Jinue

Generated by Doxygen 1.5.5

Mon Jul 20 14:42:50 2009

Contents

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 boot_t Struct Reference

`#include <boot.h>`

3.1.1 Detailed Description

Definition at line 26 of file boot.h.

Data Fields

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

3.1.2 Field Documentation

3.1.2.1 unsigned long boot_t::magic

Definition at line 27 of file boot.h.

Referenced by get_boot_data().

3.1.2.2 unsigned char boot_t::setup_sects

Definition at line 28 of file boot.h.

3.1.2.3 unsigned short boot_t::root_flags

Definition at line 29 of file boot.h.

3.1.2.4 unsigned long boot_t::sysize

Definition at line 30 of file boot.h.

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().

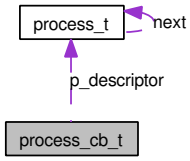
The documentation for this struct was generated from the following file:

- include/**boot.h**

3.3 process_cb_t Struct Reference

`#include <process.h>`

Collaboration diagram for process_cb_t:



3.3.1 Detailed Description

Definition at line 20 of file process.h.

Data Fields

- **process_t * p_descriptor**

3.3.2 Field Documentation

3.3.2.1 process_t* process_cb_t::p_descriptor

Definition at line 21 of file process.h.

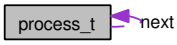
The documentation for this struct was generated from the following file:

- include/**process.h**

3.4 process_t Struct Reference

`#include <process.h>`

Collaboration diagram for process_t:



3.4.1 Detailed Description

Definition at line 11 of file process.h.

Data Fields

- **pid_t pid**
- **addr_t cr3**
- struct **process_t** * **next**
- char **name** [PROCESS_NAME_LENGTH]

3.4.2 Field Documentation

3.4.2.1 pid_t process_t::pid

Definition at line 12 of file process.h.

Referenced by kinit(), process_create(), and process_find_by_pid().

3.4.2.2 addr_t process_t::cr3

Definition at line 13 of file process.h.

Referenced by kinit().

3.4.2.3 struct process_t* process_t::next [read]

Definition at line 14 of file process.h.

Referenced by kinit(), and process_find_by_pid().

3.4.2.4 char process_t::name[PROCESS_NAME_LENGTH]

Definition at line 15 of file process.h.

Referenced by kinit().

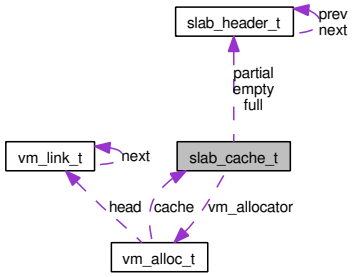
The documentation for this struct was generated from the following file:

- include/**process.h**

3.5 slab_cache_t Struct Reference

`#include <slab.h>`

Collaboration diagram for slab_cache_t:



3.5.1 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

3.5.2 Field Documentation

3.5.2.1 size_t slab_cache_t::obj_size

size of objects to allocate

Definition at line 26 of file slab.h.

Referenced by slab_create(), and slab_prepare().

3.5.2.2 count_t slab_cache_t::per_slab

number of objects per slab

Definition at line 29 of file slab.h.

Referenced by slab_create(), and slab_prepare().

3.5.2.3 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(), slab_create(), and vm_vfree_block().

3.5.2.4 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(), slab_create(), and vm_vfree_block().

3.5.2.5 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(), and slab_create().

3.5.2.6 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(), and slab_create().

3.5.2.7 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(), slab_create(), and vm_vfree_block().

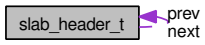
The documentation for this struct was generated from the following file:

- include/**slab.h**

3.6 slab_header_t Struct Reference

`#include <slab.h>`

Collaboration diagram for slab_header_t:



3.6.1 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

3.6.2 Field Documentation

3.6.2.1 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().

3.6.2.2 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().

3.6.2.3 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().

3.6.2.4 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**

3.7 thread_t Struct Reference

`#include <process.h>`

3.7.1 Detailed Description

Definition at line 26 of file process.h.

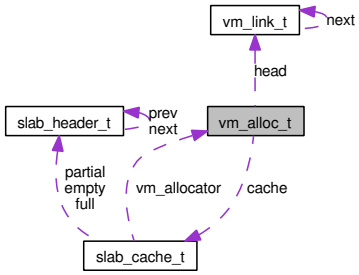
The documentation for this struct was generated from the following file:

- include/**process.h**

3.8 vm_alloc_t Struct Reference

`#include <vm_alloc.h>`

Collaboration diagram for vm_alloc_t:



3.8.1 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

3.8.2 Field Documentation

3.8.2.1 size_t vm_alloc_t::size

total amount of memory available

Definition at line 25 of file vm_alloc.h.

Referenced by vm_create_pool().

3.8.2.2 vm_link_t* vm_alloc_t::head

head of the free list

Definition at line 28 of file vm_alloc.h.

Referenced by vm_create_pool(), vm_valloc(), and vm_vfree_block().

3.8.2.3 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_create_pool(), vm_valloc(), and vm_vfree_block().

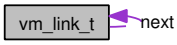
The documentation for this struct was generated from the following file:

- include/**vm_alloc.h**

3.9 vm_link_t Struct Reference

`#include <vm_alloc.h>`

Collaboration diagram for vm_link_t:



3.9.1 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

3.9.2 Field Documentation

3.9.2.1 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().

3.9.2.2 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().

3.9.2.3 addr_t vm_link_t::addr

starting address of current block

Definition at line 16 of file vm_alloc.h.

Referenced by vm_valloc(), and vm_vfree_block().

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

- include/**vm_alloc.h**

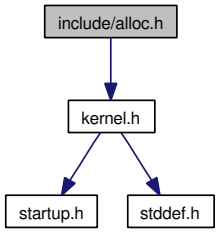
Chapter 4

File Documentation

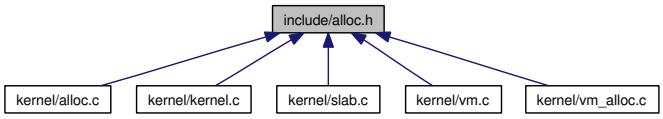
4.1 include/alloc.h File Reference

`#include <kernel.h>`

Include dependency graph for alloc.h:



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



Functions

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

4.1.1 Function Documentation

4.1.1.1 addr_t alloc (size_t *size*)

ASSERTION: returned address should be aligned with a page boundary

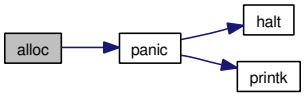
Definition at line 97 of file alloc.c.

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

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

```
97                          {
98     addr_t addr;
99     size_t pages;
100
101     pages = size >> PAGE_BITS;
102
103     if( (size & PAGE_MASK) != 0 ) {
104         ++pages;
105     }
106
107     if(_alloc_size < pages) {
108         panic("out of memory.");
109     }
110
111     addr = _alloc_addr;
112     _alloc_addr += pages * PAGE_SIZE;
113     _alloc_size -= pages;
114
116     assert( ((unsigned long)addr & PAGE_MASK) == 0 );
117
118     return addr;
119 }
```

Here is the call graph for this function:



4.1.1.2 void alloc_init (void)

Definition at line 13 of file alloc.c.

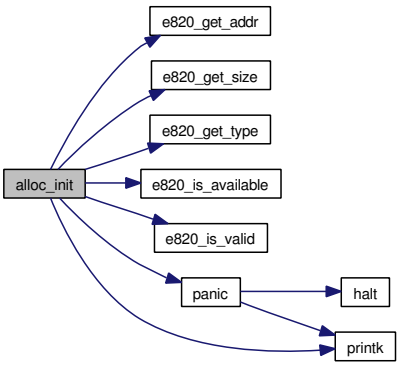
References e820_get_addr(), e820_get_size(), e820_get_type(), e820_is_available(), e820_is_valid(), kernel_region_top, kernel_start, PAGE_SIZE, panic(), and printk().

Referenced by kinit().

```
13                      {
14      unsigned int idx;
15      unsigned int remainder;
16      bool avail;
17      size_t size;
18      e820_type_t type;
19      addr_t addr, fixed_addr, best_addr;
20      size_t fixed_size, best_size;
21
22      idx = 0;
23      best_size = 0;
24
25      /*printk("Dump of the BIOS memory map:\n");
26      printk("  address  size     type\n");*/
27      while( e820_is_valid(idx) ) {
28          addr = e820_get_addr(idx);
29          size = e820_get_size(idx);
30          type = e820_get_type(idx);
31          avail = e820_is_available(idx);
32
33          ++idx;
34
35          /*printk("(%x) %c %x %x %s\n",
36              avail?'*':' ',
37              addr,
38              size,
39              e820_type_description(type) );*/
40
41          if( !avail ) {
42              continue;
43          }
44
45          fixed_addr = addr;
46          fixed_size = size;
47
48          /* is the region completely under the kernel ? */
49          if(addr + size > kernel_start) {
50              /* is the region completely above the kernel ? */
51              if(addr < kernel_region_top) {
52                  /* if the region touches the kernel, we take only
53                   * the part above the kernel, if there is one... */
54                  if(addr + size <= kernel_region_top) {
55                      /* ... and apparently, there is none */
56                      continue;
57                  }
58
59                  fixed_addr = kernel_region_top;
60                  fixed_size -= fixed_addr - addr;
61              }
62          }
63
64          /* we must make sure the starting address is aligned on a
65           * page boundary. The size will eventually be divided
66           * by the page size, and thus need not be aligned. */
67          remainder = (unsigned int)fixed_addr % PAGE_SIZE;
68          if(remainder != 0) {
69              remainder = PAGE_SIZE - remainder;
```

```
70             if(fixed_size < remainder) {
71                 continue;
72             }
73
74             fixed_addr += remainder;
75             fixed_size -= remainder;
76         }
77
78         if(fixed_size > best_size) {
79             best_addr = fixed_addr;
80             best_size = fixed_size;
81         }
82     }
83
84     _alloc_addr = (addr_t)best_addr;
85     _alloc_size = best_size / PAGE_SIZE;
86
87     if(_alloc_size == 0) {
88         panic("no memory to allocate.");
89     }
90
91     printk("%u kilobytes (%u pages) available starting at %xh.\n",
92         _alloc_size * PAGE_SIZE / 1024,
93         _alloc_size,
94         _alloc_addr );
95 }
```

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 121 of file alloc.c.

