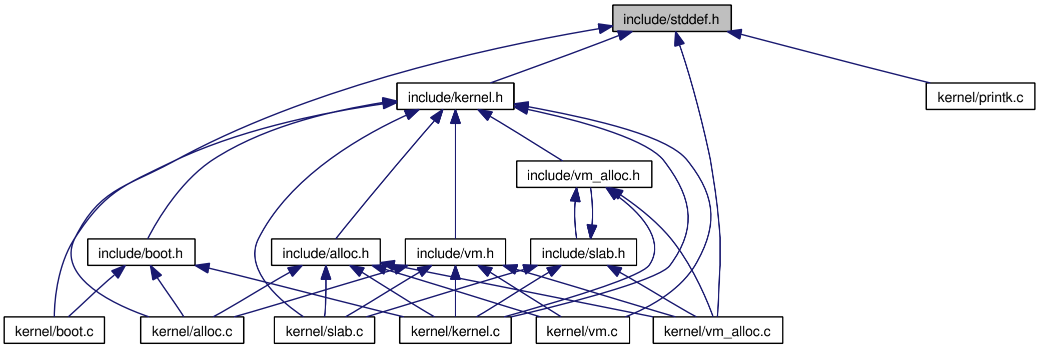
Definition at line 6 of file stdbool.h.

4.14.1.4 #define true 1

Definition at line 5 of file stdbool.h.

4.15 include/stddef.h File Reference

This graph shows which files directly or indirectly include this file:



Defines

- #define **NULL** 0
- #define **offsetof**(type, member) ((**size_t**) &(((type ∗)0) → member))

Typedefs

- typedef signed long **ptrdiff_t**
- typedef unsigned long **size_t**
- typedef int **wchar_t**

4.15.1 Define Documentation

4.15.1.1 #define NULL 0

Definition at line 9 of file stddef.h.

Referenced by slab_add(), slab_alloc(), slab_prepare(), slab_remove(), vm_free(), vm_unmap(), vm_valloc(), and vm_vfree_block().

4.15.1.2 #define offsetof(type, member) ((size_t) &(((type ∗)0) → member))

Definition at line 12 of file stddef.h.

4.15.2 Typedef Documentation

4.15.2.1 typedef signed long ptrdiff_t

Definition at line 4 of file stddef.h.

4.15.2.2 typedef unsigned long size_t

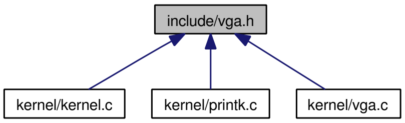
Definition at line 5 of file stddef.h.

4.15.2.3 typedef int wchar_t

Definition at line 6 of file stddef.h.

4.16 include/vga.h File Reference

This graph shows which files directly or indirectly include this file:



Defines

- #define **VGA_TEXT_VID_BASE** 0xb8000
- #define **VGA_MISC_OUT_WR** 0x3c2
- #define **VGA_MISC_OUT_RD** 0x3cc
- #define **VGA_CRTC_ADDR** 0x3d4
- #define **VGA_CRTC_DATA** 0x3d5
- #define **VGA_FB_FLAG_ACTIVE** 1
- #define **VGA_COLOR_BLACK** 0x00
- #define **VGA_COLOR_BLUE** 0x01
- #define **VGA_COLOR_GREEN** 0x02
- #define **VGA_COLOR_CYAN** 0x03
- #define **VGA_COLOR_RED** 0x04
- #define **VGA_COLOR_MAGENTA** 0x05
- #define **VGA_COLOR_BROWN** 0x06
- #define **VGA_COLOR_WHITE** 0x07
- #define **VGA_COLOR_GRAY** 0x08
- #define **VGA_COLOR_BRIGHTBLUE** 0x09
- #define **VGA_COLOR_BRIGHTGREEN** 0x0a
- #define **VGA_COLOR_BRIGHTCYAN** 0x0b
- #define **VGA_COLOR_BRIGHTRED** 0x0c
- #define **VGA_COLOR_BRIGHTMAGENTA** 0x0d
- #define **VGA_COLOR_YELLOW** 0x0e
- #define **VGA_COLOR_BRIGHTWHITE** 0x0f
- #define **VGA_COLOR_DEFAULT** VGA_COLOR_GREEN
- #define **VGA_COLOR_ERASE** VGA_COLOR_RED
- #define **VGA_LINES** 25
- #define **VGA_WIDTH** 80
- #define **VGA_TAB_WIDTH** 8
- #define **VGA_LINE**(x) ((x) / (VGA_WIDTH))
- #define **VGA_COL**(x) ((x) % (VGA_WIDTH))

Typedefs

- typedef unsigned int **vga_pos_t**

Functions

- void **vga_init** (void)
- void **vga_clear** (void)
- void **vga_print** (const char ∗message)
- void **vga_printn** (const char ∗message, unsigned int n)
- void **vga_putc** (char c)
- void **vga_scroll** (void)
- **vga_pos_t vga_get_cursor_pos** (void)
- void **vga_set_cursor_pos** (**vga_pos_t** pos)

4.16.1 Define Documentation

4.16.1.1 #define VGA_COL(x) ((x) % (VGA_WIDTH))

Definition at line 36 of file vga.h.

4.16.1.2 #define VGA_COLOR_BLACK 0x00

Definition at line 12 of file vga.h.

4.16.1.3 #define VGA_COLOR_BLUE 0x01

Definition at line 13 of file vga.h.

4.16.1.4 #define VGA_COLOR_BRIGHTBLUE 0x09

Definition at line 21 of file vga.h.

4.16.1.5 #define VGA_COLOR_BRIGHTCYAN 0x0b

Definition at line 23 of file vga.h.

4.16.1.6 #define VGA_COLOR_BRIGHTGREEN 0x0a

Definition at line 22 of file vga.h.

4.16.1.7 #define VGA_COLOR_BRIGHTMAGENTA 0x0d

Definition at line 25 of file vga.h.

4.16.1.8 #define VGA_COLOR_BRIGHTRED 0x0c

Definition at line 24 of file vga.h.

4.16.1.9 #define VGA_COLOR_BRIGHTWHITE 0x0f

Definition at line 27 of file vga.h.

4.16.1.10 #define VGA_COLOR_BROWN 0x06

Definition at line 18 of file vga.h.

4.16.1.11 #define VGA_COLOR_CYAN 0x03

Definition at line 15 of file vga.h.

4.16.1.12 #define VGA_COLOR_DEFAULT VGA_COLOR_GREEN

Definition at line 28 of file vga.h.

4.16.1.13 #define VGA_COLOR_ERASE VGA_COLOR_RED

Definition at line 29 of file vga.h.

Referenced by vga_clear(), and vga_scroll().

4.16.1.14 #define VGA_COLOR_GRAY 0x08

Definition at line 20 of file vga.h.

4.16.1.15 #define VGA_COLOR_GREEN 0x02

Definition at line 14 of file vga.h.

4.16.1.16 #define VGA_COLOR_MAGENTA 0x05

Definition at line 17 of file vga.h.

4.16.1.17 #define VGA_COLOR_RED 0x04

Definition at line 16 of file vga.h.

4.16.1.18 #define VGA_COLOR_WHITE 0x07

Definition at line 19 of file vga.h.

4.16.1.19 #define VGA_COLOR_YELLOW 0x0e

Definition at line 26 of file vga.h.

4.16.1.20 #define VGA_CRTC_ADDR 0x3d4

Definition at line 7 of file vga.h.

Referenced by vga_get_cursor_pos(), vga_init(), and vga_set_cursor_pos().

4.16.1.21 #define VGA_CRTC_DATA 0x3d5

Definition at line 8 of file vga.h.

Referenced by vga_get_cursor_pos(), vga_init(), and vga_set_cursor_pos().

4.16.1.22 #define VGA_FB_FLAG_ACTIVE 1

Definition at line 10 of file vga.h.

4.16.1.23 #define VGA_LINE(x) ((x) / (VGA_WIDTH))

Definition at line 35 of file vga.h.

4.16.1.24 #define VGA_LINES 25

Definition at line 31 of file vga.h.

Referenced by vga_clear(), and vga_scroll().

4.16.1.25 #define VGA_MISC_OUT_RD 0x3cc

Definition at line 6 of file vga.h.

Referenced by vga_init().

4.16.1.26 #define VGA_MISC_OUT_WR 0x3c2

Definition at line 5 of file vga.h.

Referenced by vga_init().

4.16.1.27 #define VGA_TAB_WIDTH 8

Definition at line 33 of file vga.h.

4.16.1.28 #define VGA_TEXT_VID_BASE 0xb8000

Definition at line 4 of file vga.h.

Referenced by vga_clear(), and vga_scroll().

4.16.1.29 #define VGA_WIDTH 80

Definition at line 32 of file vga.h.

Referenced by vga_clear(), and vga_scroll().

4.16.2 Typedef Documentation

4.16.2.1 typedef unsigned int vga_pos_t

Definition at line 38 of file vga.h.

4.16.3 Function Documentation

4.16.3.1 void vga_clear (void)

Definition at line 25 of file vga.c.

References VGA_COLOR_ERASE, VGA_LINES, VGA_TEXT_VID_BASE, and VGA_WIDTH.

Referenced by vga_init().

```
                     {
    unsigned char *buffer = (unsigned char *)VGA_TEXT_VID_BASE;
    unsigned int idx = 0;

    while( idx < (VGA_LINES * VGA_WIDTH * 2) )  {
        buffer[idx++] = 0x20;
        buffer[idx++] = VGA_COLOR_ERASE;
    }
}
```

4.16.3.2 vga_pos_t vga_get_cursor_pos (void)

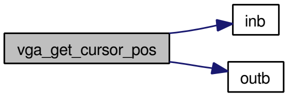
Definition at line 50 of file vga.c.

References inb(), outb(), VGA_CRTC_ADDR, and VGA_CRTC_DATA.

Referenced by vga_print(), vga_printn(), and vga_putc().

```
                                   {
    unsigned char h, l;

    outb(VGA_CRTC_ADDR, 0x0e);
    h = inb(VGA_CRTC_DATA);
    outb(VGA_CRTC_ADDR, 0x0f);
    l = inb(VGA_CRTC_DATA);

    return (h << 8) | l;
}
```

Here is the call graph for this function:



4.16.3.3 void vga_init (void)

Definition at line 7 of file vga.c.

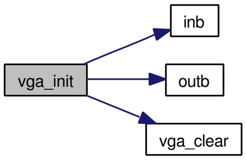
References inb(), outb(), vga_clear(), VGA_CRTC_ADDR, VGA_CRTC_DATA, VGA_MISC_OUT_RD, and VGA_MISC_OUT_WR.

