

Jinue

Generated by Doxygen 1.5.5

Sat Jul 18 14:18:05 2009

Contents

1	Data Structure Index	1
1.1	Data Structures	1
2	File Index	3
2.1	File List	3
3	Data Structure Documentation	5
3.1	boot_t Struct Reference	5
3.2	e820_t Struct Reference	7
3.3	slab_cache_t Struct Reference	8
3.4	slab_header_t Struct Reference	11
3.5	vm_alloc_t Struct Reference	13
3.6	vm_link_t Struct Reference	15
4	File Documentation	17
4.1	include/alloc.h File Reference	17
4.2	include/ascii.h File Reference	22
4.3	include/assert.h File Reference	23
4.4	include/boot.h File Reference	25
4.5	include/io.h File Reference	30
4.6	include/jinue/vm.h File Reference	31
4.7	include/vm.h File Reference	34
4.8	include/kernel.h File Reference	44
4.9	include/panic.h File Reference	48

4.10	include/printk.h File Reference	49
4.11	include/slab.h File Reference	55
4.12	include/startup.h File Reference	61
4.13	include/stdarg.h File Reference	62
4.14	include/stdbool.h File Reference	64
4.15	include/stddef.h File Reference	65
4.16	include/vga.h File Reference	67
4.17	include/vm_alloc.h File Reference	76
4.18	include/x86.h File Reference	84
4.19	kernel/alloc.c File Reference	85
4.20	kernel/assert.c File Reference	90
4.21	kernel/boot.c File Reference	92
4.22	kernel/kernel.c File Reference	96
4.23	kernel/panic.c File Reference	100
4.24	kernel/printk.c File Reference	101
4.25	kernel/slab.c File Reference	107
4.26	kernel/vga.c File Reference	113
4.27	kernel/vm.c File Reference	118
4.28	kernel/vm_alloc.c File Reference	121

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

boot_t	5
e820_t	7
slab_cache_t (Data structure describing a cache)	8
slab_header_t (Header of a slab)	11
vm_alloc_t (Data structure which keep tracks of free pages in a region of virtual memory)	13
vm_link_t (Links forming the linked lists of free virtual memory pages)	15

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

include/ alloc.h	17
include/ ascii.h	22
include/ assert.h	23
include/ boot.h	25
include/ io.h	30
include/ kernel.h	44
include/ panic.h	48
include/ printk.h	49
include/ slab.h	55
include/ startup.h	61
include/ stdarg.h	62
include/ stdbool.h	64
include/ stddef.h	65
include/ vga.h	67
include/ vm.h	34
include/ vm_alloc.h	76
include/ x86.h	84
include/jinve/ vm.h	31
kernel/ alloc.c	85
kernel/ assert.c	90
kernel/ boot.c	92
kernel/ kernel.c	96
kernel/ panic.c	100
kernel/ printk.c	101
kernel/ slab.c	107

kernel/ vga.c	113
kernel/ vm.c	118
kernel/ vm_alloc.c	121

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().

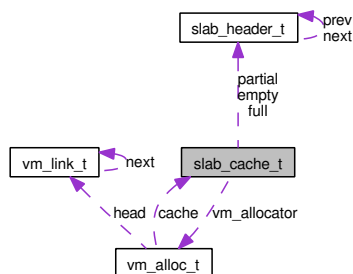
The documentation for this struct was generated from the following file:

- 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().

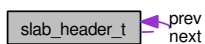
The documentation for this struct was generated from the following file:

- 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().

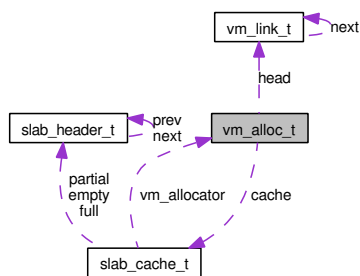
The documentation for this struct was generated from the following file:

- 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()`.

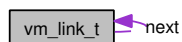
The documentation for this struct was generated from the following file:

- `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`

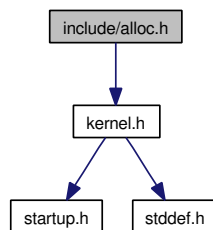
Chapter 4

File Documentation

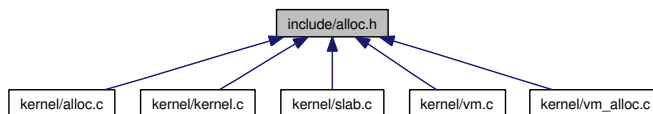
4.1 include/alloc.h File Reference

```
#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)

4.1.1.1 `addr_t alloc (size_t size)`

Definition at line 96 of file alloc.c.

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
114     assert( ((unsigned long)addr & PAGE_MASK) == 0 );
115
116     return addr;
117 }

```

```

graph LR
    alloc[alloc] --> panic[panic]
    panic --> halt[halt]
    panic --> printk[printk]
    printk --> print_hex_l[print_hex_l]
    printk --> print_hex_nibble[print_hex_nibble]
    printk --> print_unsigned_int[print_unsigned_int]
    printk --> vga_putc[vga_putc]
    printk --> vga_print[vga_print]
    printk --> vga_printf[vga_printf]
    print_hex_l --> print_hex_nibble
    print_hex_nibble --> vga_putc
    print_unsigned_int --> vga_putc
    vga_putc --> vga_get_cursor_pos[vga_get_cursor_pos]
    vga_putc --> vga_print
    vga_putc --> vga_printf
    vga_get_cursor_pos --> inb[inb]
    vga_print --> vga_set_cursor_pos[vga_set_cursor_pos]
    vga_printf --> vga_set_cursor_pos
    vga_set_cursor_pos --> outb[outb]
  
```

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;
19     size_t fixed_size, best_size;
20
21     idx = 0;
22     best_size = 0;
23
24     printk("Dump of the BIOS memory map:\n");
25     printk(" address  size    type\n");
26     while( e820_is_valid(idx) ) {
27         addr = e820_get_addr(idx);
28         size = e820_get_size(idx);
29         type = e820_get_type(idx);
30         avail = e820_is_available(idx);
31
32         ++idx;
33
34         printk("%c %x %x %s\n",
35             avail?'*':' ',
36             addr,
37             size,
38             e820_type_description(type) );
39
40         if( !avail ) {
41             continue;
42         }
43
44         fixed_addr = addr;
45         fixed_size = size;
46
47         /* is the region completely under the kernel ? */
48         if(addr + size > kernel_start) {
49             /* is the region completely above the kernel ? */
50             if(addr < kernel_top) {
51                 /* if the region touches the kernel, we take only
52                  * the part above the kernel, if there is one... */
53                 if(addr + size <= kernel_top) {
54                     /* ... and apparently, there is none */
55                     continue;
56                 }
57
58                 fixed_addr = kernel_top;
59                 fixed_size -= fixed_addr - addr;
60             }
61         }
62     }

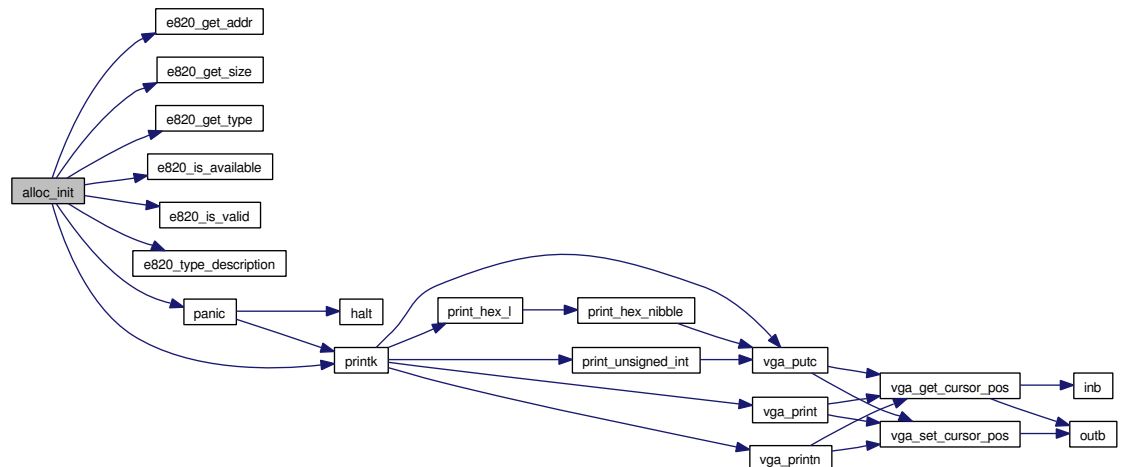
```

```

63      /* we must make sure the starting address is aligned on a
64      * page boundary. The size will eventually be divided
65      * by the page size, and thus need not be aligned. */
66      remainder = (unsigned int)fixed_addr % PAGE_SIZE;
67      if(remainder != 0) {
68          remainder = PAGE_SIZE - remainder;
69          if(fixed_size < remainder) {
70              continue;
71          }
72
73          fixed_addr += remainder;
74          fixed_size -= remainder;
75      }
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:



4.1.1.3 void free (addr_t *addr*)

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

Definition at line 120 of file alloc.c.

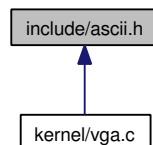
References `assert`, and `PAGE_OFFSET_OF`.

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`

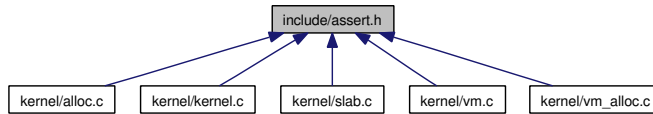
Definition at line 5 of file `ascii.h`.

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

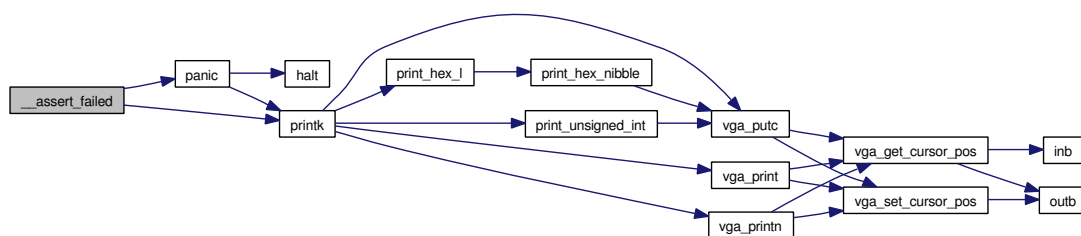
4.3.2.1 `void __assert_failed (const char * expr, const char * file, unsigned int line, const char * func)`

Definition at line 5 of file `assert.c`.