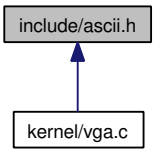
References assert, and PAGE_OFFSET_OF.

Referenced by vm_free().

```
121                         {
123     assert( PAGE_OFFSET_OF(addr) == 0 );
124 }
```

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

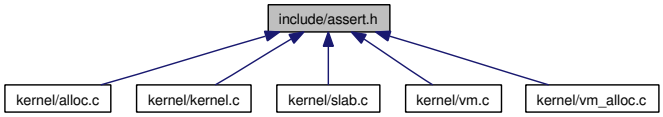
Definition at line 5 of file ascii.h.

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

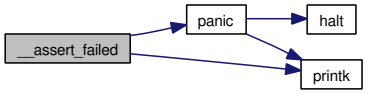
4.3.2.1 void __assert_failed (const char * *expr*, const char * *file*, unsigned int *line*, const char * *func*)

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:

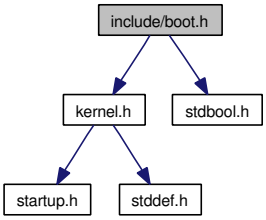


4.4 include/boot.h File Reference

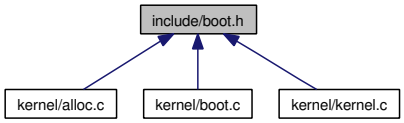
`#include <kernel.h>`

`#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.

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.

```
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)

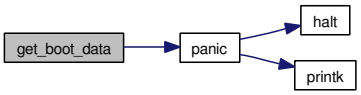
Definition at line 45 of file boot.c.

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

Referenced by kinit().

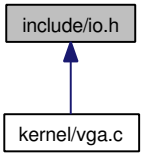
```
45                              {
46     boot_t *boot;
47
48     boot = (boot_t *)boot_data;
49
50     if(boot->signature != BOOT_SIGNATURE) {
51         panic("bad boot sector signature.");
52     }
53
54     if(boot->magic != BOOT_MAGIC) {
55         panic("bad boot sector magic.");
56     }
57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

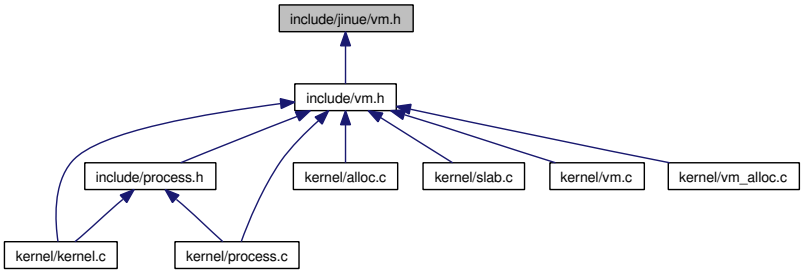
Referenced by vga_get_cursor_pos(), vga_init(), and vga_set_cursor_pos().

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.

Referenced by kinit().

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.

Referenced by kinit().

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(), slab_create(), 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 kinit(), and 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.

Referenced by kinit().

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

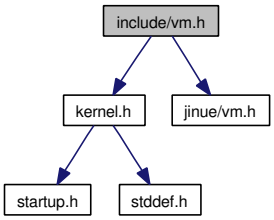
Definition at line 20 of file vm.h.

4.7 include/vm.h File Reference

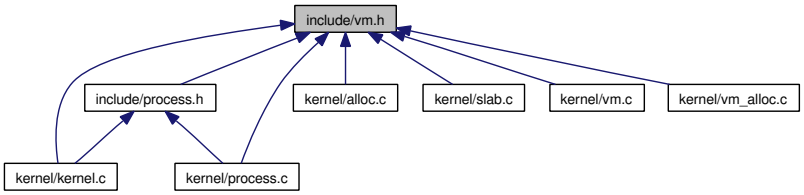
`#include <kernel.h>`

`#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** 0
  page is read only
- #define **VM_FLAG_READ_WRITE** (1<< 1)
  page is read/write accessible
- #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.

Referenced by kinit().

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(), kinit(), 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(), kinit(), 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.

Referenced by kinit().

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 94 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 100 of file vm.h.

4.7.1.17 #define VM_FLAG_CACHE_DISABLE (1<< 4)

uncached page

Definition at line 91 of file vm.h.

4.7.1.18 #define VM_FLAG_DIRTY (1<< 6)

page was written to

Definition at line 97 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 103 of file vm.h.

4.7.1.20 #define VM_FLAG_KERNEL 0

kernel mode page (default)

Definition at line 82 of file vm.h.

Referenced by kinit(), and 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 kinit(), slab_prepare(), and vm_map().

4.7.1.22 #define VM_FLAG_READ_ONLY 0

page is read only

Definition at line 76 of file vm.h.

4.7.1.23 #define VM_FLAG_READ_WRITE (1<< 1)

page is read/write accessible

Definition at line 79 of file vm.h.

Referenced by kinit(), and vm_map().

4.7.1.24 #define VM_FLAG_USER (1<< 2)

user mode page

Definition at line 85 of file vm.h.

Referenced by kinit(), and vm_map().

4.7.1.25 #define VM_FLAG_WRITE_THROUGH (1<< 3)

write-through cache policy for page

Definition at line 88 of file vm.h.

4.7.1.26 #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 106 of file vm.h.

Referenced by kinit(), and 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 14 of file vm.c.

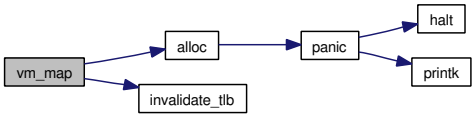
References alloc(), assert, invalidate_tlb(), PAGE_OFFSET_OF, PAGE_SIZE, PAGE_TABLE_ENTRIES, PAGE_TABLE_OF, PAGE_TABLE_PTE_OF, PDE_OF, PTE_OF, VM_FLAG_PRESENT, VM_FLAG_READ_WRITE, VM_FLAG_USER, and VM_FLAGS_PAGE_TABLE.

Referenced by vm_alloc(), and vm_vfree_block().

```
14                                                          {
15      pte_t *pte, *pde;
16      addr_t page_table;
17      int idx;
18
20      assert( PAGE_OFFSET_OF(vaddr) == 0 );
21
23      assert( PAGE_OFFSET_OF(paddr) == 0 );
24
25      /* get page directory entry */
26      pde = PDE_OF(vaddr);
27
28      /* check if page table must be created */
29      if( !(*pde & VM_FLAG_PRESENT) ) {
30          /* allocate a new page table */
31          page_table = alloc(PAGE_SIZE);
32
33          /* map page table in the region of memory reserved for that purpose */
34          pte = PAGE_TABLE_PTE_OF(vaddr);
35          *pte = (pte_t)page_table | VM_FLAGS_PAGE_TABLE | VM_FLAG_PRESENT;
36
37          /* obtain virtual address of new page table */
38          pte = PAGE_TABLE_OF(vaddr);
39
40          /* invalidate TLB entry for new page table */
41          invalidate_tlb( (addr_t)pte );
42
43          /* zero content of page table */
44          for(idx = 0; idx < PAGE_TABLE_ENTRIES; ++idx) {
45              pte[idx] = 0;
46          }
47
48          /* link to page table from page directory */
49          *pde = (pte_t)page_table | VM_FLAG_USER | VM_FLAG_READ_WRITE | VM_FLAG_PRESENT;
50      }
51
52      /* perform the actual mapping */
53      pte = PTE_OF(vaddr);
54      *pte = (pte_t)paddr | flags | VM_FLAG_PRESENT;
```

```
55
56     /* invalidate TLB entry for newly mapped page */
57     invalidate_tlb(vaddr);
58 }
```

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 64 of file vm.c.

References assert, invalidate_tlb(), NULL, PAGE_OFFSET_OF, and PTE_-OF.