Referenced by kinit().

```
                   {
```

```
8     unsigned char data;
9
10     /* Set address select bit in a known state: CRTC regs at 0x3dx */
11     data = inb(VGA_MISC_OUT_RD);
12     data |= 1;
13     outb(VGA_MISC_OUT_WR, data);
14
15     /* Move cursor to line 0 col 0 */
16     outb(VGA_CRTC_ADDR, 0x0e);
17     outb(VGA_CRTC_DATA, 0x0);
18     outb(VGA_CRTC_ADDR, 0x0f);
19     outb(VGA_CRTC_DATA, 0x0);
20
21     /* Clear the screen */
22     vga_clear();
23 }
```

Here is the call graph for this function:



4.16.3.4 void vga_print (const char * *message*)

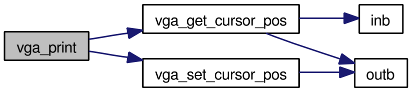
Definition at line 72 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by printk().

```
72                                       {
73     unsigned short int pos = vga_get_cursor_pos();
74     char c;
75
76     while( (c = *(message++)) ) {
77         pos = vga_raw_putc(c, pos);
78     }
79
80     vga_set_cursor_pos(pos);
81 }
```

Here is the call graph for this function:



4.16.3.5 void vga_printn (const char * *message*, unsigned int *n*)

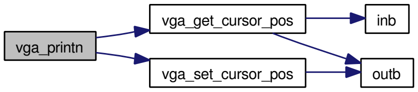
Definition at line 83 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by printk().

```
83                                                  {
84     vga_pos_t pos = vga_get_cursor_pos();
85     char c;
86 
87     while(n) {
88         c = *(message++);
89         pos = vga_raw_putc(c, pos);
90         --n;
91     }
92 
93     vga_set_cursor_pos(pos);
94 }
```

Here is the call graph for this function:



4.16.3.6 void vga_putc (char *c*)

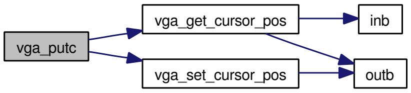
Definition at line 96 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by print_hex_nibble(), print_unsigned_int(), and printk().

```
96                       {
97     vga_pos_t pos = vga_get_cursor_pos();
98 
99     pos = vga_raw_putc(c, pos);
100 
101     vga_set_cursor_pos(pos);
102 }
```

Here is the call graph for this function:



4.16.3.7 void vga_scroll (void)

Definition at line 35 of file vga.c.

References VGA_COLOR_ERASE, VGA_LINES, VGA_TEXT_VID_BASE, and VGA_WIDTH.

```
35                      {
36     unsigned char *di = (unsigned char *)VGA_TEXT_VID_BASE;
37     unsigned char *si = (unsigned char *)(VGA_TEXT_VID_BASE + 2 * VGA_WIDTH);
38     unsigned int idx;
39 
40     for(idx = 0; idx < 2 * VGA_WIDTH * (VGA_LINES - 1); ++idx) {
41         *(di++) = *(si++);
42     }
43 
44     for(idx = 0; idx < VGA_WIDTH; ++idx) {
45         *(di++) = 0x20;
46         *(di++) = VGA_COLOR_ERASE;
47     }
48 }
```

4.16.3.8 void vga_set_cursor_pos (vga_pos_t *pos*)

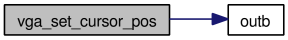
Definition at line 61 of file vga.c.

References outb(), VGA_CRTC_ADDR, and VGA_CRTC_DATA.

Referenced by vga_print(), vga_printn(), and vga_putc().

```
61                                        {
62     unsigned char h = pos >> 8;
63     unsigned char l = pos;
64 
65     outb(VGA_CRTC_ADDR, 0x0e);
66     outb(VGA_CRTC_DATA, h);
67     outb(VGA_CRTC_ADDR, 0x0f);
68     outb(VGA_CRTC_DATA, l);
69 }
```

Here is the call graph for this function:

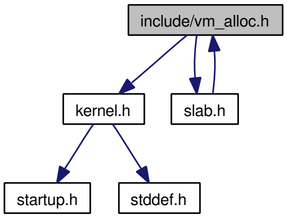


4.17 include/vm_alloc.h File Reference

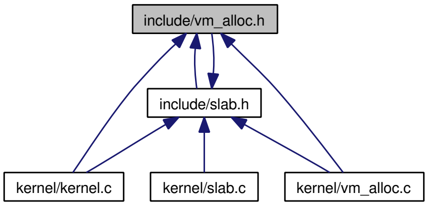
`#include <kernel.h>`

`#include <slab.h>`

Include dependency graph for vm_alloc.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct **vm_link_t**

  links forming the linked lists of free virtual memory pages

- struct **vm_alloc_t**

  data structure which keep tracks of free pages in a region of virtual memory

Typedefs

- typedef struct **vm_link_t** **vm_link_t**
- typedef struct **vm_alloc_t** **vm_alloc_t**

Functions

- **addr_t vm_valloc** (**vm_alloc_t** *pool)

  Allocate a page of virtual memory (not backed by physical memory).

- void **vm_vfree** (**vm_alloc_t** *pool, **addr_t** addr)

  Return a single page of virtual memory to a pool of available pages.

- void **vm_vfree_block** (**vm_alloc_t** *pool, **addr_t** addr, **size_t** size)

  Return a block of contiguous virtual memory pages to a pool of available pages.

- **addr_t vm_alloc** (**vm_alloc_t** *pool, unsigned long flags)

  Allocate a physical memory page and map it in virtual memory.

- void **vm_free** (**vm_alloc_t** *pool, **addr_t** addr)

  *Free a physical page mapped in virtual memory (which was typically obtained through a call to **vm_map()** (p. 41)).*

4.17.1 Typedef Documentation

4.17.1.1 typedef struct vm_alloc_t vm_alloc_t

Definition at line 34 of file vm_alloc.h.

4.17.1.2 typedef struct vm_link_t vm_link_t

Definition at line 19 of file vm_alloc.h.

4.17.2 Function Documentation

4.17.2.1 addr_t vm_alloc (vm_alloc_t * *pool*, unsigned long *flags*)

Allocate a physical memory page and map it in virtual memory.

Parameters:

pool data structure managing the virtual memory region in which page will be mapped

flags flags for page mapping (passed as-is to **vm_map()** (p. 41))

TODO: handle the NULL pointer

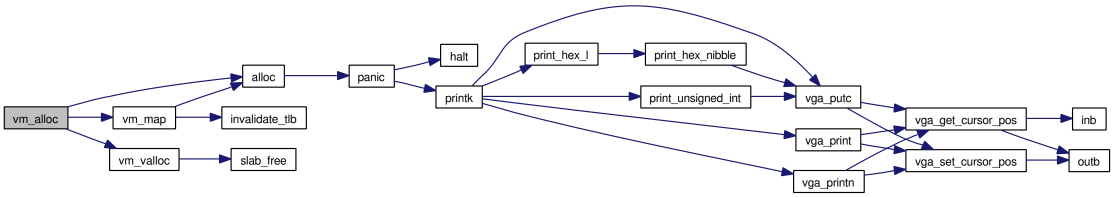
Definition at line 135 of file vm_alloc.c.

References alloc(), PAGE_SIZE, vm_map(), and vm_valloc().

Referenced by slab_alloc().

```
135                                                       {
136     addr_t paddr, vaddr;
137 
140     vaddr = vm_valloc(pool);
141     paddr = alloc(PAGE_SIZE);
142     vm_map(vaddr, paddr, flags);
143 
144     return vaddr;
145 }
```

Here is the call graph for this function:



4.17.2.2 void vm_free (vm_alloc_t * *pool*, addr_t *addr*)

Free a physical page mapped in virtual memory (which was typically obtained through a call to **vm_map()** (p. 41)).

The physical memory is freed and the virtual page is returned to the virtual address space allocator.

Parameters:

pool data structure managing the virtual memory region to which the page is returned address of page to free

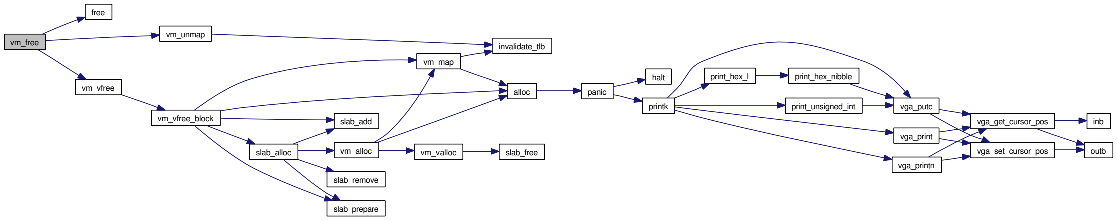
ASSERTION: address of page should not be the null pointer

Definition at line 154 of file vm_alloc.c.

References assert, free(), NULL, PAGE_MASK, PTE_OF, vm_unmap(), and vm_vfree().

```
154                                       {
    addr_t paddr;

    assert( addr != (addr_t)NULL );

    paddr = (addr_t)(*PTE_OF(addr) | ~PAGE_MASK);

    vm_unmap(addr);
    vm_vfree(pool, addr);
    free(paddr);
}
```

Here is the call graph for this function:



4.17.2.3 addr_t vm_valloc (vm_alloc_t ∗ *pool*)

Allocate a page of virtual memory (not backed by physical memory).

This page may then be used for temporary mappings, for example. Page is allocated from a specific virtual memory region managed by a **vm_alloc_t** (p. 13) data structure.

Parameters:

pool data structure managing the virtual memory region from which to allocate

Returns:

address of allocated page

ASSERTION: block size should be an integer number of pages

ASSERTION: returned address should be aligned with a page boundary

Definition at line 17 of file vm_alloc.c.

References vm_link_t::addr, assert, vm_alloc_t::cache, vm_alloc_t::head, vm_link_t::next, NULL, PAGE_OFFSET_OF, PAGE_SIZE, vm_link_t::size, vm_alloc_t::size, and slab_free().

Referenced by vm_alloc().

```
                                   {
    addr_t addr;
    vm_link_t *head;
    size_t size;

    head = pool->head;

    /* no page available */
    if(head == (addr_t)NULL) {
        return (addr_t)NULL;
    }

    addr = head->addr;
    size = head->size - PAGE_SIZE;

    assert( PAGE_OFFSET_OF(size) == 0 );

    /* if block is made of only one page, we remove it from the free list */
    if(size == 0) {
        pool->head = head->next;
        slab_free(pool->cache, head);
    }
    else {
        head->size = size;
        head->addr += PAGE_SIZE;
    }

    assert( PAGE_OFFSET_OF(addr) == 0 );

    return addr;
}
```

Here is the call graph for this function:

vm_valloc
slab_free

4.17.2.4 void vm_vfree (vm_alloc_t * *pool*, addr_t *addr*)

Return a single page of virtual memory to a pool of available pages.