References `panic()`, and `printk()`.

```
9         {
10
11     printk(
12         "ASSERTION FAILED [%s]: %s at line %u in function %s.\n",
13         expr, file, line, func );
14
15     panic("Assertion failed.");
16 }
```

Here is the call graph for this function:

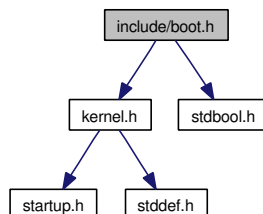


4.4 include/boot.h File Reference

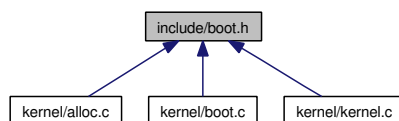
```
#include <kernel.h>
```

```
#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`.

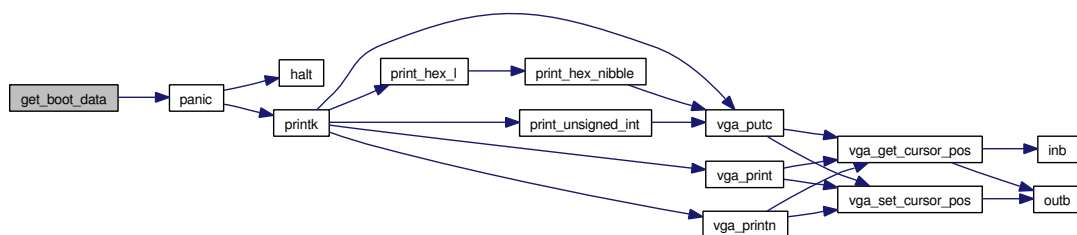
Referenced by `kinit()`.

```

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 }

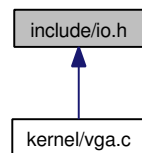
```

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*)

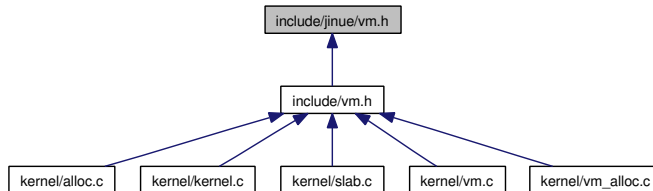
Referenced by vga_get_cursor_pos(), vga_init(), and vga_set_cursor_pos().

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

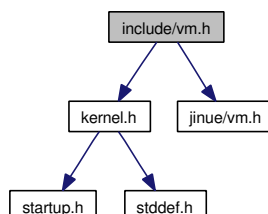
Definition at line 20 of file vm.h.

4.7 include/vm.h File Reference

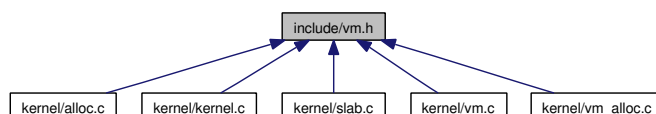
```
#include <kernel.h>
```

```
#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.

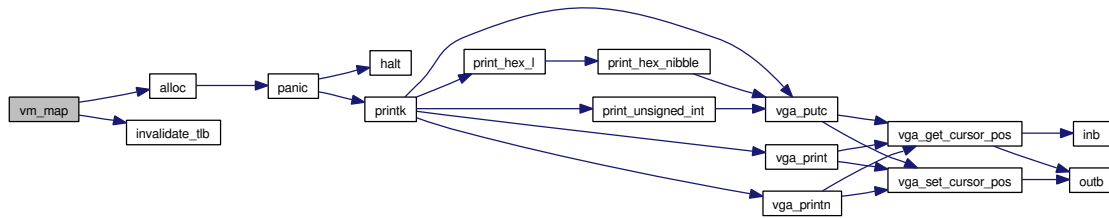
References `alloc()`, `assert`, `invalidate_tlb()`, `PAGE_OFFSET_OF`, `PAGE_SIZE`, `PAGE_TABLE_ENTRIES`, `PAGE_TABLE_OF`, `PAGE_TABLE_PTE_OF`, `PDE_OF`, `PTE_OF`, `VM_FLAG_PRESENT`, `VM_FLAG_USER`, and `VM_FLAGS_PAGE_TABLE`.

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

```

13                                     {
14     pte_t *pte, *pde;
15     addr_t page_table;
16     int idx;
17
18
19     assert( PAGE_OFFSET_OF(vaddr) == 0 );
20
21
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
66      assert( PAGE_OFFSET_OF(addr) == 0 );
67
68      pte = PTE_OF(addr);
69      *pte = NULL;
70
71      invalidate_tlb(addr);
72  }
73  }
```

Here is the call graph for this function:

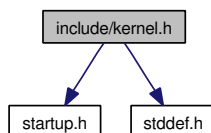


4.8 include/kernel.h File Reference

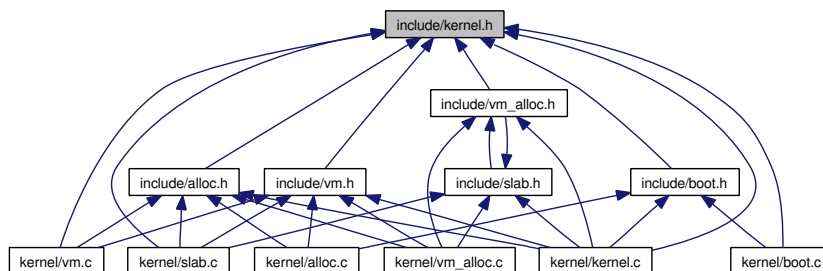
```
#include <startup.h>
```

```
#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()`.

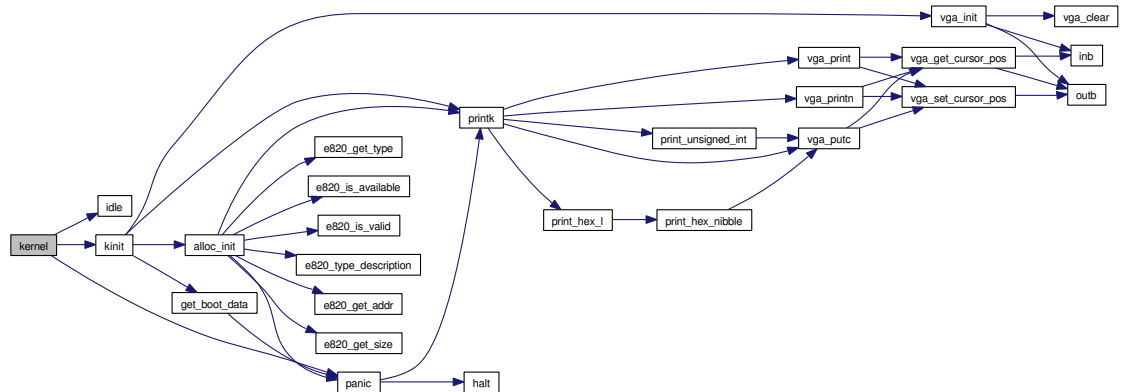
```
16     {  
17     kinit();
```

```

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
31         assert((unsigned int)kernel_start % PAGE_SIZE == 0);
32
33         /* find out kernel size and set kernel_top
34          * (top of kernel, aligned to page boundary) */
35         boot = get_boot_data();
36
37         kernel_size = boot->sysize * 16;
38         remainder = kernel_size % PAGE_SIZE;
39
40

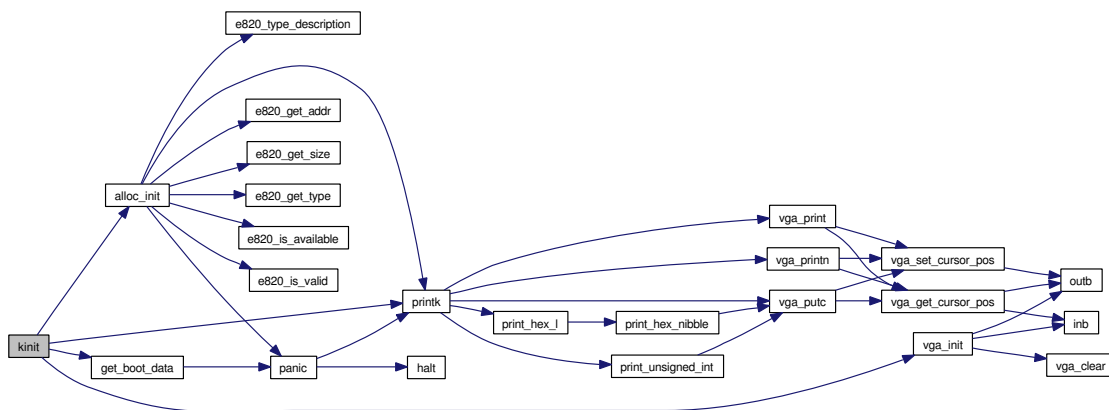
```

```

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:



4.8.4 Variable Documentation

4.8.4.1 size_t kernel_size

Definition at line 14 of file kernel.c.

Referenced by kinit().

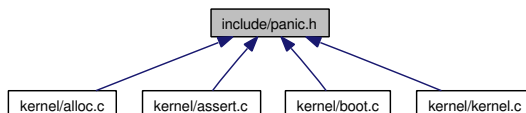
4.8.4.2 addr_t kernel_top

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.

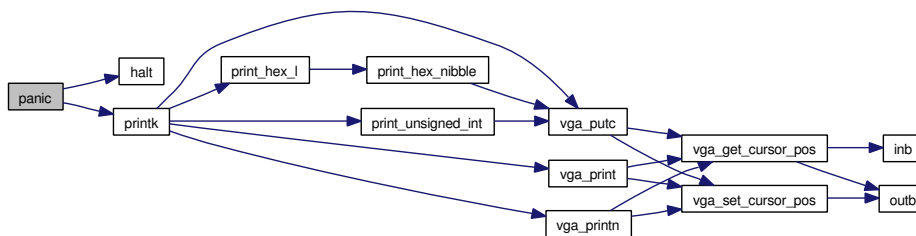
References `halt()`, and `printk()`.

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

```

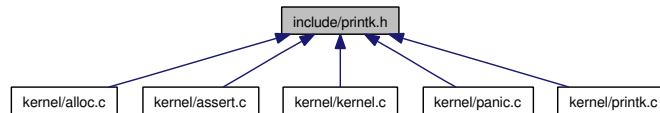
4      {
5      printk("KERNEL PANIC: %s\n", message);
6      halt();
7  }
```

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`.

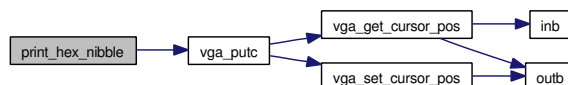
References `vga_putc()`.

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()`.