Referenced by vm_free().

```
64                              {
65     pte_t *pte;
66
68     assert( PAGE_OFFSET_OF(addr) == 0 );
69
70     pte = PTE_OF(addr);
71     *pte = NULL;
72
73     /* TODO: is this really necessary? */
74     invalidate_tlb(addr);
75 }
```

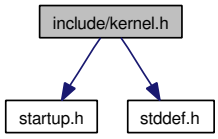
Here is the call graph for this function:

4.8 include/kernel.h File Reference

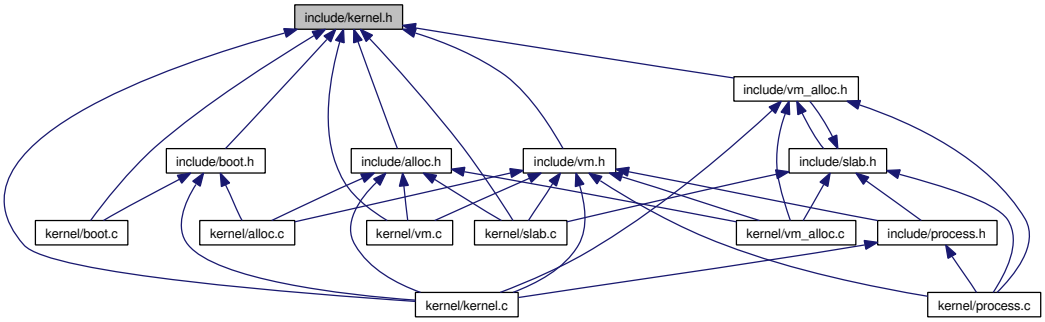
`#include <startup.h>`

`#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**

  address of top of kernel image (kernel_start + kernel_size)

- **addr_t kernel_region_top**

  top of region of memory mapped 1:1 (kernel image plus some pages for data structures allocated during initialization)

- **size_t kernel_size**

  size of the kernel image

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 252 of file kernel.c.

Referenced by kernel().

```
252                 {
253     while(1) {}
254 }
```

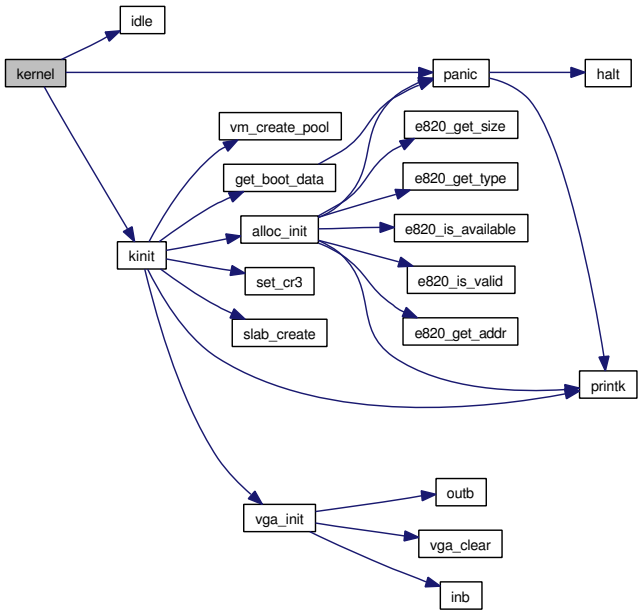
4.8.3.2 void kernel (void)

Definition at line 28 of file kernel.c.

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

```
28                  {
29     kinit();
30     idle();
31
32     panic("idle() returned.");
33 }
```

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

ASSERTION: we assume kernel_start is aligned with a page directory entry boundary

ASSERTION: we assume kernel_start is aligned with a page directory entry boundary

TODO: remove

TODO: remove

TODO: /remove

Definition at line 35 of file kernel.c.

References alloc_init(), assert, process_t::cr3, first_process, get_boot_data(), global_pool, global_pool_cache, kernel_region_top, kernel_size, kernel_start, KLIMIT, process_t::name, process_t::next, next_pid, NULL, PAGE_DIRECTORY_ADDR, PAGE_DIRECTORY_OFFSET_OF, page_directory_template, PAGE_MASK, PAGE_OFFSET_OF, PAGE_SIZE, PAGE_TABLE_ENTRIES, PAGE_TABLE_OFFSET_OF, PAGE_TABLES_ADDR, process_t::pid, printk(), PROCESS_NAME_LENGTH, process_slab_cache, set_cr3(), slab_create(), vga_init(), vm_create_pool(), VM_FLAG_KERNEL, VM_FLAG_PRESENT, VM_FLAG_READ_WRITE, VM_FLAG_USER, and VM_FLAGS_PAGE_TABLE.

Referenced by kernel().

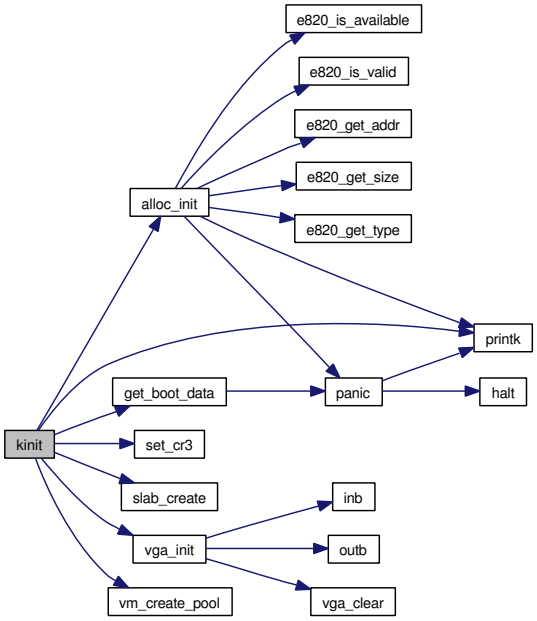
```
35                {
36     pte_t *page_table1, *page_table2, *page_directory;
37     pte_t *pte;
38     addr_t addr;
39     unsigned long idx, idy;
40     unsigned long temp;
41 
42     /* say hello */
43     vga_init();
44     printk("Kernel started.\n");
45 
47     assert( PAGE_OFFSET_OF( (unsigned int)kernel_start ) == 0 );
48 
50     assert( PAGE_TABLE_OFFSET_OF(PAGE_TABLES_ADDR) == 0 );
51     assert( PAGE_OFFSET_OF(PAGE_TABLES_ADDR) == 0 );
52 
54     assert( PAGE_TABLE_OFFSET_OF(PAGE_DIRECTORY_ADDR) == 0 );
55     assert( PAGE_OFFSET_OF(PAGE_DIRECTORY_ADDR) == 0 );
56 
57     printk("Kernel size is %u bytes.\n", kernel_size);
58 
60     printk("kernel_region_top on entry: 0x%x\n", (unsigned long)kernel_region_top );
61 
62     /* initialize data structures for caches and the global virtual page allocator */
63     slab_create(
64         &global_pool_cache,
65         &global_pool,
66         sizeof(vm_link_t),
67         VM_FLAG_KERNEL);
68 
69     vm_create_pool(&global_pool, &global_pool_cache);
70 
71     slab_create(&process_slab_cache, &global_pool,
72         sizeof(process_t), VM_FLAG_KERNEL);
73 
74     /* allocate one page for page directory template just after the
```

```
75        kernel image. Since paging is not yet activated, virtual and
76        physical address are the same. */
77     page_directory_template = (pte_t *)kernel_region_top;
78     kernel_region_top += PAGE_SIZE;
79
80     /* allocate page tables for kernel data/code region (0..KLIMIT) and add
81        relevant entries to page directory template */
82     for(idx = 0; idx < PAGE_DIRECTORY_OFFSET_OF(KLIMIT); ++idx) {
83         page_table1 = kernel_region_top;
84         kernel_region_top += PAGE_SIZE;
85
86         page_directory_template[idx] = (pte_t)page_table1 | VM_FLAG_PRESENT | VM_FLAG_KERNEL;
87
88         for(idy = 0; idy < PAGE_TABLE_ENTRIES; ++idy) {
89             page_table1[idy] = 0;
90         }
91     }
92
93     while(idx < PAGE_TABLE_ENTRIES) {
94         page_directory_template[idx] = 0;
95         ++idx;
96     }
97
98     /* allocate and fill content of a page directory and two page tables
99        for the creation of the address space of the first process (idle) */
100     page_directory = kernel_region_top;
101     kernel_region_top += PAGE_SIZE;
102
103     page_table1 = kernel_region_top;
104     kernel_region_top += PAGE_SIZE;
105
106     page_table2 = kernel_region_top;
107     kernel_region_top += PAGE_SIZE;
108
109     for(idx = 0; idx < PAGE_TABLE_ENTRIES; ++idx) {
110         temp = page_directory_template[idx];
111         page_directory[idx] = temp;
112         page_table1[idx]    = temp;
113     }
114
115     page_directory[PAGE_DIRECTORY_OFFSET_OF(PAGE_TABLES_ADDR)] =
116         (pte_t)page_table1 | VM_FLAG_PRESENT | VM_FLAG_USER | VM_FLAG_READ_WRITE;
117
118     page_directory[PAGE_DIRECTORY_OFFSET_OF(PAGE_DIRECTORY_ADDR)] =
119         (pte_t)page_table2 | VM_FLAG_PRESENT | VM_FLAG_USER | VM_FLAG_READ_WRITE;
120
121     page_table1[PAGE_DIRECTORY_OFFSET_OF(PAGE_TABLES_ADDR)] =
122         (pte_t)page_table1 | VM_FLAG_PRESENT | VM_FLAGS_PAGE_TABLE;
123
124     page_table1[PAGE_DIRECTORY_OFFSET_OF(PAGE_DIRECTORY_ADDR)] =
125         (pte_t)page_table2 | VM_FLAG_PRESENT | VM_FLAGS_PAGE_TABLE;
126
127     page_table2[0] = (pte_t)page_directory | VM_FLAG_PRESENT | VM_FLAGS_PAGE_TABLE;
128     for(idx = 1; idx < PAGE_TABLE_ENTRIES; ++idx) {
129         page_table2[idx] = 0;
130     }
131
```

```
132     /* create process descriptor for first process */
133     idle_process.pid = 0;
134     next_pid = 1;
135
136     idle_process.next = NULL;
137     first_process = &idle_process;
138
139     idle_process.cr3 = (addr_t)page_directory;
140
141     idle_process.name[0] = 'i';
142     idle_process.name[1] = 'd';
143     idle_process.name[2] = 'l';
144     idle_process.name[3] = 'e';
145     for(idx = 4; idx < PROCESS_NAME_LENGTH; ++idx) {
146         idle_process.name[idx] = 0;
147     }
148
149     /* perform 1:1 mapping of kernel image and data
150
151        note: page tables for memory region (0..KLIMIT) are contiguous
152              in memory */
153     page_table1 =
154         (pte_t *)page_directory[ PAGE_DIRECTORY_OFFSET_OF(kernel_start) ];
155     page_table1 = (pte_t *)( (unsigned int)page_table1 & ~PAGE_MASK  );
156
157     pte =
158         (pte_t *)&page_table1[ PAGE_TABLE_OFFSET_OF(kernel_start) ];
159
160     for(addr = kernel_start; addr < kernel_region_top; addr += PAGE_SIZE) {
161         *pte = (pte_t)addr | VM_FLAG_PRESENT | VM_FLAG_KERNEL;
162         ++pte;
163     }
164
166     printk("boot data: 0x%x\n", (unsigned long)get_boot_data() );
167
168     printk("page directory (0x%x):\n", (unsigned long)page_directory);
169     printk("  0x%x 0x%x 0x%x 0x%x 0x%x 0x%x 0x%x\n",
170         (unsigned long)page_directory[0],
171         (unsigned long)page_directory[1],
172         (unsigned long)page_directory[2],
173         (unsigned long)page_directory[3],
174         (unsigned long)page_directory[4],
175         (unsigned long)page_directory[5],
176         (unsigned long)page_directory[6]
177         );
178
179     if(PAGE_DIRECTORY_OFFSET_OF(kernel_start) != 0) {
180         printk("OOPS: PAGE_DIRECTORY_OFFSET_OF(kernel_start) != 0 (%u)\n",
181             PAGE_DIRECTORY_OFFSET_OF(kernel_start));
182     }
183
184     if(PAGE_TABLE_OFFSET_OF(kernel_start) != 256) {
185         printk("PAGE_TABLE_OFFSET_OF(kernel_start) != 256 (%u)\n",
186             PAGE_TABLE_OFFSET_OF(kernel_start));
187     }
188
189     page_table1 =
```

```
190         (pte_t *)page_directory[0];
191     page_table1 = (pte_t *)( (unsigned int)page_table1 & ~PAGE_MASK  );
192     pte = (pte_t *)&page_table1[250];
193     printk("Page table 0 (0x%x) offset 250 (0x%x):\n", (unsigned long)page_table1, (unsigned long) pte);
194 
195     for(idx = 0; idx < 42; ++idx) {
196         if(idx % 7 == 0) {
197             printk("  0x%x ", (unsigned long)pte[idx]);
198         }
199         else if(idx % 7 == 6) {
200             printk("0x%x\n", (unsigned long)pte[idx]);
201         }
202         else {
203             printk("0x%x ", (unsigned long)pte[idx]);
204         }
205     }
206 
207     page_table1 =
208         (pte_t *)page_directory[4];
209     page_table1 = (pte_t *)( (unsigned int)page_table1 & ~PAGE_MASK  );
210     printk("page table 4 (0x%x):\n", (unsigned long)page_table1);
211     printk("  0x%x 0x%x 0x%x 0x%x 0x%x 0x%x 0x%x\n",
212         (unsigned long)page_table1[0],
213         (unsigned long)page_table1[1],
214         (unsigned long)page_table1[2],
215         (unsigned long)page_table1[3],
216         (unsigned long)page_table1[4],
217         (unsigned long)page_table1[5],
218         (unsigned long)page_table1[6]
219         );
220 
221     page_table1 =
222         (pte_t *)page_directory[5];
223     page_table1 = (pte_t *)( (unsigned int)page_table1 & ~PAGE_MASK  );
224     printk("page table 5 (0x%x):\n", (unsigned long)page_table1);
225     printk("  0x%x 0x%x 0x%x 0x%x 0x%x 0x%x 0x%x\n",
226         (unsigned long)page_table1[0],
227         (unsigned long)page_table1[1],
228         (unsigned long)page_table1[2],
229         (unsigned long)page_table1[3],
230         (unsigned long)page_table1[4],
231         (unsigned long)page_table1[5],
232         (unsigned long)page_table1[6]
233         );
234 
241     /* activate paging */
242     set_cr3( (unsigned long)page_directory );
243 
244     /*temp = get_cr0();
245     temp |= (1 << X86_FLAG_PG);
246     set_cr0x(temp);*/
247 
248     /* initialize page frame allocator */
249     alloc_init();
250 }
```