Should not be used to free pages to which physical memory is still mapped (no physical memory is freed by this function). Use this function to return pages obtained by a call to **vm_valloc()** (p. 79) (and not **vm_alloc()** (p. 77)).

Parameters:

pool data structure managing the relevant virtual memory region

addr address of virtual page which must be freed

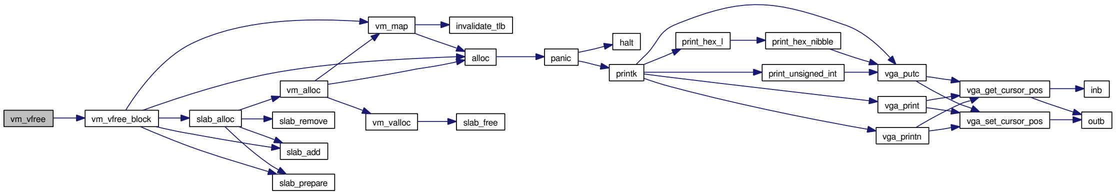
Definition at line 59 of file vm_alloc.c.

References PAGE_SIZE, and vm_vfree_block().

Referenced by vm_free().

```
                                         {
   vm_vfree_block(pool, addr, PAGE_SIZE);
}
```

Here is the call graph for this function:



4.17.2.5 void vm_vfree_block (vm_alloc_t * *pool*, addr_t *addr*, size_t *size*)

Return a block of contiguous virtual memory pages to a pool of available pages.

Should not be used to free pages to which physical memory is still mapped (no physical memory is freed by this function).

Parameters:

- ***pool*** data structure managing the relevant virtual memory region
- ***addr*** starting address of virtual memory block
- ***size*** size of block

ASSERTION: we assume starting address is aligned on a page boundary

ASSERTION: we assume size of block is an integer number of pages

ASSERTION: address of block should not be the null pointer

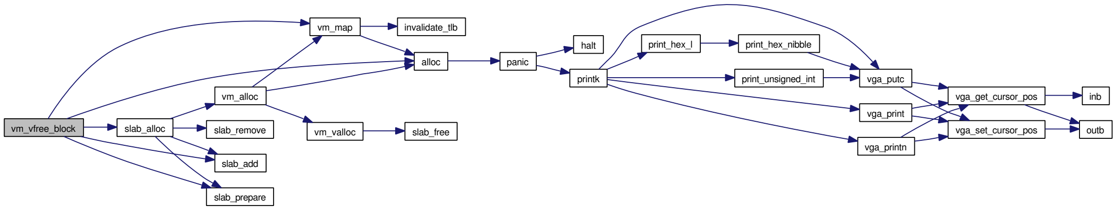
Definition at line 71 of file vm_alloc.c.

References vm_link_t::addr, alloc(), assert, vm_alloc_t::cache, slab_cache_t::empty, vm_alloc_t::head, vm_link_t::next, NULL, PAGE_OFFSET_OF, PAGE_SIZE, slab_cache_t::partial, vm_link_t::size, slab_add(), slab_alloc(), slab_prepare(), slab_cache_t::vm_allocator, VM_FLAG_KERNEL, and vm_map().

Referenced by vm_vfree().

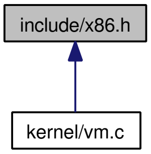
```
                                                                {
   addr_t phys_page;
   vm_link_t *link;

   assert( PAGE_OFFSET_OF(addr) == 0 );

   assert( PAGE_OFFSET_OF(size) == 0 );

   assert( addr != (addr_t)NULL );

   /*  The virtual address space allocator needs a slab cache from which to
       allocate data structures for its free list. Also, each slab cache needs
       a virtual address space allocator to allocate slabs when needed.

       There can be a mutual dependency between the virtual address space
       allocator and the slab cache. This is not a problem in general, but a
       special bootstrapping procedure is needed for initialization of the
       virtual address space allocator in that case. The virtual address space
       allocator will actually "donate" a virtual page (backed by physical ram)
       to the cache for use as a slab.

       This case is handled here
    */
   if(pool->head == NULL) {
       if(pool->cache->vm_allocator == pool) {
           if(pool->cache->empty == NULL && pool->cache->partial == NULL) {
                /* allocate a physical page for slab */
                phys_page = alloc(PAGE_SIZE);

                /* map page */
                vm_map(addr, phys_page, VM_FLAG_KERNEL);

                /* prepare the slab and add it to cache empty list */
                slab_prepare(pool->cache, addr);
                slab_add(&pool->cache->empty, addr);

                size -= PAGE_SIZE;

                /* if the block contained only one page, we have nothing left
                   to free */
                if(size == 0) {
                    return;
                }

                addr += PAGE_SIZE;
            }
        }
    }

    link = (vm_link_t *)slab_alloc(pool->cache);
    link->size = size;
    link->addr = addr;

    link->next = pool->head;
    pool->head = link;
}
```

Here is the call graph for this function:



4.18 include/x86.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void **invalidate_tlb** (**addr_t** vaddr)

4.18.1 Function Documentation

4.18.1.1 void invalidate_tlb (addr_t *vaddr*)

Referenced by vm_map(), and vm_unmap().

4.19 kernel/alloc.c File Reference

`#include <alloc.h>`

`#include <assert.h>`

`#include <boot.h>`

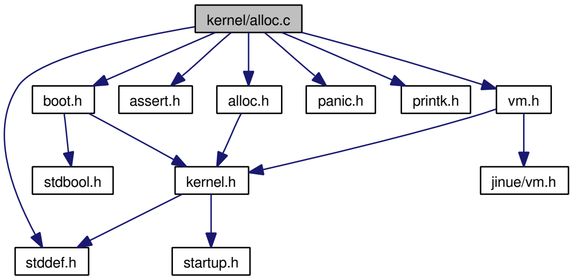
`#include <panic.h>`

`#include <printk.h>`

`#include <stddef.h>`

`#include <vm.h>`

Include dependency graph for alloc.c:



Functions

- void **alloc_init** (void)
- **addr_t alloc** (**size_t** size)
- void **free** (**addr_t** addr)

4.19.1 Function Documentation

4.19.1.1 addr_t alloc (size_t *size*)

ASSERTION: returned address should be aligned with a page boundary

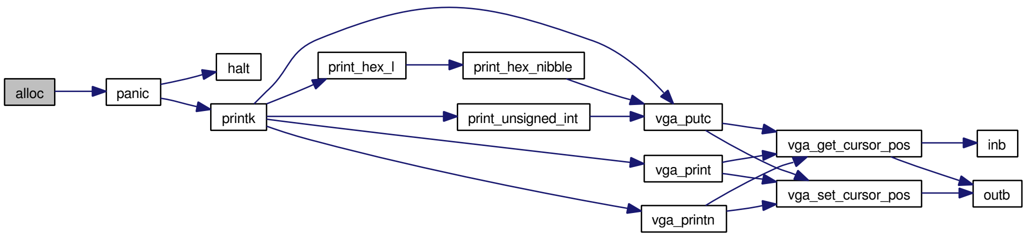
Definition at line 96 of file alloc.c.

References assert, PAGE_BITS, PAGE_MASK, PAGE_SIZE, and panic().

Referenced by vm_alloc(), vm_map(), and vm_vfree_block().

```
96                        {
97     addr_t addr;
98      size_t pages;
99
100      pages = size >> PAGE_BITS;
101
102      if( (size & PAGE_MASK) != 0 ) {
103          ++pages;
104      }
105
106      if(_alloc_size < pages) {
107          panic("out of memory.");
108      }
109
110      addr = _alloc_addr;
111      _alloc_addr += pages * PAGE_SIZE;
112      _alloc_size -= pages;
113
115      assert( ((unsigned long)addr & PAGE_MASK) == 0 );
116
117      return addr;
118 }
```

Here is the call graph for this function:



4.19.1.2 void alloc_init (void)

Definition at line 12 of file alloc.c.

References e820_get_addr(), e820_get_size(), e820_get_type(), e820_is_available(), e820_is_valid(), e820_type_description(), kernel_start, kernel_top, PAGE_SIZE, panic(), printk(), and vm_alloc_t::size.

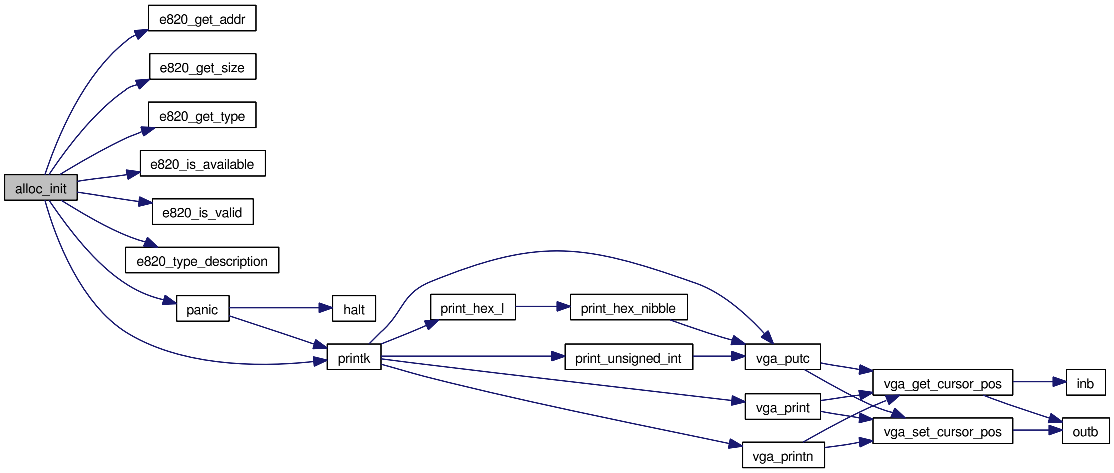
Referenced by kinit().

```
12                      {
13      unsigned int idx;
14      unsigned int remainder;
15      bool avail;
16      size_t size;
17      e820_type_t type;
18      addr_t addr, fixed_addr, best_addr;
```

```
    size_t fixed_size, best_size;

    idx = 0;
    best_size = 0;

    printk("Dump of the BIOS memory map:\n");
    printk("  address  size     type\n");
    while( e820_is_valid(idx) ) {
        addr = e820_get_addr(idx);
        size = e820_get_size(idx);
        type = e820_get_type(idx);
        avail = e820_is_available(idx);

        ++idx;

        printk("%c %x %x %s\n",
            avail?’*’:’ ’,
            addr,
            size,
            e820_type_description(type) );

        if( !avail ) {
            continue;
        }

        fixed_addr = addr;
        fixed_size = size;

        /* is the region completely under the kernel ? */
        if(addr + size > kernel_start) {
            /* is the region completely above the kernel ? */
            if(addr < kernel_top) {
                /* if the region touches the kernel, we take only
                 * the part above the kernel, if there is one... */
                if(addr + size <= kernel_top) {
                    /* ... and apparently, there is none */
                    continue;
                }

                fixed_addr = kernel_top;
                fixed_size -= fixed_addr - addr;
            }
        }

        /* we must make sure the starting address is aligned on a
         * page boundary. The size will eventually be divided
         * by the page size, and thus need not be aligned. */
        remainder = (unsigned int)fixed_addr % PAGE_SIZE;
        if(remainder != 0) {
            remainder = PAGE_SIZE - remainder;
            if(fixed_size < remainder) {
                continue;
            }

            fixed_addr += remainder;
            fixed_size -= remainder;
        }
```

```
76
77         if(fixed_size > best_size) {
78             best_addr = fixed_addr;
79             best_size = fixed_size;
80         }
81     }
82
83     _alloc_addr = (addr_t)best_addr;
84     _alloc_size = best_size / PAGE_SIZE;
85
86     if(_alloc_size == 0) {
87         panic("no memory to allocate.");
88     }
89
90     printk("%u kilobytes (%u pages) available starting at %xh.\n",
91         _alloc_size * PAGE_SIZE / 1024,
92         _alloc_size,
93         _alloc_addr );
94 }
```