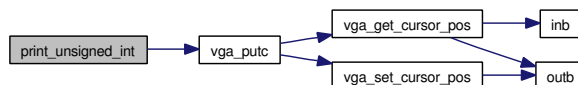
Referenced by `printk()`.

```

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()`.

```

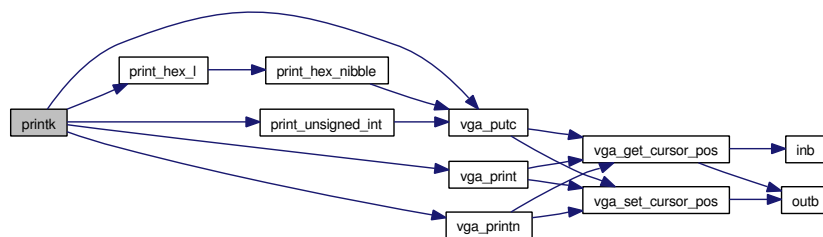
6                                     {
7     va_list ap;
8     const char *idx, *anchor;
9     ptrdiff_t size;
10
11     va_start(ap, format);
12

```



```
13     idx = format;
14
15     while(1) {
16         anchor = idx;
17
18         while( *idx != 0 && *idx != '%' ) {
19             ++idx;
20         }
21
22         size = idx - anchor;
23
24         if(size > 0) {
25             vga_printn(anchor, size);
26         }
27
28         if(*idx == 0 || *(idx+1) == 0) {
29             break;
30         }
31
32         ++idx;
33
34         switch( *idx ) {
35             case '%':
36                 vga_putc('%');
37                 break;
38
39             case 'c':
40                 /* promotion, promotion */
41                 vga_putc( (char)va_arg(ap, int) );
42                 break;
43
44             case 's':
45                 vga_print( va_arg(ap, const char *) );
46                 break;
47
48             case 'u':
49                 print_unsigned_int( va_arg(ap, unsigned int) );
50                 break;
51
52             case 'x':
53                 print_hex_l( va_arg(ap, unsigned long) );
54                 break;
55
56             default:
57                 va_end(ap);
58                 return;
59         }
60
61         ++idx;
62     }
63
64     va_end(ap);
65 }
```

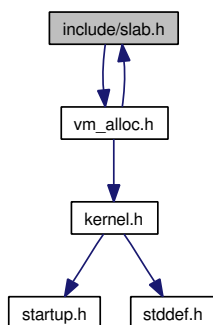
Here is the call graph for this function:



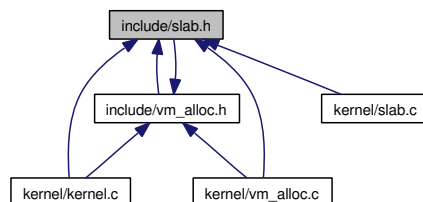
4.11 include/slab.h File Reference

```
#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);

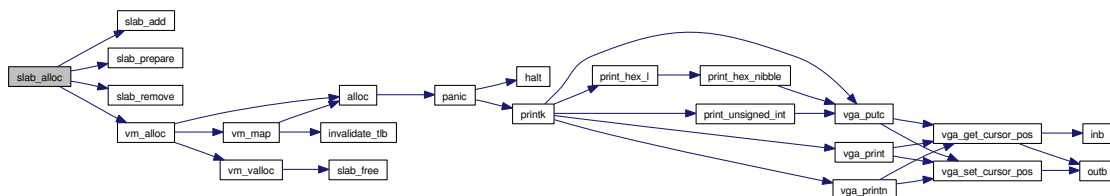
```

```

46     }
47
48     return addr;
49 }
50
51 /* ... and, as last resort, allocate a slab */
53 slab = (slab_header_t *)vm_alloc(cache->vm_allocator, cache->vm_flags);
54 slab_prepare(cache, (addr_t)slab);
55
56 /* this slab is not empty since we are allocating an object from it */
57 slab_add(&cache->partial, slab);
58
59 addr = slab->free_list;
60 slab->free_list = *(addr_t *)addr;
61
62 /* maybe the slab is now full */
63 if(--slab->available == 0) {
64     slab_remove(&cache->partial, slab);
65     slab_add(&cache->full, slab);
66 }
67
68 return addr;
69 }

```

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
87
88     assert( PAGE_OFFSET_OF(page) == 0 );
89
90
91     assert( cache->per_slab > 0 );
92
93
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().

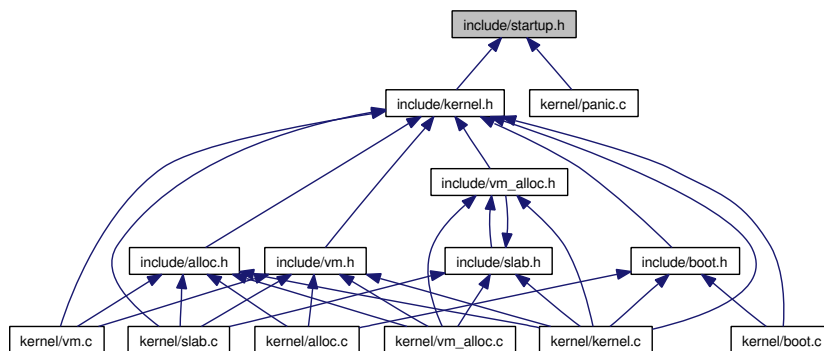
```

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

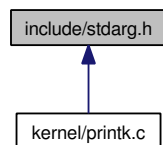
4.12.1.1 void halt (void)

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 printf().

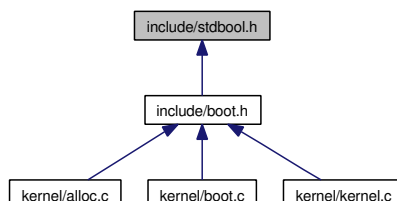
4.13.2 Typedef Documentation

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`

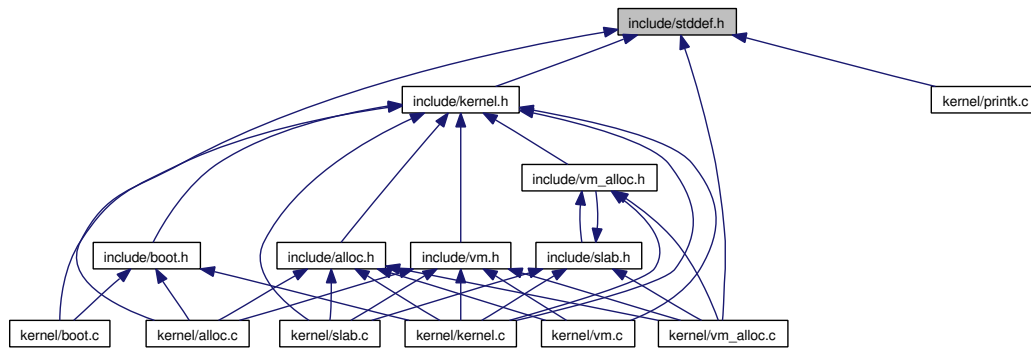
Definition at line 6 of file stdbool.h.

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

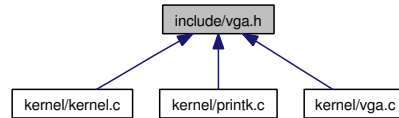
Definition at line 5 of file stddef.h.

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()`.

```

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 }
```

4.16.3.2 `vga_pos_t vga_get_cursor_pos (void)`

Definition at line 50 of file `vga.c`.

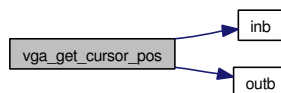
References `inb()`, `outb()`, `VGA_CRTC_ADDR`, and `VGA_CRTC_DATA`.

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

```

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:



4.16.3.3 `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()`.

```

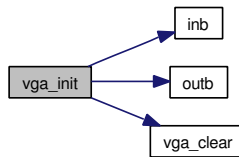
7     {
```

```

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:



4.16.3.4 void vga_print (const char * *message*)

Definition at line 72 of file `vga.c`.

References `vga_get_cursor_pos()`, and `vga_set_cursor_pos()`.

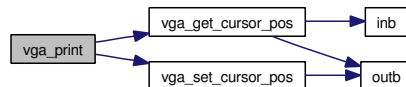
Referenced by `printk()`.

```

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.

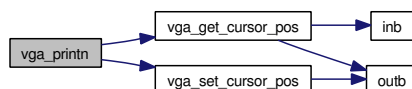
References vga_get_cursor_pos(), and vga_set_cursor_pos().

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.

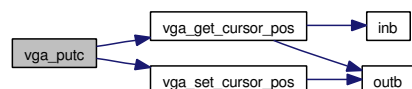
References vga_get_cursor_pos(), and vga_set_cursor_pos().

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)

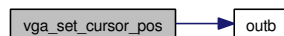
Definition at line 61 of file vga.c.

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:

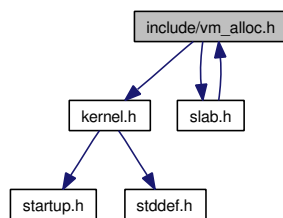


4.17 include/vm_alloc.h File Reference

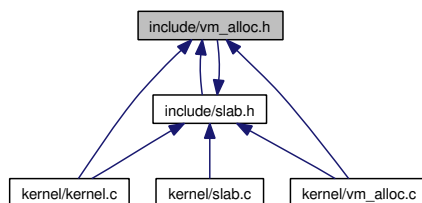
```
#include <kernel.h>
```

```
#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`.

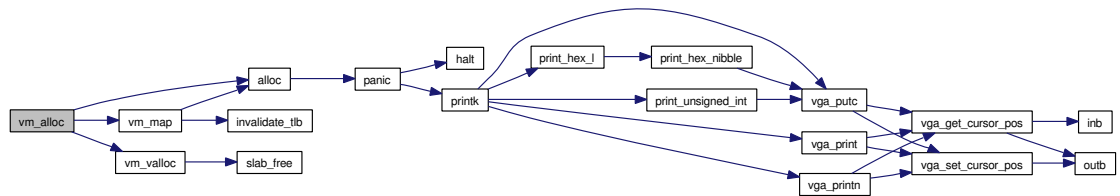
References `alloc()`, `PAGE_SIZE`, `vm_map()`, and `vm_valloc()`.