Here is the call graph for this function:



4.8.4 Variable Documentation

4.8.4.1 addr_t kernel_region_top

top of region of memory mapped 1:1 (kernel image plus some pages for data structures allocated during initialization)

Definition at line 22 of file kernel.c.

Referenced by alloc_init(), and kinit().

4.8.4.2 size_t kernel_size

size of the kernel image

Definition at line 15 of file kernel.c.

Referenced by kinit().

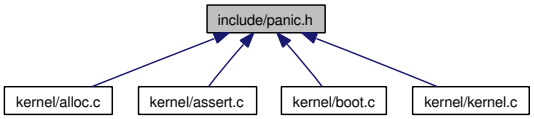
4.8.4.3 addr_t kernel_top

address of top of kernel image (kernel_start + kernel_size)

Definition at line 18 of file kernel.c.

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*)

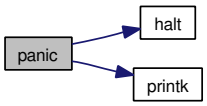
Definition at line 4 of file panic.c.

References halt(), and printk().

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

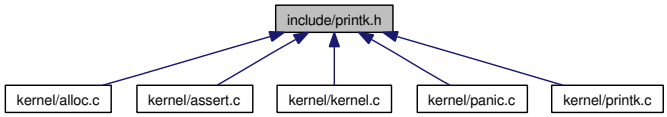
```
4                               {
5     printk("KERNEL PANIC: %s\n", message);
6     halt();
7 }
```

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 107 of file printk.c.

References print_hex_nibble().

```
107                                     {
108     print_hex_nibble( (char)byte );
109     print_hex_nibble( (char)(byte>>4) );
110 }
```

Here is the call graph for this function:

4.10.1.2 void print_hex_l (unsigned long *dword*)

Definition at line 120 of file printk.c.

References print_hex_nibble().

```
120                                     {
121     int off;
122
123     for(off=32-4; off>=0; off-=4) {
124         print_hex_nibble( (char)(dword>>off) );
125     }
126 }
```

Here is the call graph for this function:

4.10.1.3 void print_hex_nibble (unsigned char *byte*)

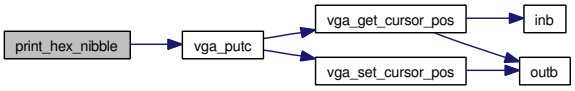
Definition at line 93 of file printk.c.

References vga_putc().

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

```
93                                          {
94     char c;
95
96     c = byte & 0xf;
97     if(c < 10) {
98         c += '0';
99     }
100     else {
101         c+= ('a' - 10);
102     }
103
104     vga_putc(c);
105 }
```

Here is the call graph for this function:



4.10.1.4 void print_hex_q (unsigned long long *qword*)

Definition at line 128 of file printk.c.

References print_hex_nibble().

```
128                                             {
129     int off;
130
131     for(off=64-4; off>=0; off-=4) {
132         print_hex_nibble( (char)(qword>>off) );
133     }
134 }
```

Here is the call graph for this function:

4.10.1.5 void print_hex_w (unsigned short *word*)

Definition at line 112 of file printk.c.

References print_hex_nibble().

```
112                                         {
113     int off;
114
115     for(off=16-4; off>=0; off-=4) {
116         print_hex_nibble( (char)(word>>off) );
117     }
118 }
```

Here is the call graph for this function:

4.10.1.6 void print_unsigned_int (unsigned int *n*)

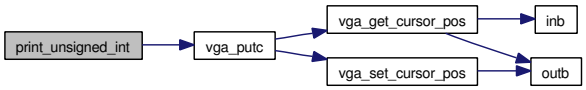
Definition at line 69 of file printk.c.

References vga_putc().

```
69                                          {
70     unsigned int flag = 0;
```

```
71     unsigned int pwr;
72     unsigned int digit;
73     char c;
74
75     if(n == 0) {
76         vga_putc('0');
77         return;
78     }
79
80     for(pwr = 1000 * 1000 * 1000; pwr > 0; pwr /= 10) {
81         digit = n / pwr;
82
83         if(digit != 0 || flag) {
84             c = (char)digit + '0';
85             vga_putc(c);
86
87             flag = 1;
88             n -= digit * pwr;
89         }
90     }
91 }
```

Here is the call graph for this function:



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

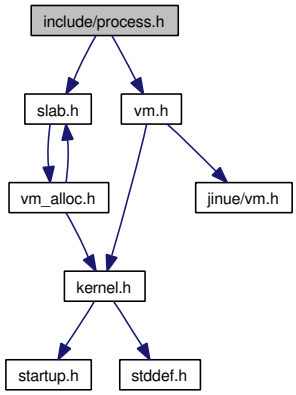
Referenced by __assert_failed(), alloc_init(), kinit(), and panic().