Here is the call graph for this function:



4.19.1.3 void free (addr_t *addr*)

ASSERTION: we assume starting address is aligned on a page boundary

Definition at line 120 of file alloc.c.

References assert, and PAGE_OFFSET_OF.

Referenced by vm_free().

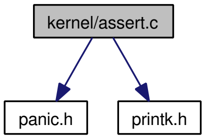
```
120                        {
    assert( PAGE_OFFSET_OF(addr) == 0 );
}
```

4.20 kernel/assert.c File Reference

`#include <panic.h>`

`#include <printk.h>`

Include dependency graph for assert.c:



Functions

- void **__assert_failed** (const char ∗expr, const char ∗file, unsigned int line, const char ∗func)

4.20.1 Function Documentation

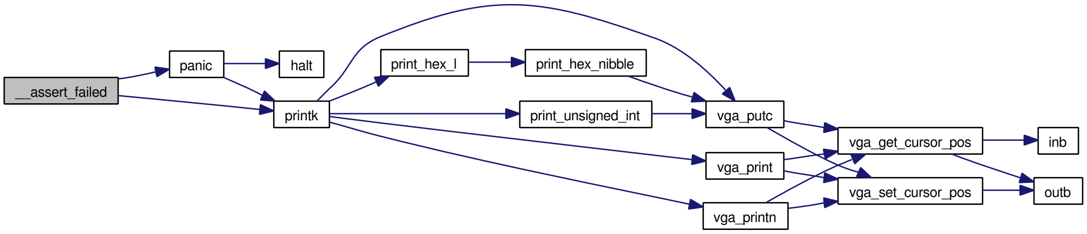
4.20.1.1 void __assert_failed (const char ∗ *expr*, const char ∗ *file*, unsigned int *line*, const char ∗ *func*)

Definition at line 5 of file assert.c.

References panic(), and printk().

```
                      {

    printk(
        "ASSERTION FAILED [%s]: %s at line %u in function %s.\n",
        expr, file, line, func );

    panic("Assertion failed.");
}
```

Here is the call graph for this function:



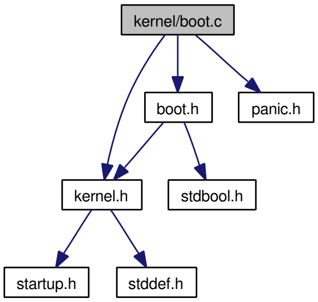
4.21 kernel/boot.c File Reference

`#include <boot.h>`

`#include <kernel.h>`

`#include <panic.h>`

Include dependency graph for boot.c:



Functions

- **addr_t e820_get_addr** (unsigned int idx)
- **size_t e820_get_size** (unsigned int idx)
- **e820_type_t e820_get_type** (unsigned int idx)
- bool **e820_is_valid** (unsigned int idx)
- bool **e820_is_available** (unsigned int idx)
- const char ∗ **e820_type_description** (**e820_type_t** type)
- **boot_t** ∗ **get_boot_data** (void)

Variables

- **e820_t** ∗ **e820_map**
- **addr_t boot_setup_addr**

4.21.1 Function Documentation

4.21.1.1 addr_t e820_get_addr (unsigned int *idx*)

Definition at line 8 of file boot.c.

Referenced by alloc_init().

```
8                                         {
9     return (addr_t)(unsigned long)e820_map[idx].addr;
10 }
```

4.21.1.2 size_t e820_get_size (unsigned int *idx*)

Definition at line 12 of file boot.c.

References e820_t::size.

Referenced by alloc_init().

```
12                                         {
13     return (size_t)e820_map[idx].size;
14 }
```

4.21.1.3 e820_type_t e820_get_type (unsigned int *idx*)

Definition at line 16 of file boot.c.

References e820_t::type.

Referenced by alloc_init().

```
16                                              {
17     return e820_map[idx].type;
18 }
```

4.21.1.4 bool e820_is_available (unsigned int *idx*)

Definition at line 24 of file boot.c.

References E820_RAM.

Referenced by alloc_init().

```
24                                           {
25     return (e820_map[idx].type == E820_RAM);
26 }
```

4.21.1.5 bool e820_is_valid (unsigned int *idx*)

Definition at line 20 of file boot.c.

References vm_alloc_t::size.

Referenced by alloc_init().

```
20                                 {
21     return (e820_map[idx].size != 0);
22 }
```

4.21.1.6 const char* e820_type_description (e820_type_t *type*)

Definition at line 28 of file boot.c.

References E820_ACPI, E820_RAM, and E820_RESERVED.

Referenced by alloc_init().

```
28                                                {
29     switch(type) {
30 
31     case E820_RAM:
32         return "available";
33 
34     case E820_RESERVED:
35         return "unavailable/reserved";
36 
37     case E820_ACPI:
38         return "unavailable/acpi";
39 
40     default:
41         return "unavailable/other";
42     }
43 }
```

4.21.1.7 boot_t* get_boot_data (void)

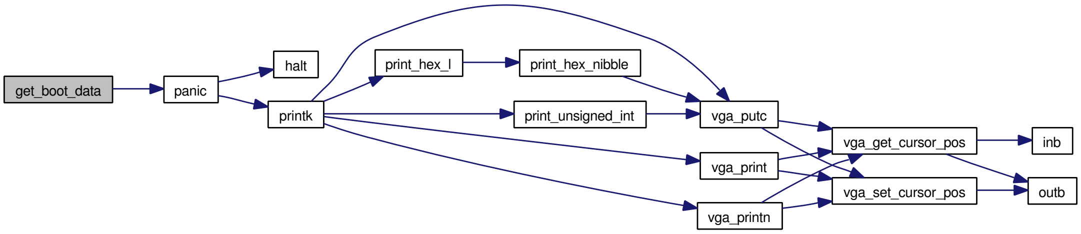
Definition at line 45 of file boot.c.

References BOOT_MAGIC, boot_setup_addr, BOOT_SIGNATURE, boot_t::magic, panic(), and boot_t::signature.

Referenced by kinit().

```
45                         {
46     boot_t *boot;
47 
48     boot = (boot_t *)( boot_setup_addr - sizeof(boot_t) );
49 
50     if(boot->signature != BOOT_SIGNATURE) {
51         panic("bad boot sector signature.");
52     }
53 
54     if(boot->magic != BOOT_MAGIC) {
55         panic("bad boot sector magic.");
56     }
57 
58      return boot;
59 }
```

Here is the call graph for this function:



4.21.2 Variable Documentation

4.21.2.1 addr_t boot_setup_addr

Definition at line 6 of file boot.c.

Referenced by get_boot_data().

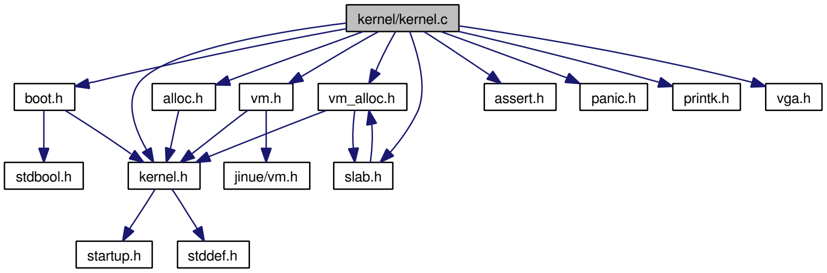
4.21.2.2 e820_t∗ e820_map

Definition at line 5 of file boot.c.

4.22 kernel/kernel.c File Reference

```
#include <alloc.h>
#include <assert.h>
#include <boot.h>
#include <kernel.h>
#include <panic.h>
#include <printk.h>
#include <vga.h>
#include <vm.h>
#include <vm_alloc.h>
#include <slab.h>
```

Include dependency graph for kernel.c:



Functions

- void **kernel** (void)
- void **kinit** (void)
- void **idle** (void)

Variables

- **addr_t kernel_top**
- **size_t kernel_size**

4.22.1 Function Documentation

4.22.1.1 void idle (void)

Definition at line 52 of file kernel.c.

Referenced by kernel().

```
52                 {
53     while(1) {}
54 }
```

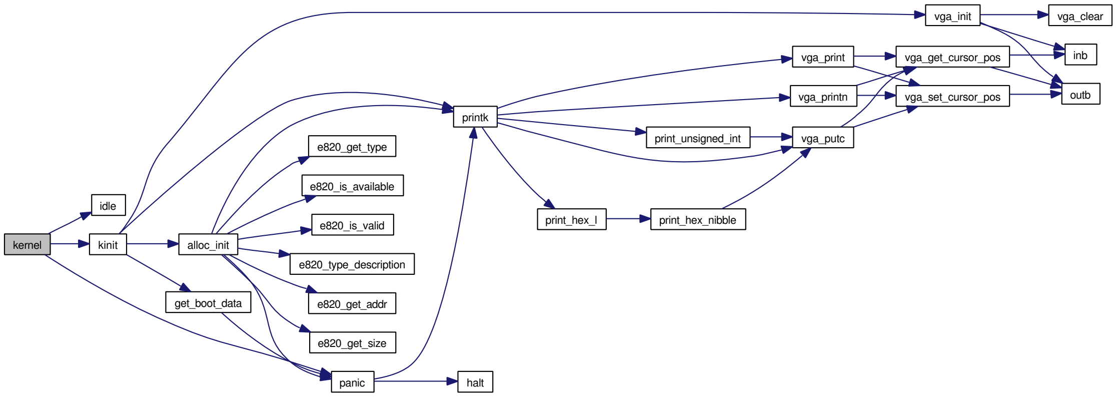
4.22.1.2 void kernel (void)

Definition at line 16 of file kernel.c.