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()`.

```

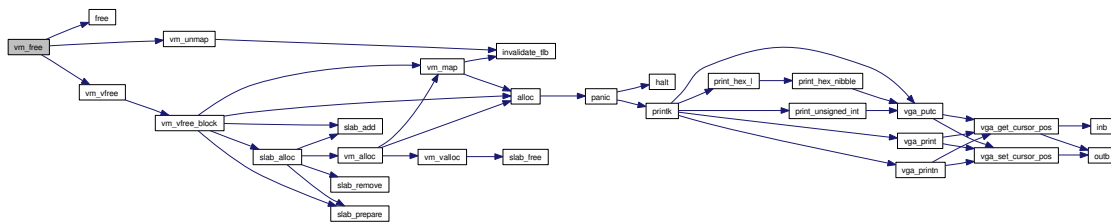
154                                     {
```

```

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

```

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()`.

```

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
32     assert( PAGE_OFFSET_OF(size) == 0 );
33
34     /* if block is made of only one page, we remove it from the free list */
35     if(size == 0) {
36         pool->head = head->next;
37         slab_free(pool->cache, head);
38     }
39     else {
40         head->size = size;
41         head->addr += PAGE_SIZE;
42     }
43
44     assert( PAGE_OFFSET_OF(addr) == 0 );
45
46     return addr;
47 }

```

Here is the call graph for this function:



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.

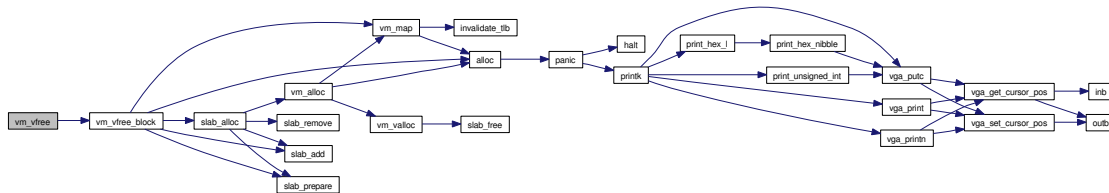
References PAGE_SIZE, and vm_vfree_block().

Referenced by vm_free().

```

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

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().

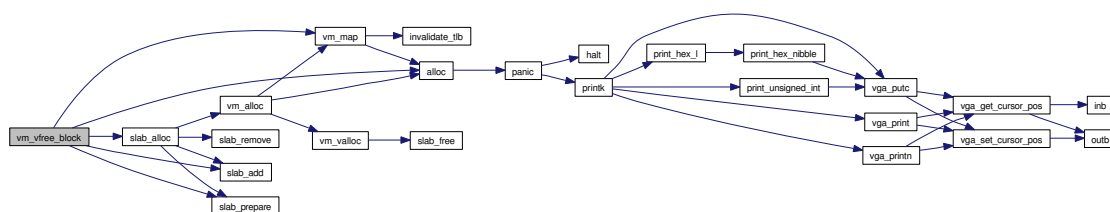
Referenced by vm_vfree().

```

71                                     {
72     addr_t phys_page;
73     vm_link_t *link;
74
75     assert( PAGE_OFFSET_OF(addr) == 0 );
76
77     assert( PAGE_OFFSET_OF(size) == 0 );
78
79     assert( addr != (addr_t)NULL );
80
81     /* The virtual address space allocator needs a slab cache from which to
82        allocate data structures for its free list. Also, each slab cache needs
83        a virtual address space allocator to allocate slabs when needed.
84
85        There can be a mutual dependency between the virtual address space
86        allocator and the slab cache. This is not a problem in general, but a
87        special bootstrapping procedure is needed for initialization of the
88        virtual address space allocator in that case. The virtual address space
89        allocator will actually "donate" a virtual page (backed by physical ram)
90        to the cache for use as a slab.
91
92        This case is handled here
93     */
94     if(pool->head == NULL) {
95         if(pool->cache->vm_allocator == pool) {
96             if(pool->cache->empty == NULL && pool->cache->partial == NULL) {
97                 /* allocate a physical page for slab */
98                 phys_page = alloc(PAGE_SIZE);
99
100                 /* map page */
101                 vm_map(addr, phys_page, VM_FLAG_KERNEL);
102
103                 /* prepare the slab and add it to cache empty list */
104                 slab_prepare(pool->cache, addr);
105                 slab_add(&pool->cache->empty, addr);
106
107                 size -= PAGE_SIZE;
108
109                 /* if the block contained only one page, we have nothing left
110                    to free */
111                 if(size == 0) {
112                     return;
113                 }
114
115                 addr += PAGE_SIZE;
116             }
117         }
118     }
119
120     link = (vm_link_t *)slab_alloc(pool->cache);
121     link->size = size;
122     link->addr = addr;
123
124     link->next = pool->head;
125     pool->head = link;
126 }

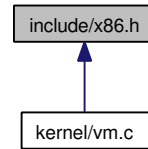
```

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

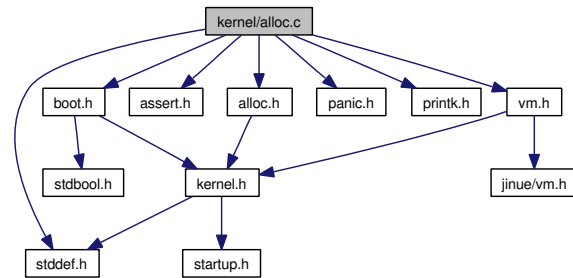
4.18.1.1 void **invalidate_tlb** (addr_t *vaddr*)

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()`.

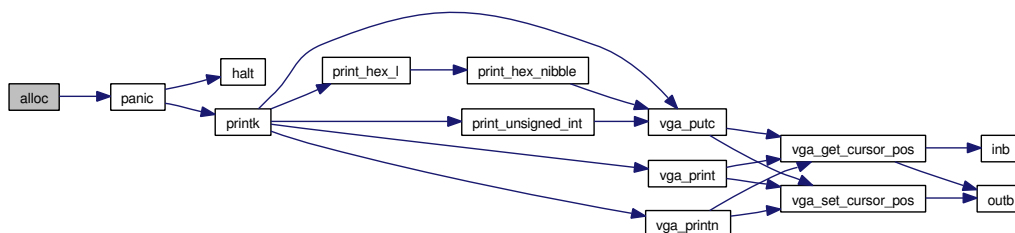
```
96      {
97      addr_t addr;
```

```

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
114
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;

```

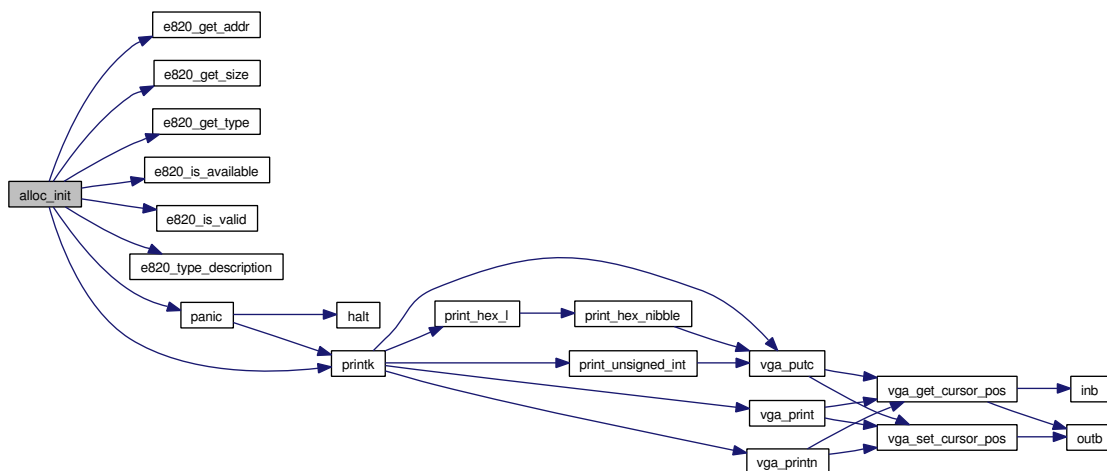
```
19     size_t fixed_size, best_size;
20
21     idx = 0;
22     best_size = 0;
23
24     printk("Dump of the BIOS memory map:\n");
25     printk("  address  size      type\n");
26     while( e820_is_valid(idx) ) {
27         addr = e820_get_addr(idx);
28         size = e820_get_size(idx);
29         type = e820_get_type(idx);
30         avail = e820_is_available(idx);
31
32         ++idx;
33
34         printk("%c %x %x %s\n",
35             avail?'*':' ',
36             addr,
37             size,
38             e820_type_description(type) );
39
40         if( !avail ) {
41             continue;
42         }
43
44         fixed_addr = addr;
45         fixed_size = size;
46
47         /* is the region completely under the kernel ? */
48         if(addr + size > kernel_start) {
49             /* is the region completely above the kernel ? */
50             if(addr < kernel_top) {
51                 /* if the region touches the kernel, we take only
52                  * the part above the kernel, if there is one... */
53                 if(addr + size <= kernel_top) {
54                     /* ... and apparently, there is none */
55                     continue;
56                 }
57
58                 fixed_addr = kernel_top;
59                 fixed_size -= fixed_addr - addr;
60             }
61         }
62
63         /* we must make sure the starting address is aligned on a
64          * page boundary. The size will eventually be divided
65          * by the page size, and thus need not be aligned. */
66         remainder = (unsigned int)fixed_addr % PAGE_SIZE;
67         if(remainder != 0) {
68             remainder = PAGE_SIZE - remainder;
69             if(fixed_size < remainder) {
70                 continue;
71             }
72
73             fixed_addr += remainder;
74             fixed_size -= remainder;
75     }
```

```

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     {

```

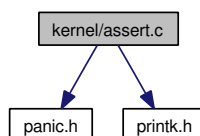
```
122     assert( PAGE_OFFSET_OF(addr) == 0 );  
123 }
```

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

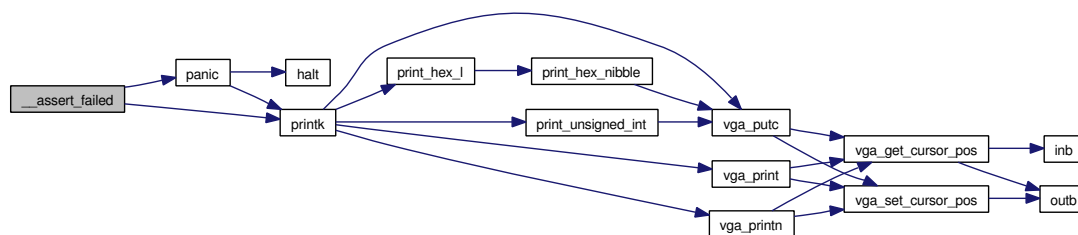
4.20.1.1 void __assert_failed (const char * *expr*, const char * *file*, unsigned int *line*, const char * *func*)

Definition at line 5 of file assert.c.

References panic(), and printk().