4.11 include/process.h File Reference

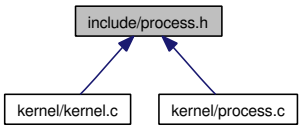
`#include <slab.h>`

`#include <vm.h>`

Include dependency graph for process.h:



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



Data Structures

- struct **process_t**
- struct **process_cb_t**
- struct **thread_t**

Defines

- #define **PROCESS_NAME_LENGTH** 16

Typedefs

- typedef unsigned long **pid_t**

- typedef struct **process_t process_t**
- typedef struct **process_cb_t process_cb_t**
- typedef struct **thread_t thread_t**

Functions

- **process_t** * **process_create** (void)
- void **process_destroy** (**process_t** *p)
- void **process_destroy_by_pid** (**pid_t** pid)
- **process_t** * **process_find_by_pid** (**pid_t** pid)

Variables

- **pid_t next_pid**

  PID for next process creation.

- **process_t** * **first_process**

  head of process descriptors linked list

- **slab_cache_t process_slab_cache**

  slab cache for allocation of process descriptors

- **pte_t** * **page_directory_template**

  template for the creation of a new page directory

4.11.1 Define Documentation

4.11.1.1 #define PROCESS_NAME_LENGTH 16

Definition at line 7 of file process.h.

Referenced by kinit().

4.11.2 Typedef Documentation

4.11.2.1 typedef unsigned long pid_t

Definition at line 9 of file process.h.

4.11.2.2 typedef struct process_cb_t process_cb_t

Definition at line 24 of file process.h.

4.11.2.3 typedef struct process_t process_t

Definition at line 18 of file process.h.

4.11.2.4 typedef struct thread_t thread_t

Definition at line 29 of file process.h.

4.11.3 Function Documentation

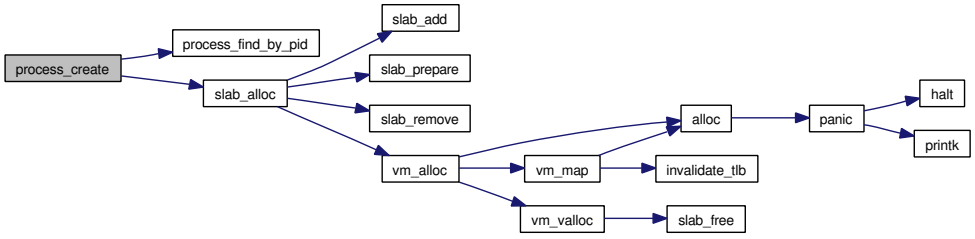
4.11.3.1 process_t* process_create (void)

Definition at line 20 of file process.c.

References next_pid, NULL, process_t::pid, process_find_by_pid(), and slab_alloc().

```
20                                {
21     process_t *p = slab_alloc(&process_slab_cache);
22
23     while( process_find_by_pid(next_pid) != NULL ) {
24         ++next_pid;
25     }
26
27     p->pid = next_pid++;
28
29     /* TODO: actual implementation */
30     return p;
31 }
```

Here is the call graph for this function:



4.11.3.2 void process_destroy (process_t * p)

Definition at line 33 of file process.c.

Referenced by process_destroy_by_pid().

```
33                                  {
34     /* TODO: actual implementation */
35 }
```

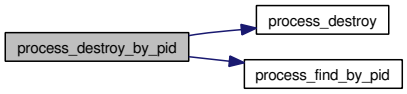
4.11.3.3 void process_destroy_by_pid (pid_t pid)

Definition at line 37 of file process.c.

References process_destroy(), and process_find_by_pid().

```
37                                          {
38     process_destroy( process_find_by_pid(pid) );
39 }
```

Here is the call graph for this function:



4.11.3.4 process_t* process_find_by_pid (pid_t pid)

Definition at line 41 of file process.c.

References process_t::next, NULL, and process_t::pid.

Referenced by process_create(), and process_destroy_by_pid().

```
41                                              {
42     process_t *p;
43
44     p = first_process;
45
46     while(p != NULL) {
47         if(p->pid == pid) {
48             return p;
49         }
50
51         p = p->next;
52     }
53
54     return NULL;
55 }
```

4.11.4 Variable Documentation

4.11.4.1 process_t* first_process

head of process descriptors linked list

Definition at line 11 of file process.c.

Referenced by kinit().

4.11.4.2 pid_t next_pid

PID for next process creation.

Definition at line 8 of file process.c.

Referenced by kinit(), and process_create().

4.11.4.3 pte_t* page_directory_template

template for the creation of a new page directory

Definition at line 17 of file process.c.

Referenced by kinit().

4.11.4.4 slab_cache_t process_slab_cache

slab cache for allocation of process descriptors

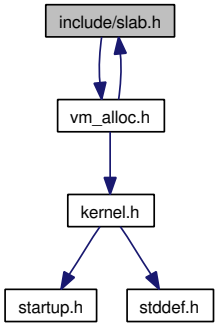
Definition at line 14 of file process.c.

Referenced by kinit().

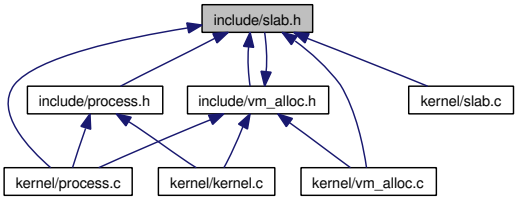
4.12 include/slab.h File Reference

`#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** (struct **slab_cache_t** *cache, struct **vm_alloc_t** *pool, **size_t** obj_size, 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.

4.12.1 Typedef Documentation

4.12.1.1 typedef struct slab_cache_t slab_cache_t

Definition at line 47 of file slab.h.

4.12.1.2 typedef struct slab_header_t slab_header_t

Definition at line 21 of file slab.h.

4.12.2 Function Documentation

4.12.2.1 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 136 of file slab.c.

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

Referenced by slab_alloc(), and vm_vfree_block().

```
136                                                         {
137     slab->next = *head;
138     slab->prev = NULL;
139
140     (*head)->prev = slab;
141     *head = slab;
142 }
```

4.12.2.2 addr_t slab_alloc (slab_cache_t * *cache*)

TODO: handle the NULL pointer

Definition at line 27 of file slab.c.

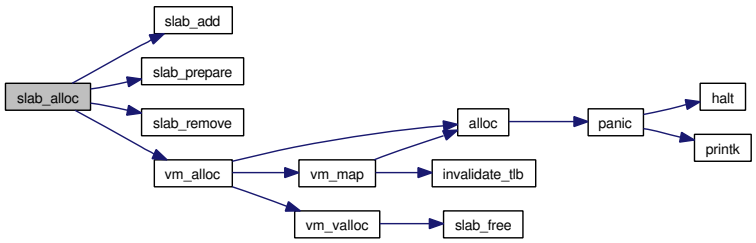
References slab_header_t::available, slab_cache_t::empty, slab_header_t::free_list, slab_cache_t::full, NULL, slab_cache_t::partial, slab_add(), slab_prepare(), slab_remove(), vm_alloc(), slab_cache_t::vm_allocator, and slab_cache_t::vm_flags.

Referenced by process_create(), and vm_vfree_block().

```
27                                          {
28     slab_header_t *slab;
29     addr_t addr;
30
31     /* use a partial slab if one is available... */
32     slab = cache->partial;
33     if(slab != NULL) {
34         addr = slab->free_list;
35         slab->free_list = *(addr_t *)addr;
36
37         /* maybe the slab is now full */
38         if(--slab->available == 0) {
39             slab_remove(&cache->partial, slab);
40             slab_add(&cache->full, slab);
41         }
42
43         return addr;
44     }
45
46     /* ... otherwise, use an empty slab ... */
47     slab = cache->empty;
48     if(slab != NULL) {
49         /* the slab is no longer empty */
50         slab_remove(&cache->empty, slab);
51         slab_add(&cache->partial, slab);
52
53         addr = slab->free_list;
54         slab->free_list = *(addr_t *)addr;
55
56         /* maybe the slab is now full */
57         if(--slab->available == 0) {
58             slab_remove(&cache->partial, slab);
59             slab_add(&cache->full, slab);
```

```
60          }
61
62          return addr;
63      }
64
65      /* ... and, as last resort, allocate a slab */
67      slab = (slab_header_t *)vm_alloc(cache->vm_allocator, cache->vm_flags);
68      slab_prepare(cache, (addr_t)slab);
69
70      /* this slab is not empty since we are allocating an object from it */
71      slab_add(&cache->partial, slab);
72
73      addr = slab->free_list;
74      slab->free_list = *(addr_t *)addr;
75
76      /* maybe the slab is now full */
77      if(--slab->available == 0) {
78          slab_remove(&cache->partial, slab);
79          slab_add(&cache->full, slab);
80      }
81
82      return addr;
83 }
```

Here is the call graph for this function:



4.12.2.3 void slab_create (struct slab_cache_t * *cache*, struct vm_alloc_t * *pool*, size_t *obj_size*, unsigned long *flags*)

Definition at line 7 of file slab.c.

References slab_cache_t::empty, slab_cache_t::full, NULL, slab_cache_t::obj_size, PAGE_SIZE, slab_cache_t::partial, slab_cache_t::per_slab, slab_cache_t::vm_allocator, and slab_cache_t::vm_flags.