References idle(), kinit(), and panic().

```
16                   {
17     kinit();
18     idle();
19
20     panic("idle() returned.");
21 }
```

Here is the call graph for this function:



4.22.1.3 void kinit (void)

ASSERTION: we assume the kernel starts on a page boundary

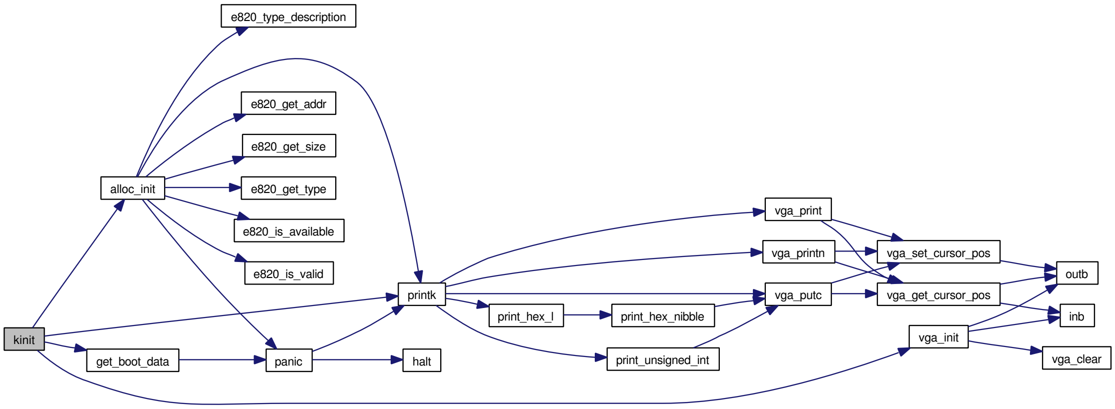
Definition at line 23 of file kernel.c.

References alloc_init(), assert, get_boot_data(), kernel_size, kernel_start, kernel_top, PAGE_SIZE, printk(), boot_t::sysize, and vga_init().

Referenced by kernel().

```
23                 {
24     boot_t *boot;
25     unsigned int remainder;
26 
27     /* say hello */
28     vga_init();
29     printk("Kernel started.\n");
30 
32     assert((unsigned int)kernel_start % PAGE_SIZE == 0);
33 
34     /* find out kernel size and set kernel_top
35      * (top of kernel, aligned to page boundary) */
36     boot = get_boot_data();
37 
38     kernel_size = boot->sysize * 16;
39     remainder   = kernel_size % PAGE_SIZE;
40 
41     printk("Kernel size is %u (+%u) bytes.\n", kernel_size, PAGE_SIZE - remainder);
42 
43     if(remainder != 0) {
44         kernel_size += PAGE_SIZE - remainder;
45     }
46     kernel_top  = kernel_start + kernel_size;
47 
48     /* initialize allocator */
49     alloc_init();
50 }
```

Here is the call graph for this function:



4.22.2 Variable Documentation

4.22.2.1 size_t kernel_size

Definition at line 14 of file kernel.c.

Referenced by kinit().

4.22.2.2 addr_t kernel_top

Definition at line 13 of file kernel.c.

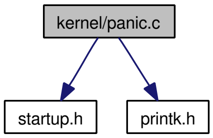
Referenced by alloc_init(), and kinit().

4.23 kernel/panic.c File Reference

`#include <startup.h>`

`#include <printk.h>`

Include dependency graph for panic.c:



Functions

- void **panic** (const char ∗message)

4.23.1 Function Documentation

4.23.1.1 void panic (const char ∗ *message*)

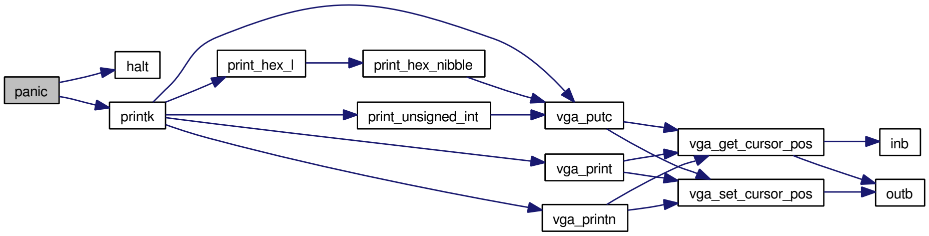
Definition at line 4 of file panic.c.

References halt(), and printk().

Referenced by __assert_failed(), alloc(), alloc_init(), get_boot_data(), and kernel().

```
4                              {
5     printk("KERNEL PANIC: %s\n", message);
6     halt();
7 }
```

Here is the call graph for this function:



4.24 kernel/printk.c File Reference

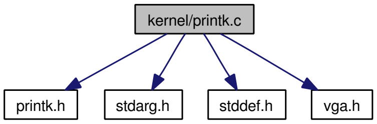
`#include <printk.h>`

`#include <stdarg.h>`

`#include <stddef.h>`

`#include <vga.h>`

Include dependency graph for printk.c:



Functions

- void **printk** (const char ∗format,...)
- void **print_unsigned_int** (unsigned int n)
- void **print_hex_nibble** (unsigned char byte)
- void **print_hex_b** (unsigned char byte)
- void **print_hex_w** (unsigned short word)
- void **print_hex_l** (unsigned long dword)
- void **print_hex_q** (unsigned long long qword)

4.24.1 Function Documentation

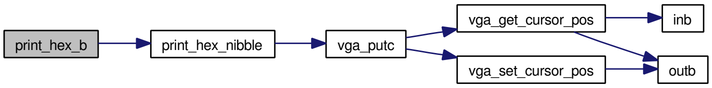
4.24.1.1 void print_hex_b (unsigned char *byte*)

Definition at line 105 of file printk.c.

References print_hex_nibble().

```
105                                   {
106     print_hex_nibble( (char)byte );
107     print_hex_nibble( (char)(byte>>4) );
108 }
```

Here is the call graph for this function:



4.24.1.2 void print_hex_l (unsigned long *dword*)

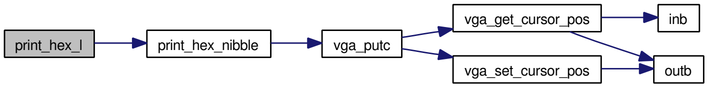
Definition at line 118 of file printk.c.

References print_hex_nibble().

Referenced by printk().

```
118                                     {
119     int off;
120 
121     for(off=32-4; off>=0; off-=4) {
122         print_hex_nibble( (char)(dword>>off) );
123     }
124 }
```

Here is the call graph for this function:



4.24.1.3 void print_hex_nibble (unsigned char *byte*)

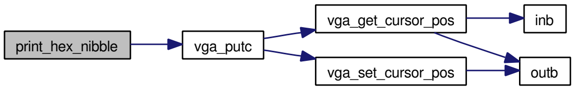
Definition at line 91 of file printk.c.

References vga_putc().

Referenced by print_hex_b(), print_hex_l(), print_hex_q(), and print_hex_w().

```
91                                          {
92      char c;
93 
94      c = byte & 0xf;
95      if(c < 10) {
96          c += '0';
97      }
98      else {
99          c+= ('a' - 10);
100     }
101 
102     vga_putc(c);
103 }
```

Here is the call graph for this function:



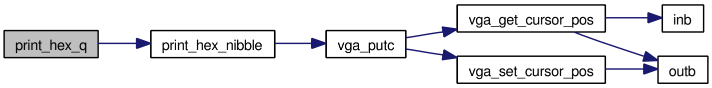
4.24.1.4 void print_hex_q (unsigned long long *qword*)

Definition at line 126 of file printk.c.

References print_hex_nibble().

```
126                                           {
127     int off;
128 
129     for(off=64-4; off>=0; off-=4) {
130         print_hex_nibble( (char)(qword>>off) );
131     }
132 }
```

Here is the call graph for this function:



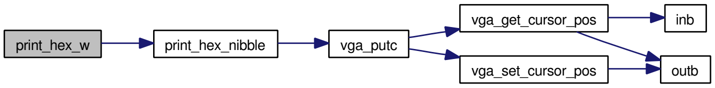
4.24.1.5 void print_hex_w (unsigned short *word*)

Definition at line 110 of file printk.c.

References print_hex_nibble().

```
110                                      {
111     int off;
112 
113     for(off=16-4; off>=0; off-=4) {
114         print_hex_nibble( (char)(word>>off) );
115     }
116 }
```

Here is the call graph for this function:



4.24.1.6 void print_unsigned_int (unsigned int *n*)

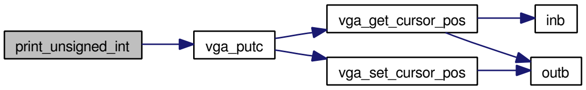
Definition at line 67 of file printk.c.

References vga_putc().

Referenced by printk().

```
67                                        {
68     unsigned int flag = 0;
69     unsigned int pwr;
70     unsigned int digit;
71     char c;
72
73     if(n == 0) {
74         vga_putc('0');
75         return;
76     }
77
78     for(pwr = 1000 * 1000 * 1000; pwr > 0; pwr /= 10) {
79         digit = n / pwr;
80
81         if(digit != 0 || flag) {
82             c = (char)digit + '0';
83             vga_putc(c);
84
85             flag = 1;
86             n -= digit * pwr;
87         }
88     }
89 }
```

Here is the call graph for this function:



4.24.1.7 void printk (const char * *format*, *...*)

Definition at line 6 of file printk.c.

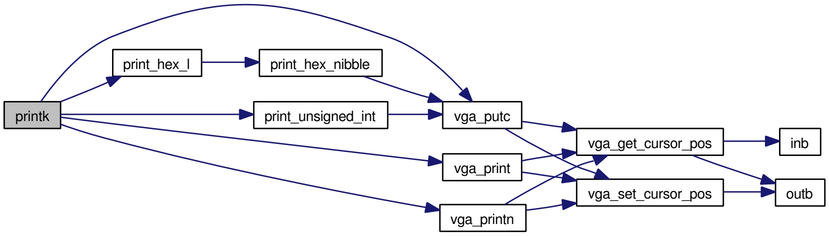
References print_hex_l(), print_unsigned_int(), vm_alloc_t::size, va_arg, va_end, va_start, vga_print(), vga_printn(), and vga_putc().