```
9             {
10
11     printk(
12         "ASSERTION FAILED [%s]: %s at line %u in function %s.\n",
13         expr, file, line, func );
14
15     panic("Assertion failed.");
16 }
```

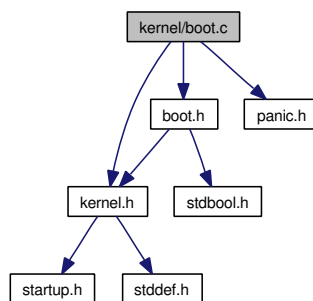
Here is the call graph for this function:



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()`.

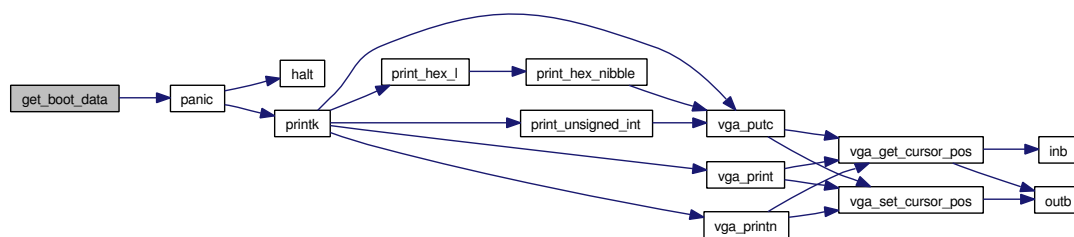
```

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().

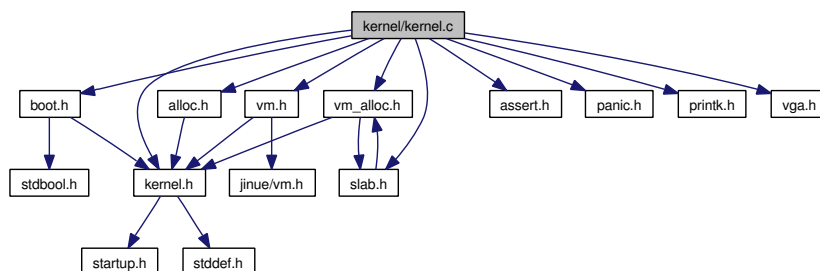
4.21.2.2 e820_t* e820_map

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)

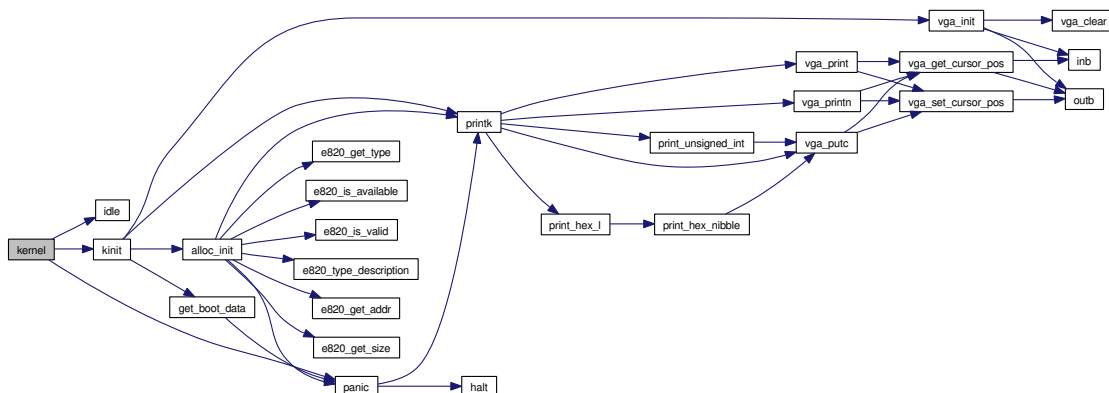
Definition at line 16 of file kernel.c.

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()`.

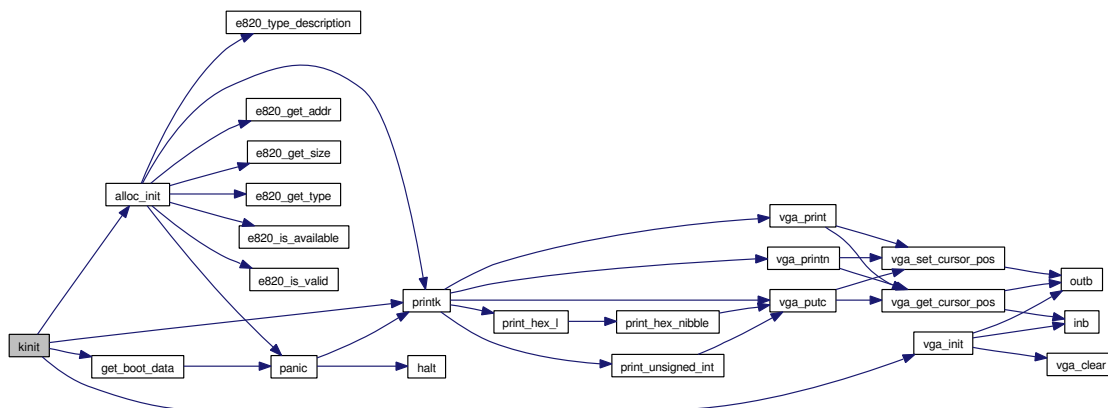
Referenced by `kernel()`.

```

23         {
24     boot_t *boot;
25     unsigned int remainder;
26
27     /* say hello */
28     vga_init();
29     printk("Kernel started.\n");
30
31
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.

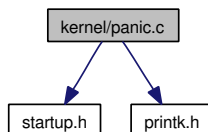
Referenced by `alloc_init()`, and `kinit()`.

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.

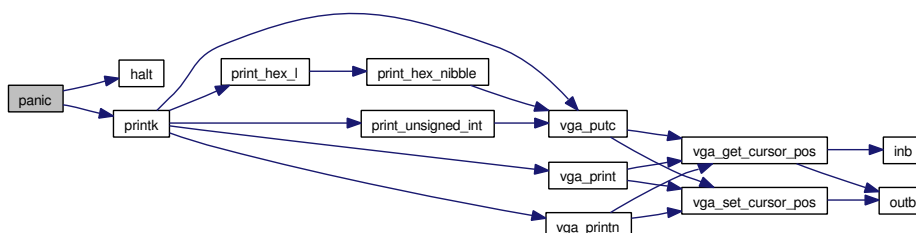
References `halt()`, and `printk()`.

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

```

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

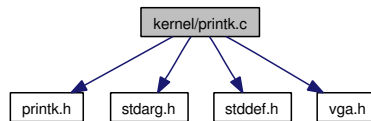
Here is the call graph for this function:



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

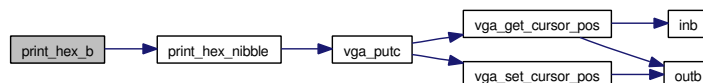
4.24.1.1 void print_hex_b (unsigned char *byte*)

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.

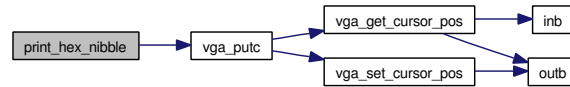
References vga_putc().

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.

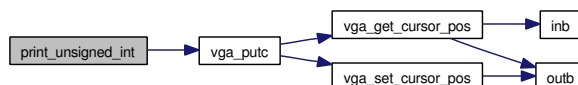
References vga_putc().

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()`.

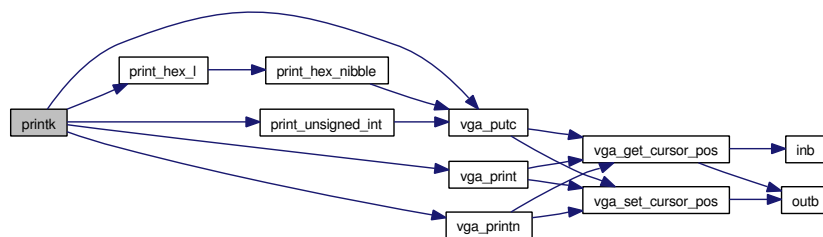
Referenced by `__assert_failed()`, `alloc_init()`, `kinit()`, and `panic()`.

```

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

```
9   ptrdiff_t size;
10
11   va_start(ap, format);
12
13   idx = format;
14
15   while(1) {
16       anchor = idx;
17
18       while( *idx != 0 && *idx != '%' ) {
19           ++idx;
20       }
21
22       size = idx - anchor;
23
24       if(size > 0) {
25           vga_printn(anchor, size);
26       }
27
28       if(*idx == 0 || *(idx+1) == 0) {
29           break;
30       }
31
32       ++idx;
33
34       switch( *idx ) {
35       case '%':
36           vga_putc('%');
37           break;
38
39       case 'c':
40           /* promotion, promotion */
41           vga_putc( (char)va_arg(ap, int) );
42           break;
43
44       case 's':
45           vga_print( va_arg(ap, const char *) );
46           break;
47
48       case 'u':
49           print_unsigned_int( va_arg(ap, unsigned int) );
50           break;
51
52       case 'x':
53           print_hex_l( va_arg(ap, unsigned long) );
54           break;
55
56       default:
57           va_end(ap);
58           return;
59       }
60
61       ++idx;
62   }
63
64   va_end(ap);
65 }
```

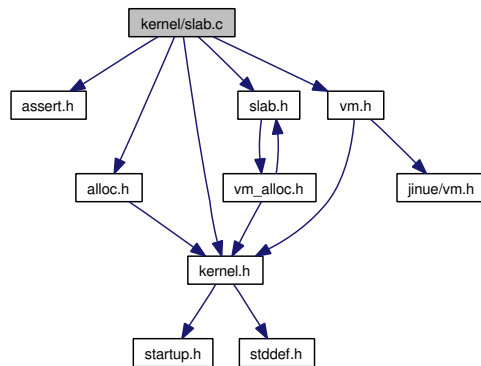
Here is the call graph for this function:



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().

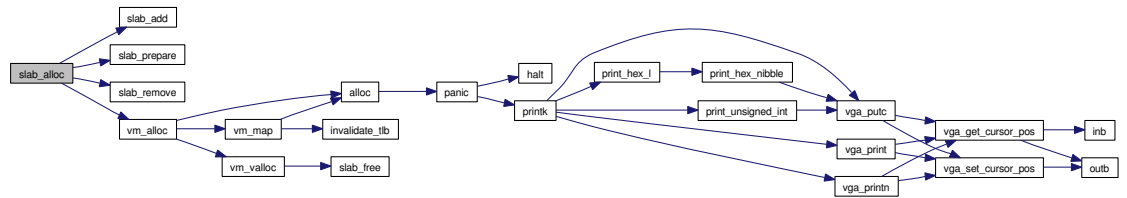
```