Referenced by kinit().

```
11                          {
12
13
```

```
14     cache->obj_size = obj_size;
15     cache->per_slab = ( PAGE_SIZE - sizeof(slab_header_t) ) / obj_size;
16     cache->empty = NULL;
17     cache->partial = NULL;
18     cache->full = NULL;
19     cache->vm_flags = flags;
20     cache->vm_allocator = pool;
21 }
```

4.12.2.4 void slab_destroy (slab_cache_t * *cache*)

Definition at line 23 of file slab.c.

```
23                                      {
24     /* TODO: implement slab_destroy */
25 }
```

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

Definition at line 85 of file slab.c.

Referenced by vm_valloc().

```
85                                                  {
86 }
```

4.12.2.6 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 93 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().

```
93                                                  {
94     unsigned int cx;
95     size_t obj_size;
96     count_t per_slab;
97     slab_header_t *slab;
98     addr_t *ptr;
99     addr_t next;
100
102     assert( PAGE_OFFSET_OF(page) == 0 );
103
105     assert( cache->per_slab > 0 );
106
108     assert( (*PDE_OF(page) & ~PAGE_MASK) != NULL &&  (*PDE_OF(page) & VM_FLAG_PRESENT) != 0 );
109     assert( (*PTE_OF(page) & ~PAGE_MASK) != NULL &&  (*PTE_OF(page) & VM_FLAG_PRESENT) != 0 );
110
111     obj_size = cache->obj_size;
112     per_slab = cache->per_slab;
113
114     /* initialize slab header */
115     slab = (slab_header_t *)page;
116     slab->available = per_slab;
117     slab->free_list = page + sizeof(slab_header_t);
118
119     /* create free list */
120     ptr = (addr_t *)slab->free_list;
121
122     for(cx = 0; cx < per_slab - 1; ++cx) {
123         next = ptr + obj_size;
124         *ptr = next;
125         ptr = (addr_t *)next;
126     }
127
128     *ptr = NULL;
129 }
```

4.12.2.7 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 149 of file slab.c.

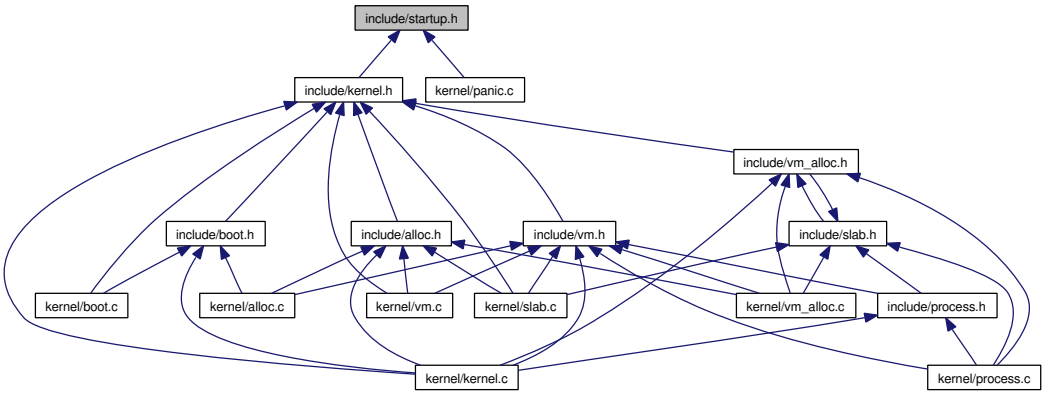
References slab_header_t::next, NULL, and slab_header_t::prev.

Referenced by slab_alloc().

```
149                                                                 {
150     if(slab->next != NULL) {
151         slab->next->prev = slab->prev;
152     }
153
154     if(slab->prev != NULL) {
155         slab->prev->next = slab->next;
156     }
157     else {
158         *head = slab->next;
159     }
160 }
```

4.13 include/startup.h File Reference

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



Defines

- #define **KERNEL_STACK_SIZE** 8192

Functions

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

4.13.1 Define Documentation

4.13.1.1 #define KERNEL_STACK_SIZE 8192

Definition at line 4 of file startup.h.

4.13.2 Function Documentation

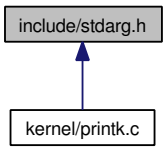
4.13.2.1 void halt (void)

Referenced by panic().

4.13.2.2 void start (void)

4.14 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**

4.14.1 Define Documentation

4.14.1.1 #define va_arg __builtin_va_arg

Definition at line 7 of file stdarg.h.

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

Definition at line 9 of file stdarg.h.

4.14.1.3 #define va_end __builtin_va_end

Definition at line 8 of file stdarg.h.

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

Definition at line 6 of file stdarg.h.

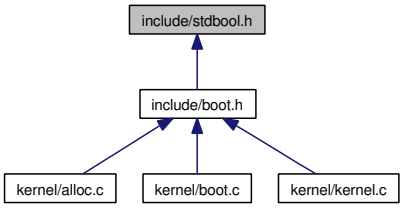
4.14.2 Typedef Documentation

4.14.2.1 typedef __builtin_va_list va_list

Definition at line 4 of file stdarg.h.

4.15 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

4.15.1 Define Documentation

4.15.1.1 #define __bool_true_false_are_defined 1

Definition at line 8 of file stdbool.h.

4.15.1.2 #define bool _Bool

Definition at line 4 of file stdbool.h.

4.15.1.3 #define false 0

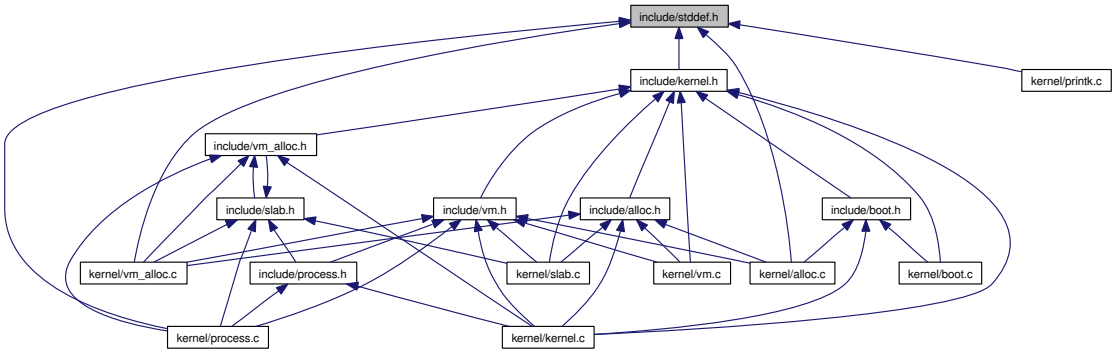
Definition at line 6 of file stdbool.h.

4.15.1.4 #define true 1

Definition at line 5 of file stdbool.h.

4.16 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**

4.16.1 Define Documentation

4.16.1.1 #define NULL 0

Definition at line 9 of file stddef.h.

Referenced by kinit(), process_create(), process_find_by_pid(), slab_add(), slab_alloc(), slab_create(), slab_prepare(), slab_remove(), vm_create_pool(), vm_free(), vm_unmap(), vm_valloc(), and vm_vfree_block().

4.16.1.2 #define offsetof(type, member) ((size_t) &(((type *)0) → member))

Definition at line 12 of file stddef.h.

4.16.2 Typedef Documentation

4.16.2.1 typedef signed long ptrdiff_t

Definition at line 4 of file stddef.h.

4.16.2.2 typedef unsigned long size_t

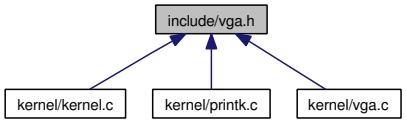
Definition at line 5 of file stddef.h.

4.16.2.3 typedef int wchar_t

Definition at line 6 of file stddef.h.

4.17 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)

4.17.1 Define Documentation

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

Definition at line 36 of file vga.h.

4.17.1.2 #define VGA_COLOR_BLACK 0x00

Definition at line 12 of file vga.h.

4.17.1.3 #define VGA_COLOR_BLUE 0x01

Definition at line 13 of file vga.h.

4.17.1.4 #define VGA_COLOR_BRIGHTBLUE 0x09

Definition at line 21 of file vga.h.

4.17.1.5 #define VGA_COLOR_BRIGHTCYAN 0x0b

Definition at line 23 of file vga.h.

4.17.1.6 #define VGA_COLOR_BRIGHTGREEN 0x0a

Definition at line 22 of file vga.h.

4.17.1.7 #define VGA_COLOR_BRIGHTMAGENTA 0x0d

Definition at line 25 of file vga.h.

4.17.1.8 #define VGA_COLOR_BRIGHTRED 0x0c

Definition at line 24 of file vga.h.

4.17.1.9 #define VGA_COLOR_BRIGHTWHITE 0x0f

Definition at line 27 of file vga.h.

4.17.1.10 #define VGA_COLOR_BROWN 0x06

Definition at line 18 of file vga.h.

4.17.1.11 #define VGA_COLOR_CYAN 0x03

Definition at line 15 of file vga.h.

4.17.1.12 #define VGA_COLOR_DEFAULT VGA_COLOR_GREEN

Definition at line 28 of file vga.h.

4.17.1.13 #define VGA_COLOR_ERASE VGA_COLOR_RED

Definition at line 29 of file vga.h.