Referenced by __assert_failed(), alloc_init(), kinit(), and panic().

```
6                                     {
7     va_list ap;
8     const char *idx, *anchor;
```

```
    ptrdiff_t size;

    va_start(ap, format);

    idx = format;

    while(1) {
        anchor = idx;

        while( *idx != 0 && *idx != ’%’ ) {
            ++idx;
        }

        size = idx - anchor;

        if(size > 0) {
            vga_printn(anchor, size);
        }

        if(*idx == 0 || *(idx+1) == 0) {
            break;
        }

        ++idx;

        switch( *idx ) {
        case ’%’:
            vga_putc(’%’);
            break;

        case ’c’:
            /* promotion, promotion */
            vga_putc( (char)va_arg(ap, int) );
            break;

        case ’s’:
            vga_print( va_arg(ap, const char *) );
            break;

        case ’u’:
            print_unsigned_int( va_arg(ap, unsigned int) );
            break;

        case ’x’:
            print_hex_l( va_arg(ap, unsigned long) );
            break;

        default:
            va_end(ap);
            return;
        }

        ++idx;
    }

    va_end(ap);
}
```

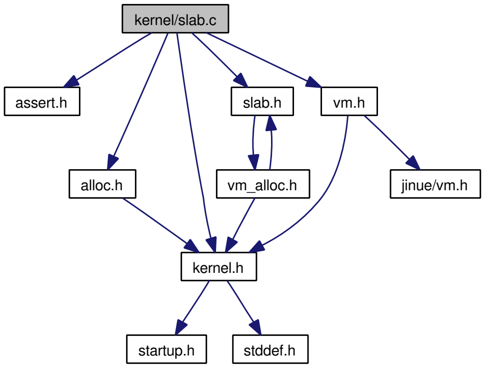
Here is the call graph for this function:



4.25 kernel/slab.c File Reference

```
#include <assert.h>
#include <alloc.h>
#include <kernel.h>
#include <slab.h>
#include <vm.h>
```

Include dependency graph for slab.c:



Functions

- void **slab_create** (**slab_cache_t** *cache, unsigned long flags)
- void **slab_destroy** (**slab_cache_t** *cache)
- **addr_t slab_alloc** (**slab_cache_t** *cache)
- void **slab_free** (**slab_cache_t** *cache, **addr_t** obj)
- void **slab_prepare** (**slab_cache_t** *cache, **addr_t** page)

  Prepare a memory page for use as a slab.

- void **slab_add** (**slab_header_t** **head, **slab_header_t** *slab)

  Add a slab to a linked list of slabs.

- void **slab_remove** (**slab_header_t** **head, **slab_header_t** *slab)

  Remove a slab from a linked list of slab.

4.25.1 Function Documentation

4.25.1.1 void slab_add (slab_header_t ** *head*, slab_header_t * *slab*)

Add a slab to a linked list of slabs.

Parameters:

- ***head*** of list (typically &C->empty, &C->partial or &C->full of some cache C)
- ***slab*** to add to list

Definition at line 122 of file slab.c.

References slab_header_t::next, NULL, and slab_header_t::prev.

Referenced by slab_alloc(), and vm_vfree_block().

```
122                                                    {
123     slab->next = *head;
124     slab->prev = NULL;
125 
126     (*head)->prev = slab;
127     *head = slab;
128 }
```

4.25.1.2 addr_t slab_alloc (slab_cache_t * *cache*)

TODO: handle the NULL pointer

Definition at line 13 of file slab.c.

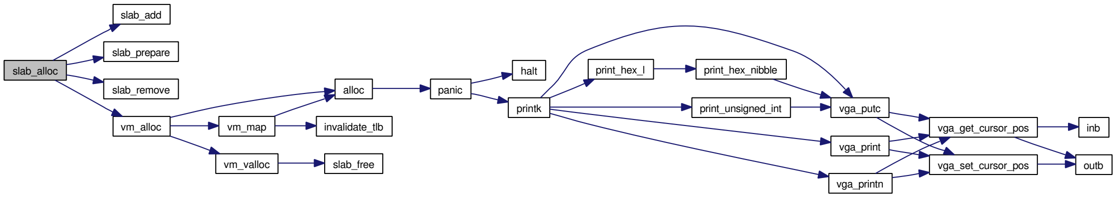
References slab_header_t::available, slab_cache_t::empty, slab_header_t::free_list, slab_cache_t::full, NULL, slab_cache_t::partial, slab_add(), slab_prepare(), slab_remove(), vm_alloc(), slab_cache_t::vm_allocator, and slab_cache_t::vm_flags.

Referenced by vm_vfree_block().

```
13                                      {
14     slab_header_t *slab;
15     addr_t addr;
16 
17     /* use a partial slab if one is available... */
18     slab = cache->partial;
19     if(slab != NULL) {
20         addr = slab->free_list;
21         slab->free_list = *(addr_t *)addr;
22 
```

```
        /* maybe the slab is now full */
        if(--slab->available == 0) {
            slab_remove(&cache->partial, slab);
            slab_add(&cache->full, slab);
        }

        return addr;
    }

    /* ... otherwise, use an empty slab ... */
    slab = cache->empty;
    if(slab != NULL) {
        /* the slab is no longer empty */
        slab_remove(&cache->empty, slab);
        slab_add(&cache->partial, slab);

        addr = slab->free_list;
        slab->free_list = *(addr_t *)addr;

        /* maybe the slab is now full */
        if(--slab->available == 0) {
            slab_remove(&cache->partial, slab);
            slab_add(&cache->full, slab);
        }

        return addr;
    }

    /* ... and, as last resort, allocate a slab */
    slab = (slab_header_t *)vm_alloc(cache->vm_allocator, cache->vm_flags);
    slab_prepare(cache, (addr_t)slab);

    /* this slab is not empty since we are allocating an object from it */
    slab_add(&cache->partial, slab);

    addr = slab->free_list;
    slab->free_list = *(addr_t *)addr;

    /* maybe the slab is now full */
    if(--slab->available == 0) {
        slab_remove(&cache->partial, slab);
        slab_add(&cache->full, slab);
    }

    return addr;
}
```

Here is the call graph for this function:



4.25.1.3 void slab_create (slab_cache_t ∗ *cache*, unsigned long *flags*)

Definition at line 7 of file slab.c.

```
7                                                               {
8 }
```

4.25.1.4 void slab_destroy (slab_cache_t ∗ *cache*)

Definition at line 10 of file slab.c.

```
10                                          {
11 }
```

4.25.1.5 void slab_free (slab_cache_t ∗ *cache*, addr_t *obj*)

Definition at line 71 of file slab.c.

Referenced by vm_valloc().

```
71                                                   {
72 }
```

4.25.1.6 void slab_prepare (slab_cache_t ∗ *cache*, addr_t *page*)

Prepare a memory page for use as a slab.

Initialize fields of the slab header and create the free list.

Parameters:

cache slab cache to which the slab is to be added

page memory page from which to create a slab

ASSERTION: we assume "page" is the starting address of a page

ASSERTION: we assume at least one object can be allocated on slab

ASSERTION: we assume a physical memory page is mapped at "page"

Definition at line 79 of file slab.c.

References assert, slab_header_t::available, slab_header_t::free_list, NULL, slab_cache_t::obj_size, PAGE_MASK, PAGE_OFFSET_OF, PDE_OF, slab_cache_t::per_slab, PTE_OF, and VM_FLAG_PRESENT.

Referenced by slab_alloc(), and vm_vfree_block().

```
79                                                     {
80      unsigned int cx;
81      size_t obj_size;
82      count_t per_slab;
83      slab_header_t *slab;
84      addr_t *ptr;
85      addr_t next;
86
88      assert( PAGE_OFFSET_OF(page) == 0 );
89
91      assert( cache->per_slab > 0 );
92
94      assert( (*PDE_OF(page) & ~PAGE_MASK) != NULL &&  (*PDE_OF(page) & VM_FLAG_PRESENT) != 0 );
95      assert( (*PTE_OF(page) & ~PAGE_MASK) != NULL &&  (*PTE_OF(page) & VM_FLAG_PRESENT) != 0 );
96
97      obj_size = cache->obj_size;
98      per_slab = cache->per_slab;
99
100      /* initialize slab header */
101      slab = (slab_header_t *)page;
102      slab->available = per_slab;
103      slab->free_list = page + sizeof(slab_header_t);
104
105      /* create free list */
106      ptr = (addr_t *)slab->free_list;
107
108      for(cx = 0; cx < per_slab - 1; ++cx) {
109          next = ptr + obj_size;
110          *ptr = next;
111          ptr = (addr_t *)next;
112      }
113
114      *ptr = NULL;
115 }
```

4.25.1.7 void slab_remove (slab_header_t ** *head*, slab_header_t * *slab*)

Remove a slab from a linked list of slab.

Parameters:

- ***head*** of list (typically &C->empty, &C->partial or &C->full of some cache C)
- ***slab*** to remove from list

Definition at line 135 of file slab.c.

References slab_header_t::next, NULL, and slab_header_t::prev.

Referenced by slab_alloc().

```
135                                                          {
136     if(slab->next != NULL) {
137         slab->next->prev = slab->prev;
138     }
139 
140     if(slab->prev != NULL) {
141         slab->prev->next = slab->next;
142     }
143     else {
144         *head = slab->next;
145     }
146 }
```

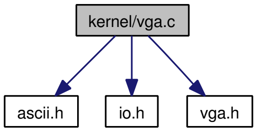
4.26 kernel/vga.c File Reference

`#include <ascii.h>`

`#include <io.h>`

`#include <vga.h>`

Include dependency graph for vga.c:



Functions

- void **vga_init** (void)
- void **vga_clear** (void)
- void **vga_scroll** (void)
- **vga_pos_t vga_get_cursor_pos** (void)
- void **vga_set_cursor_pos** (**vga_pos_t** pos)
- void **vga_print** (const char ∗message)
- void **vga_printn** (const char ∗message, unsigned int n)
- void **vga_putc** (char c)

4.26.1 Function Documentation

4.26.1.1 void vga_clear (void)

Definition at line 25 of file vga.c.

References VGA_COLOR_ERASE, VGA_LINES, VGA_TEXT_VID_BASE, and VGA_WIDTH.

Referenced by vga_init().

```
25                     {
26     unsigned char *buffer = (unsigned char *)VGA_TEXT_VID_BASE;
27     unsigned int idx = 0;
28 
29     while( idx < (VGA_LINES * VGA_WIDTH * 2) )  {
30         buffer[idx++] = 0x20;
31         buffer[idx++] = VGA_COLOR_ERASE;
32     }
33 }
```

4.26.1.2 vga_pos_t vga_get_cursor_pos (void)

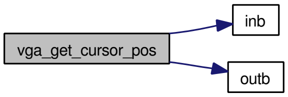
Definition at line 50 of file vga.c.