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 }
```



```
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);
46     }
47
48     return addr;
49 }
50
51 /* ... and, as last resort, allocate a slab */
53 slab = (slab_header_t *)vm_alloc(cache->vm_allocator, cache->vm_flags);
54 slab_prepare(cache, (addr_t)slab);
55
56 /* this slab is not empty since we are allocating an object from it */
57 slab_add(&cache->partial, slab);
58
59 addr = slab->free_list;
60 slab->free_list = *(addr_t *)addr;
61
62 /* maybe the slab is now full */
63 if(--slab->available == 0) {
64     slab_remove(&cache->partial, slab);
65     slab_add(&cache->full, slab);
66 }
67
68 return addr;
69 }
```

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
87     assert( PAGE_OFFSET_OF(page) == 0 );
88
89     assert( cache->per_slab > 0 );
90
91     assert( (*PDE_OF(page) & ~PAGE_MASK) != NULL && (*PDE_OF(page) & VM_FLAG_PRESENT) != 0 );
92     assert( (*PTE_OF(page) & ~PAGE_MASK) != NULL && (*PTE_OF(page) & VM_FLAG_PRESENT) != 0 );
93
94     obj_size = cache->obj_size;
95     per_slab = cache->per_slab;
96
97     /* initialize slab header */
98     slab = (slab_header_t *)page;
99     slab->available = per_slab;
100    slab->free_list = page + sizeof(slab_header_t);
101
102    /* create free list */
103    ptr = (addr_t *)slab->free_list;
104
105    for(cx = 0; cx < per_slab - 1; ++cx) {
106        next = ptr + obj_size;
107        *ptr = next;
108        ptr = (addr_t *)next;
109    }
110
111    *ptr = NULL;
112 }

```

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 }
```

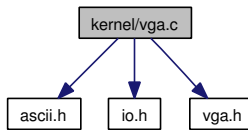
4.26 kernel/vga.c File Reference

```
#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.

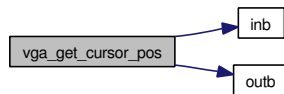
References `inb()`, `outb()`, `VGA_CRTC_ADDR`, and `VGA_CRTC_DATA`.

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.

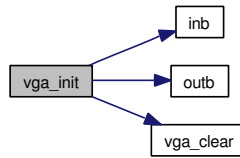
References `inb()`, `outb()`, `vga_clear()`, `VGA_CRTC_ADDR`, `VGA_CRTC_DATA`, `VGA_MISC_OUT_RD`, and `VGA_MISC_OUT_WR`.

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`.

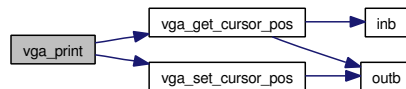
References `vga_get_cursor_pos()`, and `vga_set_cursor_pos()`.

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()`.

```

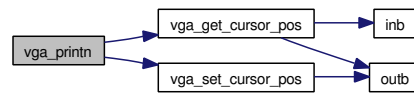
83      {
84      vga_pos_t pos = vga_get_cursor_pos();
85      char c;
86  }
```

```

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()`.

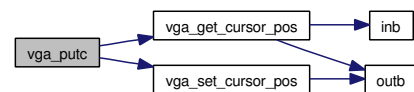
Referenced by `print_hex_nibble()`, `print_unsigned_int()`, and `printk()`.

```

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)

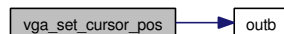
Definition at line 61 of file vga.c.

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 }
```

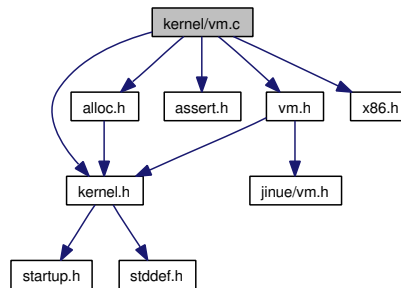
Here is the call graph for this function:



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.

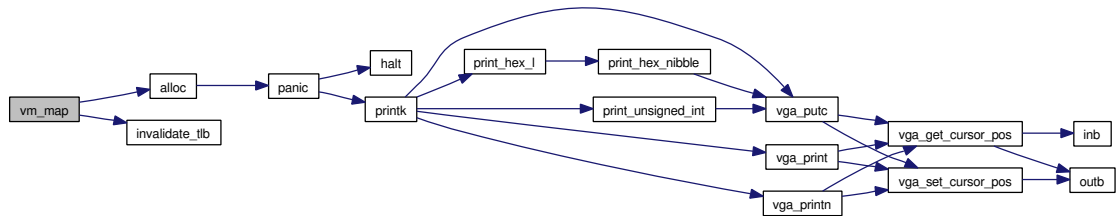
References `alloc()`, `assert`, `invalidate_tlb()`, `PAGE_OFFSET_OF`, `PAGE_SIZE`, `PAGE_TABLE_ENTRIES`, `PAGE_TABLE_OF`, `PAGE_TABLE_PTE_OF`, `PDE_OF`, `PTE_OF`, `VM_FLAG_PRESENT`, `VM_FLAG_USER`, and `VM_FLAGS_PAGE_TABLE`.

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

```

13                                     {
14     pte_t *pte, *pde;
15     addr_t page_table;
16     int idx;
17
18
19     assert( PAGE_OFFSET_OF(vaddr) == 0 );
20
21
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
66     assert( PAGE_OFFSET_OF(addr) == 0 );
67
68     pte = PTE_OF(addr);
69     *pte = NULL;
70
71     invalidate_tlb(addr);
72 }

```

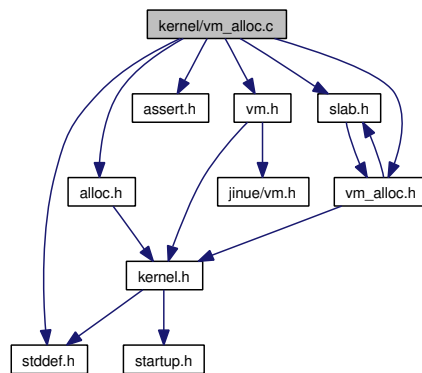
Here is the call graph for this function:



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`.

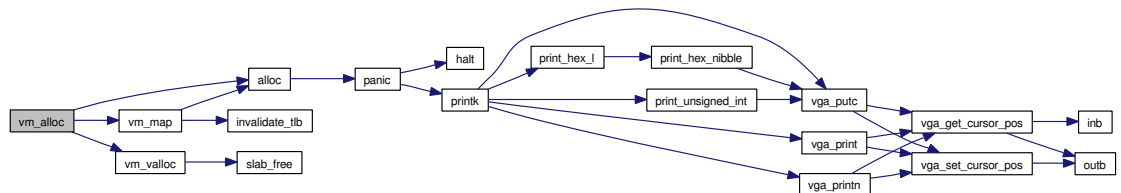
References `alloc()`, `PAGE_SIZE`, `vm_map()`, and `vm_valloc()`.

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.

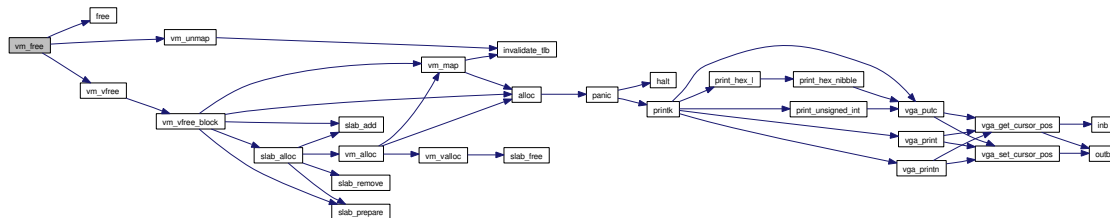
References `assert()`, `free()`, `NULL`, `PAGE_MASK`, `PTE_OF`, `vm_unmap()`, and `vm_vfree()`.

```

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

```

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
32     assert( PAGE_OFFSET_OF(size) == 0 );
33
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
45     assert( PAGE_OFFSET_OF(addr) == 0 );
46
47     return addr;
48 }

```



```
graph LR; vm_valloc --> slab_free
```

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

Parameters:

addr address of virtual page which must be freed

References PAGE_SIZE, and vm_vfree_block().

```

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

```

```

graph LR
    vm_vfree[vm_vfree] --> vm_vfree_block[vm_vfree_block]
    vm_vfree_block --> slab_alloc[slab_alloc]
    vm_vfree_block --> vm_map[vm_map]
    vm_vfree_block --> vm_remove[vm_remove]
    vm_vfree_block --> vm_valloc[vm_valloc]
    vm_vfree_block --> vm_alloc[vm_alloc]
    slab_alloc --> slab_remove[slab_remove]
    slab_alloc --> slab_add[slab_add]
    slab_alloc --> slab_prepare[slab_prepare]
    vm_remove --> slab_free[slab_free]
    vm_valloc --> slab_free
    vm_alloc --> vm_map
    vm_map --> invalidate_sb[invalidate_sb]
    invalidate_sb --> alloc[alloc]
    alloc --> panic[panic]
    panic --> halt[halt]
    panic --> printk[printk]
    printk --> print_hex_l[print_hex_l]
    printk --> print_hex_nibble[print_hex_nibble]
    printk --> print_unsigned_int[print_unsigned_int]
    print_hex_l --> vga_pulc[vga_pulc]
    print_hex_nibble --> vga_pulc
    print_unsigned_int --> vga_pulc
    vga_pulc --> vga_print[vga_print]
    vga_print --> vga_get_cursor_pos[vga_get_cursor_pos]
    vga_print --> vga_set_cursor_pos[vga_set_cursor_pos]
    vga_get_cursor_pos --> inb[inb]
    vga_set_cursor_pos --> outb[outb]
  