Referenced by vga_clear(), and vga_scroll().

4.17.1.14 #define VGA_COLOR_GRAY 0x08

Definition at line 20 of file vga.h.

4.17.1.15 #define VGA_COLOR_GREEN 0x02

Definition at line 14 of file vga.h.

4.17.1.16 #define VGA_COLOR_MAGENTA 0x05

Definition at line 17 of file vga.h.

4.17.1.17 #define VGA_COLOR_RED 0x04

Definition at line 16 of file vga.h.

4.17.1.18 #define VGA_COLOR_WHITE 0x07

Definition at line 19 of file vga.h.

4.17.1.19 #define VGA_COLOR_YELLOW 0x0e

Definition at line 26 of file vga.h.

4.17.1.20 #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().

4.17.1.21 #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().

4.17.1.22 #define VGA_FB_FLAG_ACTIVE 1

Definition at line 10 of file vga.h.

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

Definition at line 35 of file vga.h.

4.17.1.24 #define VGA_LINES 25

Definition at line 31 of file vga.h.

Referenced by vga_clear(), and vga_scroll().

4.17.1.25 #define VGA_MISC_OUT_RD 0x3cc

Definition at line 6 of file vga.h.

Referenced by vga_init().

4.17.1.26 #define VGA_MISC_OUT_WR 0x3c2

Definition at line 5 of file vga.h.

Referenced by vga_init().

4.17.1.27 #define VGA_TAB_WIDTH 8

Definition at line 33 of file vga.h.

4.17.1.28 #define VGA_TEXT_VID_BASE 0xb8000

Definition at line 4 of file vga.h.

Referenced by vga_clear(), and vga_scroll().

4.17.1.29 #define VGA_WIDTH 80

Definition at line 32 of file vga.h.

Referenced by vga_clear(), and vga_scroll().

4.17.2 Typedef Documentation

4.17.2.1 typedef unsigned int vga_pos_t

Definition at line 38 of file vga.h.

4.17.3 Function Documentation

4.17.3.1 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 }
```

4.17.3.2 vga_pos_t vga_get_cursor_pos (void)

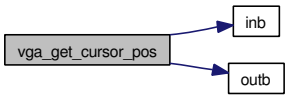
Definition at line 50 of file vga.c.

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:



4.17.3.3 void vga_init (void)

Definition at line 7 of file vga.c.

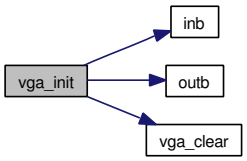
References inb(), outb(), vga_clear(), VGA_CRTC_ADDR, VGA_CRTC_-DATA, VGA_MISC_OUT_RD, and VGA_MISC_OUT_WR.

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:



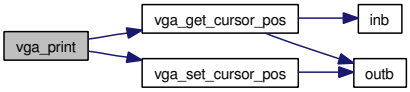
4.17.3.4 void vga_print (const char * *message*)

Definition at line 72 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

```
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:



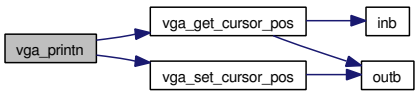
4.17.3.5 void vga_printn (const char * *message*, unsigned int *n*)

Definition at line 83 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

```
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:



4.17.3.6 void vga_putc (char *c*)

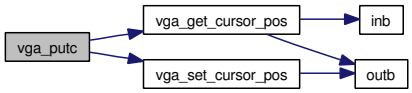
Definition at line 96 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by print_hex_nibble(), and print_unsigned_int().

```
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:



4.17.3.7 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 }
```

4.17.3.8 void vga_set_cursor_pos (vga_pos_t *pos*)

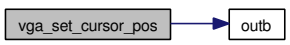
Definition at line 61 of file vga.c.

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:

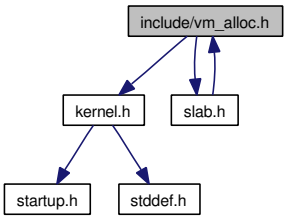


4.18 include/vm_alloc.h File Reference

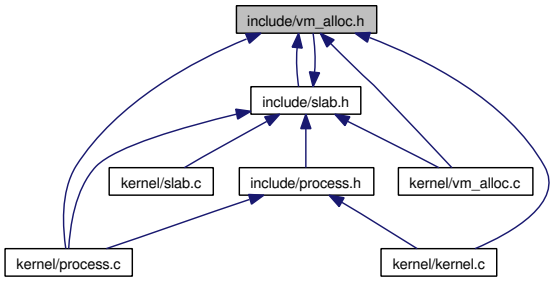
`#include <kernel.h>`

`#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

- void **vm_create_pool** (**vm_alloc_t** *pool, struct **slab_cache_t** *cache)
- **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. 46)).*

Variables

- struct **vm_alloc_t global_pool**
- struct **slab_cache_t global_pool_cache**

4.18.1 Typedef Documentation

4.18.1.1 typedef struct vm_alloc_t vm_alloc_t

Definition at line 34 of file vm_alloc.h.

4.18.1.2 typedef struct vm_link_t vm_link_t

Definition at line 19 of file vm_alloc.h.

4.18.2 Function Documentation

4.18.2.1 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. 46))

TODO: handle the NULL pointer

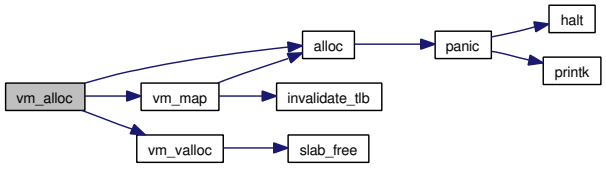
Definition at line 146 of file vm_alloc.c.

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

Referenced by slab_alloc().

```
146                                                     {
147     addr_t paddr, vaddr;
148
151     vaddr = vm_valloc(pool);
152     paddr = alloc(PAGE_SIZE);
153     vm_map(vaddr, paddr, flags);
154
155     return vaddr;
156 }
```

Here is the call graph for this function:



4.18.2.2 void vm_create_pool (vm_alloc_t * *pool*, struct slab_cache_t * *cache*)

Definition at line 13 of file vm_alloc.c.

References vm_alloc_t::cache, vm_alloc_t::head, NULL, and vm_alloc_t::size.

Referenced by kinit().

```
13                                                      {
14     pool->size = 0;
15     pool->head = NULL;
16     pool->cache = cache;
17 }
```

4.18.2.3 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. 46)).

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

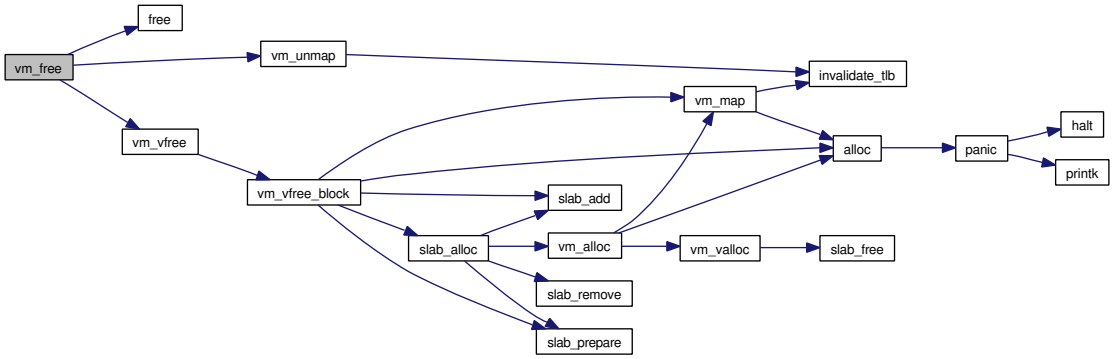
ASSERTION: address of page should not be the null pointer

Definition at line 165 of file vm_alloc.c.

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

```
165                                             {
166     addr_t paddr;
167
169     assert( addr != (addr_t)NULL );
170
171     paddr = (addr_t)(*PTE_OF(addr) | ~PAGE_MASK);
172
173     vm_unmap(addr);
174     vm_vfree(pool, addr);
175     free(paddr);
176 }
```

Here is the call graph for this function:



4.18.2.4 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. 17) 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 28 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, and slab_free().

Referenced by vm_alloc().

```
28                                      {
29     addr_t addr;
30     vm_link_t *head;
31     size_t size;
```

```
32
33     head = pool->head;
34
35     /* no page available */
36     if(head == (addr_t)NULL) {
37         return (addr_t)NULL;
38     }
39
40     addr = head->addr;
41     size = head->size - PAGE_SIZE;
42
44     assert( PAGE_OFFSET_OF(size) == 0 );
45
46     /* if block is made of only one page, we remove it from the free list */
47     if(size == 0) {
48         pool->head = head->next;
49         slab_free(pool->cache, head);
50     }
51     else {
52         head->size = size;
53         head->addr += PAGE_SIZE;
54     }
55
57     assert( PAGE_OFFSET_OF(addr) == 0 );
58
59     return addr;
60 }
```

Here is the call graph for this function:

4.18.2.5 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. 94) (and not **vm_alloc()** (p. 92)).

Parameters:

pool data structure managing the relevant virtual memory region

addr address of virtual page which must be freed

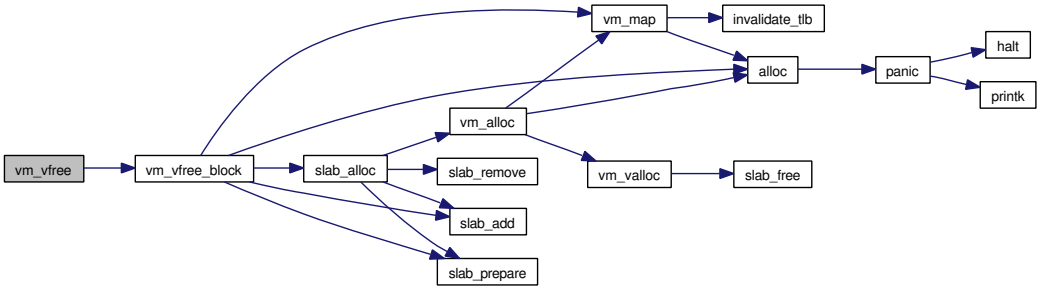
Definition at line 70 of file vm_alloc.c.

References PAGE_SIZE, and vm_vfree_block().

Referenced by vm_free().