References inb(), outb(), VGA_CRTC_ADDR, and VGA_CRTC_DATA.

Referenced by vga_print(), vga_printn(), and vga_putc().

```
50                                  {
51     unsigned char h, l;
52 
53     outb(VGA_CRTC_ADDR, 0x0e);
54     h = inb(VGA_CRTC_DATA);
55     outb(VGA_CRTC_ADDR, 0x0f);
56     l = inb(VGA_CRTC_DATA);
57 
58     return (h << 8) | l;
59 }
```

Here is the call graph for this function:



4.26.1.3 void vga_init (void)

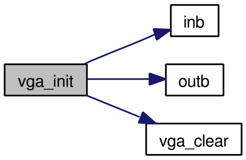
Definition at line 7 of file vga.c.

References inb(), outb(), vga_clear(), VGA_CRTC_ADDR, VGA_CRTC_DATA, VGA_MISC_OUT_RD, and VGA_MISC_OUT_WR.

Referenced by kinit().

```
7                    {
8     unsigned char data;
9 
10     /* Set address select bit in a known state: CRTC regs at 0x3dx */
11     data = inb(VGA_MISC_OUT_RD);
12     data |= 1;
13     outb(VGA_MISC_OUT_WR, data);
14 
15     /* Move cursor to line 0 col 0 */
16     outb(VGA_CRTC_ADDR, 0x0e);
17     outb(VGA_CRTC_DATA, 0x0);
18     outb(VGA_CRTC_ADDR, 0x0f);
19     outb(VGA_CRTC_DATA, 0x0);
20 
21     /* Clear the screen */
22     vga_clear();
23 }
```

Here is the call graph for this function:



4.26.1.4 void vga_print (const char ∗ *message*)

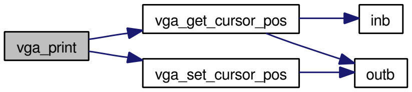
Definition at line 72 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by printk().

```
72                                     {
73     unsigned short int pos = vga_get_cursor_pos();
74     char c;
75 
76     while( (c = *(message++)) ) {
77         pos = vga_raw_putc(c, pos);
78     }
79 
80     vga_set_cursor_pos(pos);
81 }
```

Here is the call graph for this function:



4.26.1.5 void vga_printn (const char ∗ *message*, unsigned int *n*)

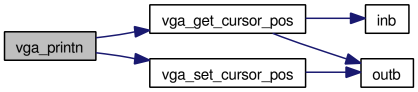
Definition at line 83 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by printk().

```
83                                                      {
84     vga_pos_t pos = vga_get_cursor_pos();
85     char c;
86 
87     while(n) {
88         c = *(message++);
89         pos = vga_raw_putc(c, pos);
90         --n;
91     }
92 
93     vga_set_cursor_pos(pos);
94 }
```

Here is the call graph for this function:



4.26.1.6 void vga_putc (char *c*)

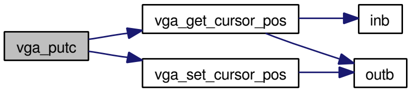
Definition at line 96 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by print_hex_nibble(), print_unsigned_int(), and printk().

```
96                      {
97     vga_pos_t pos = vga_get_cursor_pos();
98 
99     pos = vga_raw_putc(c, pos);
100 
101     vga_set_cursor_pos(pos);
102 }
```

Here is the call graph for this function:



4.26.1.7 void vga_scroll (void)

Definition at line 35 of file vga.c.

References VGA_COLOR_ERASE, VGA_LINES, VGA_TEXT_VID_BASE, and VGA_WIDTH.

```
35                      {
36     unsigned char *di = (unsigned char *)VGA_TEXT_VID_BASE;
37     unsigned char *si = (unsigned char *)(VGA_TEXT_VID_BASE + 2 * VGA_WIDTH);
38     unsigned int idx;
39 
40     for(idx = 0; idx < 2 * VGA_WIDTH * (VGA_LINES - 1); ++idx) {
41         *(di++) = *(si++);
42     }
43 
44     for(idx = 0; idx < VGA_WIDTH; ++idx) {
45         *(di++) = 0x20;
46         *(di++) = VGA_COLOR_ERASE;
47     }
48 }
```

4.26.1.8 void vga_set_cursor_pos (vga_pos_t *pos*)

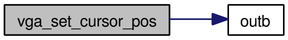
Definition at line 61 of file vga.c.

References outb(), VGA_CRTC_ADDR, and VGA_CRTC_DATA.

Referenced by vga_print(), vga_printn(), and vga_putc().

```
61                                      {
62     unsigned char h = pos >> 8;
63     unsigned char l = pos;
64 
65     outb(VGA_CRTC_ADDR, 0x0e);
66     outb(VGA_CRTC_DATA, h);
67     outb(VGA_CRTC_ADDR, 0x0f);
68     outb(VGA_CRTC_DATA, l);
69 }
```

Here is the call graph for this function:



4.27 kernel/vm.c File Reference

`#include <kernel.h>`

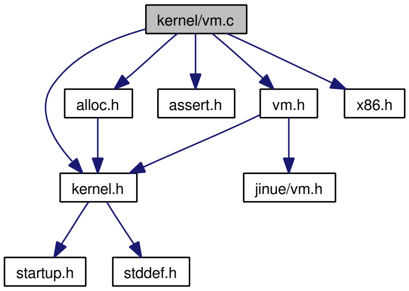
`#include <alloc.h>`

`#include <assert.h>`

`#include <vm.h>`

`#include <x86.h>`

Include dependency graph for vm.c:



Functions

- void **vm_map** (**addr_t** vaddr, **addr_t** paddr, unsigned long flags)

  Map a page frame (physical page) to a virtual memory page.

- void **vm_unmap** (**addr_t** addr)

  Unmap a page from virtual memory.

4.27.1 Function Documentation

4.27.1.1 void vm_map (addr_t *vaddr*, addr_t *paddr*, unsigned long *flags*)

Map a page frame (physical page) to a virtual memory page.

Parameters:

vaddr virtual address of mapping
paddr address of page frame
flags flags used for mapping (see VM_FLAG_x constants in vm.h)

ASSERTION: we assume vaddr is aligned on a page boundary

ASSERTION: we assume paddr is aligned on a page boundary

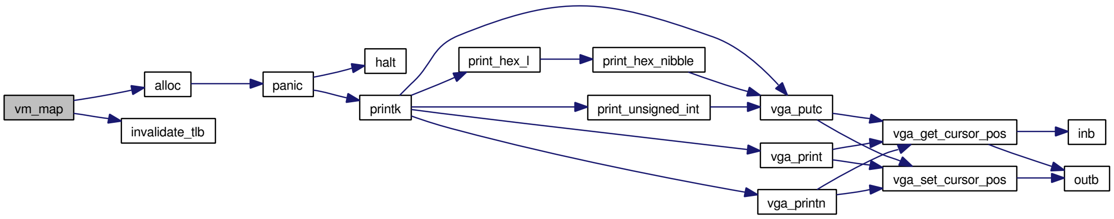
Definition at line 13 of file vm.c.

References alloc(), assert, invalidate_tlb(), PAGE_OFFSET_OF, PAGE_SIZE, PAGE_TABLE_ENTRIES, PAGE_TABLE_OF, PAGE_TABLE_PTE_OF, PDE_OF, PTE_OF, VM_FLAG_PRESENT, VM_FLAG_USER, and VM_FLAGS_PAGE_TABLE.

Referenced by vm_alloc(), and vm_vfree_block().

```
13                                                                  {
14     pte_t *pte, *pde;
15     addr_t page_table;
16     int idx;
17
19     assert( PAGE_OFFSET_OF(vaddr) == 0 );
20
22     assert( PAGE_OFFSET_OF(paddr) == 0 );
23
24     /* get page directory entry */
25     pde = PDE_OF(vaddr);
26
27     /* check if page table must be created */
28     if( !(*pde & VM_FLAG_PRESENT) ) {
29         /* allocate a new page table */
30         page_table = alloc(PAGE_SIZE);
31
32         /* map page table in the region of memory reserved for that purpose */
33         pte = PAGE_TABLE_PTE_OF(vaddr);
34         *pte = (pte_t)page_table | VM_FLAGS_PAGE_TABLE | VM_FLAG_PRESENT;
35
36         /* obtain virtual address of new page table */
37         pte = PAGE_TABLE_OF(vaddr);
38
39         /* invalidate TLB entry for new page table */
40         invalidate_tlb( (addr_t)pte );
41
42         /* zero content of page table */
43         for(idx = 0; idx < PAGE_TABLE_ENTRIES; ++idx) {
44             pte[idx] = 0;
45         }
46
47         /* link to page table from page directory */
48         *pde = (pte_t)page_table | VM_FLAG_USER | VM_FLAG_PRESENT;
49     }
50
51     /* perform the actual mapping */
52     pte = PTE_OF(vaddr);
53     *pte = (pte_t)paddr | flags | VM_FLAG_PRESENT;
54
55     /* invalidate TLB entry for newly mapped page */
56     invalidate_tlb(vaddr);
57 }
```

Here is the call graph for this function:



4.27.1.2 void vm_unmap (addr_t *addr*)

Unmap a page from virtual memory.

Parameters:

addr address of page to unmap

ASSERTION: we assume addr is aligned on a page boundary

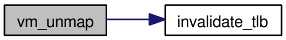
Definition at line 63 of file vm.c.

References assert, invalidate_tlb(), NULL, PAGE_OFFSET_OF, and PTE_OF.

Referenced by vm_free().

```
63                          {
64     pte_t *pte;
65 
67     assert( PAGE_OFFSET_OF(addr) == 0 );
68 
69     pte = PTE_OF(addr);
70     *pte = NULL;
71 
72     invalidate_tlb(addr);
73 }
```

Here is the call graph for this function:



4.28 kernel/vm_alloc.c File Reference

`#include <alloc.h>`

`#include <assert.h>`

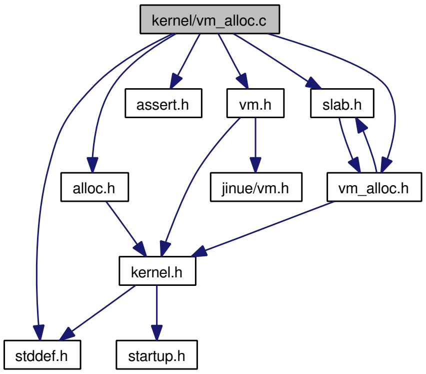
`#include <slab.h>`

`#include <stddef.h>`

`#include <vm.h>`

`#include <vm_alloc.h>`

Include dependency graph for vm_alloc.c:



Functions

- **addr_t vm_valloc** (**vm_alloc_t** *pool)

  Allocate a page of virtual memory (not backed by physical memory).