```

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

Generated on Sat Jul 18 14:18:05 2009 for Jinue by Doxygen

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().

```

71                                     {
72     addr_t phys_page;
73     vm_link_t *link;
74
75     assert( PAGE_OFFSET_OF(addr) == 0 );
76
77     assert( PAGE_OFFSET_OF(size) == 0 );
78
79     assert( addr != (addr_t)NULL );
80
81
82     /* The virtual address space allocator needs a slab cache from which to
83        allocate data structures for its free list. Also, each slab cache needs
84        a virtual address space allocator to allocate slabs when needed.
85
86        There can be a mutual dependency between the virtual address space
87        allocator and the slab cache. This is not a problem in general, but a
88        special bootstrapping procedure is needed for initialization of the
89        virtual address space allocator in that case. The virtual address space
90        allocator will actually "donate" a virtual page (backed by physical ram)
91        to the cache for use as a slab.
92
93        This case is handled here
94
95        */
96     if(pool->head == NULL) {
97         if(pool->cache->vm_allocator == pool) {
98             if(pool->cache->empty == NULL && pool->cache->partial == NULL) {
99                 /* allocate a physical page for slab */
100                phys_page = alloc(PAGE_SIZE);
101
102
103                /* map page */
104                vm_map(addr, phys_page, VM_FLAG_KERNEL);
105
106                /* prepare the slab and add it to cache empty list */

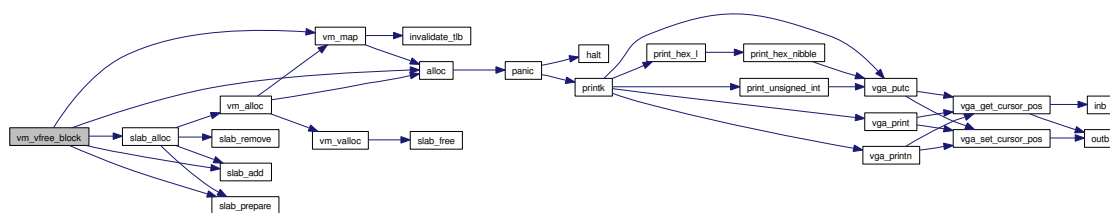
```

```

107         slab_prepare(pool->cache, addr);
108         slab_add(&pool->cache->empty, addr);
109
110         size -= PAGE_SIZE;
111
112         /* if the block contained only one page, we have nothing left
113            to free */
114         if(size == 0) {
115             return;
116         }
117
118         addr += PAGE_SIZE;
119     }
120 }
121 }
122
123 link = (vm_link_t *)slab_alloc(pool->cache);
124 link->size = size;
125 link->addr = addr;
126
127 link->next = pool->head;
128 pool->head = link;
129 }

```

Here is the call graph for this function:



Index

- `__assert_failed`
 - `assert.c`, 90
 - `assert.h`, 23
 - `__bool_true_false_are_defined`
 - `stdbool.h`, 64
- `addr`
 - `e820_t`, 7
 - `vm_link_t`, 15
- `addr_t`
 - `kernel.h`, 45
- `alloc`
 - `alloc.c`, 85
 - `alloc.h`, 18
- `alloc.c`
 - `alloc`, 85
 - `alloc_init`, 86
 - `free`, 88
- `alloc.h`
 - `alloc`, 18
 - `alloc_init`, 18
 - `free`, 20
- `alloc_init`
 - `alloc.c`, 86
 - `alloc.h`, 18
- `ascii.h`
 - `CHAR_BS`, 22
 - `CHAR_CR`, 22
 - `CHAR_HT`, 22
 - `CHAR_LF`, 22
- `assert`
 - `assert.h`, 23
- `assert.c`
 - `__assert_failed`, 90
- `assert.h`
 - `__assert_failed`, 23
 - `assert`, 23
- `available`
 - `slab_header_t`, 11
- `bool`
 - `stdbool.h`, 64
- `boot.c`
 - `boot_setup_addr`, 95
 - `e820_get_addr`, 92
 - `e820_get_size`, 93
 - `e820_get_type`, 93
 - `e820_is_available`, 93
 - `e820_is_valid`, 93
 - `e820_map`, 95
 - `e820_type_description`, 94
 - `get_boot_data`, 94
- `boot.h`
 - `BOOT_MAGIC`, 26
 - `BOOT_SIGNATURE`, 26
 - `E820_ACPI`, 26
 - `e820_addr_t`, 27
 - `e820_get_addr`, 27
 - `e820_get_size`, 27
 - `e820_get_type`, 27
 - `e820_is_available`, 28
 - `e820_is_valid`, 28
 - `E820_RAM`, 26
 - `E820_RESERVED`, 26
 - `e820_size_t`, 27
 - `e820_type_description`, 28
 - `e820_type_t`, 27
 - `get_boot_data`, 29
 - `SETUP_HEADER`, 26
- `BOOT_MAGIC`
 - `boot.h`, 26
- `boot_setup_addr`
 - `boot.c`, 95
- `BOOT_SIGNATURE`

- boot.h, 26
- boot_t, 5
 - magic, 5
 - ram_size, 6
 - root_dev, 6
 - root_flags, 6
 - setup_sects, 6
 - signature, 6
 - sysize, 6
 - vid_mode, 6
- cache
 - vm_alloc_t, 14
- CHAR_BS
 - ascii.h, 22
- CHAR_CR
 - ascii.h, 22
- CHAR_HT
 - ascii.h, 22
- CHAR_LF
 - ascii.h, 22
- count_t
 - kernel.h, 45
- E820_ACPI
 - boot.h, 26
- e820_addr_t
 - boot.h, 27
- e820_get_addr
 - boot.c, 92
 - boot.h, 27
- e820_get_size
 - boot.c, 93
 - boot.h, 27
- e820_get_type
 - boot.c, 93
 - boot.h, 27
- e820_is_available
 - boot.c, 93
 - boot.h, 28
- e820_is_valid
 - boot.c, 93
 - boot.h, 28
- e820_map
 - boot.c, 95
- E820_RAM
 - boot.h, 26
- E820_RESERVED
 - boot.h, 26
- e820_size_t
 - boot.h, 27
- e820_t, 7
 - addr, 7
 - size, 7
 - type, 7
- e820_type_description
 - boot.c, 94
 - boot.h, 28
- e820_type_t
 - boot.h, 27
- empty
 - slab_cache_t, 9
- false
 - stdbool.h, 64
- free
 - alloc.c, 88
 - alloc.h, 20
- free_list
 - slab_header_t, 11
- full
 - slab_cache_t, 9
- get_boot_data
 - boot.c, 94
 - boot.h, 29
- halt
 - startup.h, 61
- head
 - vm_alloc_t, 13
- idle
 - kernel.c, 97
 - kernel.h, 45
- inb
 - io.h, 30
- include/alloc.h, 17
- include/ascii.h, 22
- include/assert.h, 23
- include/boot.h, 25
- include/io.h, 30

- include/jinue/vm.h, 31
- include/kernel.h, 44
- include/panic.h, 48
- include/printk.h, 49
- include/slab.h, 55
- include/startup.h, 61
- include/stdarg.h, 62
- include/stdbool.h, 64
- include/stddef.h, 65
- include/vga.h, 67
- include/vm.h, 34
- include/vm_alloc.h, 76
- include/x86.h, 84
- inl
 - io.h, 30
- invalidate_tlb
 - x86.h, 84
- inw
 - io.h, 30
- io.h
 - inb, 30
 - inl, 30
 - inw, 30
 - outb, 30
 - outl, 30
 - outw, 30
- jinue/vm.h
 - KLIMIT, 32
 - PAGE_BITS, 32
 - PAGE_DIRECTORY_ADDR, 32
 - PAGE_SIZE, 32
 - PAGE_TABLE_BITS, 32
 - PAGE_TABLE_ENTRIES, 33
 - PAGE_TABLE_SIZE, 33
 - PAGE_TABLES_ADDR, 33
 - PLIMIT, 33
 - PTE_SIZE, 33
- kernel
 - kernel.c, 97
 - kernel.h, 45
- kernel.c
 - idle, 97
 - kernel, 97
 - kernel_size, 99
 - kernel_top, 99
 - kinit, 97
- kernel.h
 - addr_t, 45
 - count_t, 45
 - idle, 45
 - kernel, 45
 - kernel_size, 47
 - kernel_start, 45
 - kernel_top, 47
 - kinit, 46
- kernel/alloc.c, 85
- kernel/assert.c, 90
- kernel/boot.c, 92
- kernel/kernel.c, 96
- kernel/panic.c, 100
- kernel/printk.c, 101
- kernel/slab.c, 107
- kernel/vga.c, 113
- kernel/vm.c, 118
- kernel/vm_alloc.c, 121
- kernel_size
 - kernel.c, 99
 - kernel.h, 47
- kernel_start
 - kernel.h, 45
- kernel_top
 - kernel.c, 99
 - kernel.h, 47
- kinit
 - kernel.c, 97
 - kernel.h, 46
- KLIMIT
 - jinue/vm.h, 32
- magic
 - boot_t, 5
- next
 - slab_header_t, 12
 - vm_link_t, 15
- NULL
 - stddef.h, 65
- obj_size

- slab_cache_t, 9
- offsetof
 - stddef.h, 65
- outb
 - io.h, 30
- outl
 - io.h, 30
- outw
 - io.h, 30
- PAGE_BITS
 - jinue/vm.h, 32
- PAGE_DIRECTORY
 - vm.h, 37
- PAGE_DIRECTORY_ADDR
 - jinue/vm.h, 32
- PAGE_DIRECTORY_OFFSET_OF
 - vm.h, 37
- PAGE_MASK
 - vm.h, 37
- PAGE_OFFSET_OF
 - vm.h, 37
- PAGE_SIZE
 - jinue/vm.h, 32
- PAGE_TABLE_BITS
 - jinue/vm.h, 32
- PAGE_TABLE_ENTRIES
 - jinue/vm.h, 33
- PAGE_TABLE_MASK
 - vm.h, 37
- PAGE_TABLE_OF
 - vm.h, 38
- PAGE_TABLE_OFFSET_OF
 - vm.h, 38
- PAGE_TABLE_PTE_OF
 - vm.h, 38
- PAGE_TABLE_SIZE
 - jinue/vm.h, 33
- page_table_t
 - vm.h, 41
- PAGE_TABLES
 - vm.h, 38
- PAGE_TABLES_ADDR
 - jinue/vm.h, 33
- PAGE_TABLES_TABLE
 - vm.h, 38
- panic
 - panic.c, 100
 - panic.h, 48
- panic.c
 - panic, 100
- panic.h
 - panic, 48
- partial
 - slab_cache_t, 9
- PDE_OF
 - vm.h, 38
- per_slab
 - slab_cache_t, 9
- PLIMIT
 - jinue/vm.h, 33
- PMAPPING_END
 - vm.h, 39
- PMAPPING_START
 - vm.h, 39
- prev
 - slab_header_t, 12
- print_hex_b
 - printk.c, 101
 - printk.h, 49
- print_hex_l
 - printk.c, 101
 - printk.h, 49
- print_hex_nibble
 - printk.c, 102
 - printk.h, 50
- print_hex_q
 - printk.c, 103
 - printk.h, 50
- print_hex_w
 - printk.c, 103
 - printk.h, 51
- print_unsigned_int
 - printk.c, 104
 - printk.h, 51
- printk
 - printk.c, 104
 - printk.h, 52
- printk.c
 - print_hex_b, 101
 - print_hex_l, 101

- print_hex_nibble, 102
- print_hex_q, 103
- print_hex_w, 103
- print_unsigned_int, 104
- printk, 104
- printk.h
 - print_hex_b, 49
 - print_hex_l, 49
 - print_hex_nibble, 50
 - print_hex_q, 50
 - print_hex_w, 51
 - print_unsigned_int, 51
 - printk, 52
- PTE_OF
 - vm.h, 39
- PTE_SIZE
 - jinue/vm.h, 33
- pte_t
 - vm.h, 41
- ptrdiff_t
 - stddef.h, 66
- ram_size
 - boot_t, 6
- root_dev
 - boot_t, 6
- root_flags
 - boot_t, 6
- SETUP_HEADER
 - boot.h, 26
- setup_sects
 - boot_t, 6
- signature
 - boot_t, 6
- size
 - e820_t, 7
 - vm_alloc_t, 13
 - vm_link_t, 15
- size_t
 - stddef.h, 66
- slab.c
 - slab_add, 108
 - slab_alloc, 108
 - slab_create, 110
 - slab_destroy, 110
- slab_free, 110
- slab_prepare, 110
- slab_remove, 111
- slab.h
 - slab_add, 56
 - slab_alloc, 57
 - slab_cache_t, 56
 - slab_create, 58
 - slab_destroy, 58
 - slab_free, 58
 - slab_header_t, 56
 - slab_prepare, 59
 - slab_remove, 60
- slab_add
 - slab.c, 108
 - slab.h, 56
- slab_alloc
 - slab.c, 108
 - slab.h, 57
- slab_cache_t, 8
 - empty, 9
 - full, 9
 - obj_size, 9
 - partial, 9
 - per_slab, 9
 - slab.h, 56
 - vm_allocator, 10
 - vm_flags, 9
- slab_create
 - slab.c, 110
 - slab.h, 58
- slab_destroy
 - slab.c, 110
 - slab.h, 58
- slab_free
 - slab.c, 110
 - slab.h, 58
- slab_header_t, 11
 - available, 11
 - free_list, 11
 - next, 12
 - prev, 12
 - slab.h, 56
- slab_prepare
 - slab.c, 110
 - slab.h, 59

-
- slab_remove
 - slab.c, 111
 - slab.h, 60
 - start
 - startup.h, 61
 - startup.h
 - halt, 61
 - start, 61
 - stdarg.h
 - va_arg, 62
 - va_copy, 62
 - va_end, 62
 - va_list, 63
 - va_start, 62
 - stdbool.h
 - __bool_true_false_are_defined, 64
 - bool, 64
 - false, 64
 - true, 64
 - stddef.h
 - NULL, 65
 - offsetof, 65
 - ptrdiff_t, 66
 - size_t, 66
 - wchar_t, 66
 - sysize
 - boot_t, 6
 - true
 - stdbool.h, 64
 - type
 - e820_t, 7
 - va_arg
 - stdarg.h, 62
 - va_copy
 - stdarg.h, 62
 - va_end
 - stdarg.h, 62
 - va_list
 - stdarg.h, 63
 - va_start
 - stdarg.h, 62
 - vga.c
 - vga_clear, 113
 - vga_get_cursor_pos, 113
 - vga_init, 114
 - vga_print, 115
 - vga_printn, 115
 - vga_putc, 116
 - vga_scroll, 116
 - vga_set_cursor_pos, 117
 - vga.h
 - vga_clear, 71
 - VGA_COL, 68
 - VGA_COLOR_BLACK, 68
 - VGA_COLOR_BLUE, 68
 - VGA_COLOR_BRIGHTBLUE, 68
 - VGA_COLOR_BRIGHTCYAN, 68
 - VGA_COLOR_BRIGHTGREEN, 68
 - VGA_COLOR_BRIGHTMAGENTA, 68
 - VGA_COLOR_BRIGHTRED, 69
 - VGA_COLOR_BRIGHTWHITE, 69
 - VGA_COLOR_BROWN, 69
 - VGA_COLOR_CYAN, 69
 - VGA_COLOR_DEFAULT, 69
 - VGA_COLOR_ERASE, 69
 - VGA_COLOR_GRAY, 69
 - VGA_COLOR_GREEN, 69
 - VGA_COLOR_MAGENTA, 69
 - VGA_COLOR_RED, 70
 - VGA_COLOR_WHITE, 70
 - VGA_COLOR_YELLOW, 70
 - VGA_CRTC_ADDR, 70
 - VGA_CRTC_DATA, 70
 - VGA_FB_FLAG_ACTIVE, 70
 - vga_get_cursor_pos, 72
 - vga_init, 72
 - VGA_LINE, 70
 - VGA_LINES, 70
 - VGA_MISC_OUT_RD, 71
 - VGA_MISC_OUT_WR, 71
 - vga_pos_t, 71
 - vga_print, 73
 - vga_printn, 74
-

- vga_putc, 74
- vga_scroll, 74
- vga_set_cursor_pos, 75
- VGA_TAB_WIDTH, 71
- VGA_TEXT_VID_BASE, 71
- VGA_WIDTH, 71
- vga_clear
 - vga.c, 113
 - vga.h, 71
- VGA_COL
 - vga.h, 68
- VGA_COLOR_BLACK
 - vga.h, 68
- VGA_COLOR_BLUE
 - vga.h, 68
- VGA_COLOR_BRIGHTBLUE
 - vga.h, 68
- VGA_COLOR_BRIGHTCYAN
 - vga.h, 68
- VGA_COLOR_BRIGHTGREEN
 - vga.h, 68
- VGA_COLOR_ -
 - BRIGHTMAGENTA
 - vga.h, 68
- VGA_COLOR_BRIGHTRED
 - vga.h, 69
- VGA_COLOR_BRIGHTWHITE
 - vga.h, 69
- VGA_COLOR_BROWN
 - vga.h, 69
- VGA_COLOR_CYAN
 - vga.h, 69
- VGA_COLOR_DEFAULT
 - vga.h, 69
- VGA_COLOR_ERASE
 - vga.h, 69
- VGA_COLOR_GRAY
 - vga.h, 69
- VGA_COLOR_GREEN
 - vga.h, 69
- VGA_COLOR_MAGENTA
 - vga.h, 69
- VGA_COLOR_RED
 - vga.h, 70
- VGA_COLOR_WHITE
 - vga.h, 70
- VGA_COLOR_YELLOW
 - vga.h, 70
- VGA_CRTC_ADDR
 - vga.h, 70
- VGA_CRTC_DATA
 - vga.h, 70
- VGA_FB_FLAG_ACTIVE
 - vga.h, 70
- vga_get_cursor_pos
 - vga.c, 113
 - vga.h, 72
- vga_init
 - vga.c, 114
 - vga.h, 72
- VGA_LINE
 - vga.h, 70
- VGA_LINES
 - vga.h, 70
- VGA_MISC_OUT_RD
 - vga.h, 71
- VGA_MISC_OUT_WR
 - vga.h, 71
- vga_pos_t
 - vga.h, 71
- vga_print
 - vga.c, 115
 - vga.h, 73
- vga_printn
 - vga.c, 115
 - vga.h, 74
- vga_putc
 - vga.c, 116
 - vga.h, 74
- vga_scroll
 - vga.c, 116
 - vga.h, 74
- vga_set_cursor_pos
 - vga.c, 117
 - vga.h, 75
- VGA_TAB_WIDTH
 - vga.h, 71
- VGA_TEXT_VID_BASE
 - vga.h, 71
- VGA_WIDTH
 - vga.h, 71
- vid_mode

- boot_t, 6
- vm.c
 - vm_map, 118
 - vm_unmap, 120
- vm.h
 - PAGE_DIRECTORY, 37
 - PAGE_DIRECTORY_-
 OFFSET_OF, 37
 - PAGE_MASK, 37
 - PAGE_OFFSET_OF, 37
 - PAGE_TABLE_MASK, 37
 - PAGE_TABLE_OF, 38
 - PAGE_TABLE_OFFSET_OF,
 38
 - PAGE_TABLE_PTE_OF, 38
 - page_table_t, 41
 - PAGE_TABLES, 38
 - PAGE_TABLES_TABLE, 38
 - PDE_OF, 38
 - PMAPPING_END, 39
 - PMAPPING_START, 39
 - PTE_OF, 39
 - pte_t, 41
 - VM_FLAG_ACCESSED, 39
 - VM_FLAG_BIG_PAGE, 39
 - VM_FLAG_CACHE_-
 DISABLE, 39
 - VM_FLAG_DIRTY, 40
 - VM_FLAG_GLOBAL, 40
 - VM_FLAG_KERNEL, 40
 - VM_FLAG_PRESENT, 40
 - VM_FLAG_READ_ONLY, 40
 - VM_FLAG_USER, 40
 - VM_FLAG_WRITE_-
 THROUGH, 40
 - VM_FLAGS_PAGE_TABLE,
 41
 - vm_map, 41
 - vm_unmap, 43
- vm_alloc
 - vm_alloc.c, 122
 - vm_alloc.h, 77
- vm_alloc.c
 - vm_alloc, 122
 - vm_free, 122
 - vm_valloc, 123
- vm_vfree, 125
- vm_vfree_block, 125
- vm_alloc.h
 - vm_alloc, 77
 - vm_alloc_t, 77
 - vm_free, 78
 - vm_link_t, 77
 - vm_valloc, 79
 - vm_vfree, 80
 - vm_vfree_block, 81
- vm_alloc_t, 13
 - cache, 14
 - head, 13
 - size, 13
 - vm_alloc.h, 77
- vm_allocator
 - slab_cache_t, 10
- VM_FLAG_ACCESSED
 - vm.h, 39
- VM_FLAG_BIG_PAGE
 - vm.h, 39
- VM_FLAG_CACHE_DISABLE
 - vm.h, 39
- VM_FLAG_DIRTY
 - vm.h, 40
- VM_FLAG_GLOBAL
 - vm.h, 40
- VM_FLAG_KERNEL
 - vm.h, 40
- VM_FLAG_PRESENT
 - vm.h, 40
- VM_FLAG_READ_ONLY
 - vm.h, 40
- VM_FLAG_USER
 - vm.h, 40
- VM_FLAG_WRITE_THROUGH
 - vm.h, 40
- vm_flags
 - slab_cache_t, 9
- VM_FLAGS_PAGE_TABLE
 - vm.h, 41
- vm_free
 - vm_alloc.c, 122
 - vm_alloc.h, 78
- vm_link_t, 15
 - addr, 15

- next, 15
 - size, 15
 - vm_alloc.h, 77
- vm_map
 - vm.c, 118
 - vm.h, 41
- vm_unmap
 - vm.c, 120
 - vm.h, 43
- vm_valloc
 - vm_alloc.c, 123
 - vm_alloc.h, 79
- vm_vfree
 - vm_alloc.c, 125
 - vm_alloc.h, 80
- vm_vfree_block
 - vm_alloc.c, 125
 - vm_alloc.h, 81
- wchar_t
 - stddef.h, 66
- x86.h
 - invalidate_tlb, 84