- void **vm_vfree** (**vm_alloc_t** *pool, **addr_t** addr)

  Return a single page of virtual memory to a pool of available pages.

- void **vm_vfree_block** (**vm_alloc_t** *pool, **addr_t** addr, **size_t** size)

  Return a block of contiguous virtual memory pages to a pool of available pages.

- **addr_t vm_alloc** (**vm_alloc_t** *pool, unsigned long flags)

  Allocate a physical memory page and map it in virtual memory.

- void **vm_free** (**vm_alloc_t** *pool, **addr_t** addr)

*Free a physical page mapped in virtual memory (which was typically obtained through a call to **vm_map()** (p. 41)).*

4.28.1 Function Documentation

4.28.1.1 addr_t vm_alloc (vm_alloc_t * *pool*, unsigned long *flags*)

Allocate a physical memory page and map it in virtual memory.

Parameters:

pool data structure managing the virtual memory region in which page will be mapped

flags flags for page mapping (passed as-is to **vm_map()** (p. 41))

TODO: handle the NULL pointer

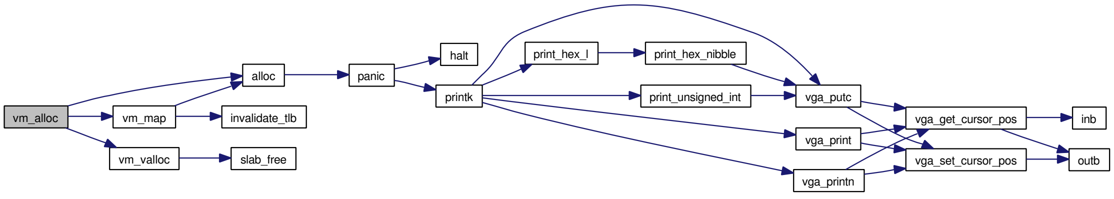
Definition at line 135 of file vm_alloc.c.

References alloc(), PAGE_SIZE, vm_map(), and vm_valloc().

Referenced by slab_alloc().

```
135                                                         {
136     addr_t paddr, vaddr;
137 
140     vaddr = vm_valloc(pool);
141     paddr = alloc(PAGE_SIZE);
142     vm_map(vaddr, paddr, flags);
143 
144     return vaddr;
145 }
```

Here is the call graph for this function:



4.28.1.2 void vm_free (vm_alloc_t * *pool*, addr_t *addr*)

Free a physical page mapped in virtual memory (which was typically obtained through a call to **vm_map()** (p. 41)).

The physical memory is freed and the virtual page is returned to the virtual address space allocator.

Parameters:

pool data structure managing the virtual memory region to which the page is returned address of page to free

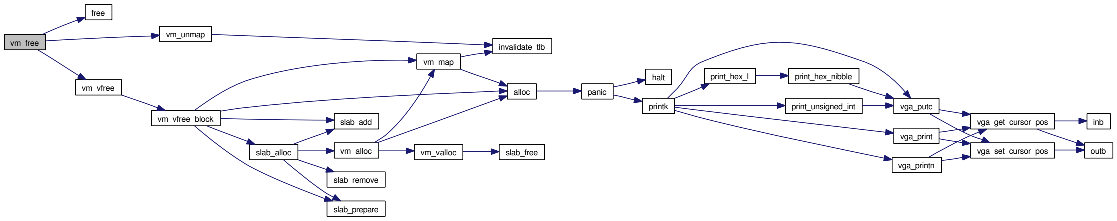
ASSERTION: address of page should not be the null pointer

Definition at line 154 of file vm_alloc.c.

References assert, free(), NULL, PAGE_MASK, PTE_OF, vm_unmap(), and vm_vfree().

```
154                                             {
155     addr_t paddr;
156
158     assert( addr != (addr_t)NULL );
159
160     paddr = (addr_t)(*PTE_OF(addr) | ~PAGE_MASK);
161
162     vm_unmap(addr);
163     vm_vfree(pool, addr);
164     free(paddr);
165 }
```

Here is the call graph for this function:



4.28.1.3 addr_t vm_valloc (vm_alloc_t * *pool*)

Allocate a page of virtual memory (not backed by physical memory).

This page may then be used for temporary mappings, for example. Page is allocated from a specific virtual memory region managed by a **vm_alloc_t** (p. 13) data structure.

Parameters:

pool data structure managing the virtual memory region from which to allocate

Returns:

address of allocated page

ASSERTION: block size should be an integer number of pages

ASSERTION: returned address should be aligned with a page boundary

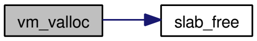
Definition at line 17 of file vm_alloc.c.

References vm_link_t::addr, assert, vm_alloc_t::cache, vm_alloc_t::head, vm_link_t::next, NULL, PAGE_OFFSET_OF, PAGE_SIZE, vm_link_t::size, vm_alloc_t::size, and slab_free().

Referenced by vm_alloc().

```
17                                    {
18     addr_t addr;
19     vm_link_t *head;
20     size_t size;
21 
22     head = pool->head;
23 
24     /* no page available */
25     if(head == (addr_t)NULL) {
26         return (addr_t)NULL;
27     }
28 
29     addr = head->addr;
30     size = head->size - PAGE_SIZE;
31 
33     assert( PAGE_OFFSET_OF(size) == 0 );
34 
35     /* if block is made of only one page, we remove it from the free list */
36     if(size == 0) {
37         pool->head = head->next;
38         slab_free(pool->cache, head);
39     }
40     else {
41         head->size = size;
42         head->addr += PAGE_SIZE;
43     }
44 
46     assert( PAGE_OFFSET_OF(addr) == 0 );
47 
48     return addr;
49 }
```

Here is the call graph for this function:



4.28.1.4 void vm_vfree (vm_alloc_t * *pool*, addr_t *addr*)

Return a single page of virtual memory to a pool of available pages.

Should not be used to free pages to which physical memory is still mapped (no physical memory is freed by this function). Use this function to return pages obtained by a call to **vm_valloc()** (p. 79) (and not **vm_alloc()** (p. 77)).

Parameters:

pool data structure managing the relevant virtual memory region

addr address of virtual page which must be freed

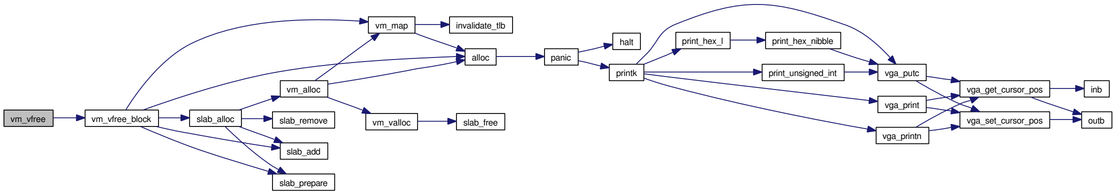
Definition at line 59 of file vm_alloc.c.

References PAGE_SIZE, and vm_vfree_block().

Referenced by vm_free().

```
59                                          {
60     vm_vfree_block(pool, addr, PAGE_SIZE);
61 }
```

Here is the call graph for this function:



4.28.1.5 void vm_vfree_block (vm_alloc_t * *pool*, addr_t *addr*, size_t *size*)

Return a block of contiguous virtual memory pages to a pool of available pages.

Should not be used to free pages to which physical memory is still mapped (no physical memory is freed by this function).

Parameters:

- ***pool*** data structure managing the relevant virtual memory region
- ***addr*** starting address of virtual memory block
- ***size*** size of block

ASSERTION: we assume starting address is aligned on a page boundary

ASSERTION: we assume size of block is an integer number of pages

ASSERTION: address of block should not be the null pointer

Definition at line 71 of file vm_alloc.c.

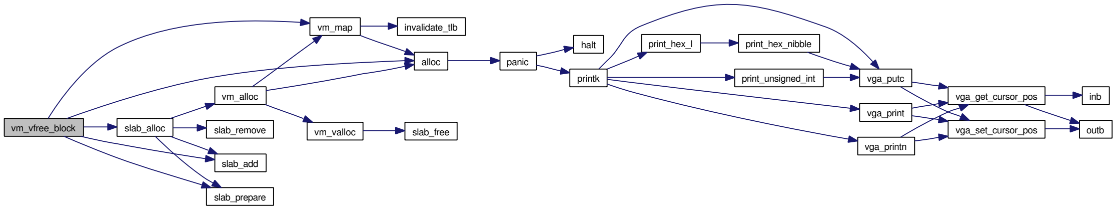
References vm_link_t::addr, alloc(), assert, vm_alloc_t::cache, slab_cache_t::empty, vm_alloc_t::head, vm_link_t::next, NULL, PAGE_OFFSET_OF, PAGE_SIZE, slab_cache_t::partial, vm_link_t::size, slab_add(), slab_alloc(), slab_prepare(), slab_cache_t::vm_allocator, VM_FLAG_KERNEL, and vm_map().

Referenced by vm_vfree().

```
                                                            {
    addr_t phys_page;
    vm_link_t *link;

    assert( PAGE_OFFSET_OF(addr) == 0 );

    assert( PAGE_OFFSET_OF(size) == 0 );

    assert( addr != (addr_t)NULL );

    /*  The virtual address space allocator needs a slab cache from which to
        allocate data structures for its free list. Also, each slab cache needs
        a virtual address space allocator to allocate slabs when needed.

        There can be a mutual dependency between the virtual address space
        allocator and the slab cache. This is not a problem in general, but a
        special bootstrapping procedure is needed for initialization of the
        virtual address space allocator in that case. The virtual address space
        allocator will actually "donate" a virtual page (backed by physical ram)
        to the cache for use as a slab.

        This case is handled here
     */
    if(pool->head == NULL) {
        if(pool->cache->vm_allocator == pool) {
            if(pool->cache->empty == NULL && pool->cache->partial == NULL) {
                 /* allocate a physical page for slab */
                 phys_page = alloc(PAGE_SIZE);

                 /* map page */
                 vm_map(addr, phys_page, VM_FLAG_KERNEL);

                 /* prepare the slab and add it to cache empty list */
```

```
                slab_prepare(pool->cache, addr);
                slab_add(&pool->cache->empty, addr);

                size -= PAGE_SIZE;

                /* if the block contained only one page, we have nothing left
                   to free */
                if(size == 0) {
                    return;
                }

                addr += PAGE_SIZE;
            }
        }
    }

    link = (vm_link_t *)slab_alloc(pool->cache);
    link->size = size;
    link->addr = addr;

    link->next = pool->head;
    pool->head = link;
}
```

Here is the call graph for this function:



Index