```
70                                              {
71     vm_vfree_block(pool, addr, PAGE_SIZE);
72 }
```

Here is the call graph for this function:



4.18.2.6 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

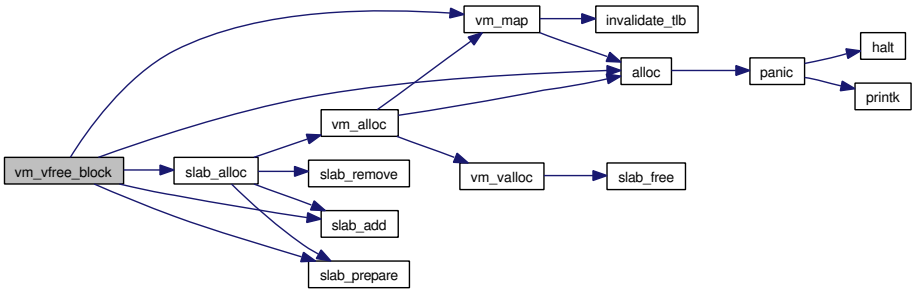
Definition at line 82 of file vm_alloc.c.

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().

```
82                                                                  {
83      addr_t phys_page;
84      vm_link_t *link;
85
87      assert( PAGE_OFFSET_OF(addr) == 0 );
88
90      assert( PAGE_OFFSET_OF(size) == 0 );
91
93      assert( addr != (addr_t)NULL );
94
95      /*  The virtual address space allocator needs a slab cache from which to
96          allocate data structures for its free list. Also, each slab cache needs
97          a virtual address space allocator to allocate slabs when needed.
98
99          There can be a mutual dependency between the virtual address space
100          allocator and the slab cache. This is not a problem in general, but a
101          special bootstrapping procedure is needed for initialization of the
102          virtual address space allocator in that case. The virtual address space
103          allocator will actually "donate" a virtual page (backed by physical ram)
104          to the cache for use as a slab.
105
106         This case is handled here
107       */
108      if(pool->head == NULL) {
109          if(pool->cache->vm_allocator == pool) {
110              if(pool->cache->empty == NULL && pool->cache->partial == NULL) {
111                  /* allocate a physical page for slab */
112                  phys_page = alloc(PAGE_SIZE);
113
114                  /* map page */
115                  vm_map(addr, phys_page, VM_FLAG_KERNEL);
116
117                  /* prepare the slab and add it to cache empty list */
118                  slab_prepare(pool->cache, addr);
119                  slab_add(&pool->cache->empty, addr);
120
121                  size -= PAGE_SIZE;
122
123                  /* if the block contained only one page, we have nothing left
124                     to free */
125                  if(size == 0) {
126                      return;
127                  }
128
129                  addr += PAGE_SIZE;
130              }
131          }
132      }
133
134      link = (vm_link_t *)slab_alloc(pool->cache);
135      link->size = size;
136      link->addr = addr;
137
138      link->next = pool->head;
139      pool->head = link;
140  }
```

Here is the call graph for this function:



4.18.3 Variable Documentation

4.18.3.1 struct vm_alloc_t global_pool

Definition at line 8 of file vm_alloc.c.

Referenced by kinit().

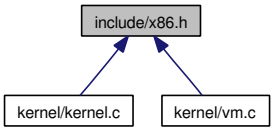
4.18.3.2 struct slab_cache_t global_pool_cache

Definition at line 10 of file vm_alloc.c.

Referenced by kinit().

4.19 include/x86.h File Reference

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



Defines

- #define **X86_FLAG_PG** 31

Functions

- void **invalidate_tlb** (**addr_t** vaddr)
- unsigned long **get_cr0** (void)
- unsigned long **get_cr1** (void)
- unsigned long **get_cr2** (void)
- unsigned long **get_cr3** (void)
- unsigned long **get_cr4** (void)
- void **set_cr0** (unsigned long val)
- void **set_cr0x** (unsigned long val)
- void **set_cr1** (unsigned long val)
- void **set_cr2** (unsigned long val)
- void **set_cr3** (unsigned long val)
- void **set_cr4** (unsigned long val)

4.19.1 Define Documentation

4.19.1.1 #define X86_FLAG_PG 31

Definition at line 4 of file x86.h.

4.19.2 Function Documentation

4.19.2.1 unsigned long get_cr0 (void)

4.19.2.2 unsigned long get_cr1 (void)

4.19.2.3 unsigned long get_cr2 (void)

4.19.2.4 unsigned long get_cr3 (void)

4.19.2.5 unsigned long get_cr4 (void)

4.19.2.6 void invalidate_tlb (addr_t *vaddr*)

Referenced by vm_map(), and vm_unmap().

4.19.2.7 void set_cr0 (unsigned long *val*)

4.19.2.8 void set_cr0x (unsigned long *val*)

4.19.2.9 void set_cr1 (unsigned long *val*)

4.19.2.10 void set_cr2 (unsigned long *val*)

4.19.2.11 void set_cr3 (unsigned long *val*)

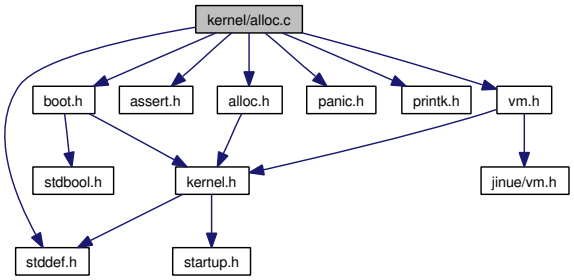
Referenced by kinit().

4.19.2.12 void set_cr4 (unsigned long *val*)

4.20 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)

4.20.1 Function Documentation

4.20.1.1 addr_t alloc (size_t *size*)

ASSERTION: returned address should be aligned with a page boundary

Definition at line 97 of file alloc.c.

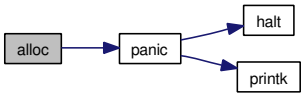
References assert, PAGE_BITS, PAGE_MASK, PAGE_SIZE, and panic().

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

```
97                          {
98     addr_t addr;
```

```
99      size_t pages;
100
101      pages = size >> PAGE_BITS;
102
103      if( (size & PAGE_MASK) != 0 ) {
104          ++pages;
105      }
106
107      if(_alloc_size < pages) {
108          panic("out of memory.");
109      }
110
111      addr = _alloc_addr;
112      _alloc_addr += pages * PAGE_SIZE;
113      _alloc_size -= pages;
114
116      assert( ((unsigned long)addr & PAGE_MASK) == 0 );
117
118      return addr;
119  }
```

Here is the call graph for this function:



4.20.1.2 void alloc_init (void)

Definition at line 13 of file alloc.c.

References e820_get_addr(), e820_get_size(), e820_get_type(), e820_is_available(), e820_is_valid(), kernel_region_top, kernel_start, PAGE_SIZE, panic(), and printk().

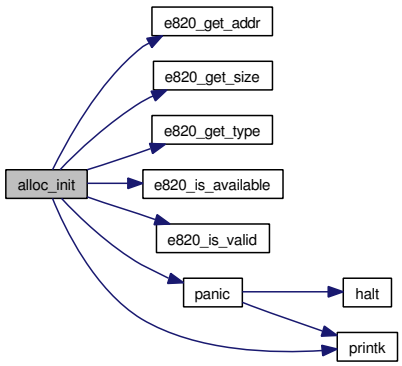
Referenced by kinit().

```
13                      {
14      unsigned int idx;
15      unsigned int remainder;
16      bool avail;
17      size_t size;
18      e820_type_t type;
19      addr_t addr, fixed_addr, best_addr;
20      size_t fixed_size, best_size;
21
22      idx = 0;
23      best_size = 0;
24
25      /*printk("Dump of the BIOS memory map:\n");
```

```
26      printk("  address  size     type\n");*/
27      while( e820_is_valid(idx) ) {
28          addr = e820_get_addr(idx);
29          size = e820_get_size(idx);
30          type = e820_get_type(idx);
31          avail = e820_is_available(idx);
32
33          ++idx;
34
35          /*printk("(%x) %c %x %x %s\n",
36              avail?'*':' ',
37              addr,
38              size,
39              e820_type_description(type) );*/
40
41          if( !avail ) {
42              continue;
43          }
44
45          fixed_addr = addr;
46          fixed_size = size;
47
48          /* is the region completely under the kernel ? */
49          if(addr + size > kernel_start) {
50              /* is the region completely above the kernel ? */
51              if(addr < kernel_region_top) {
52                  /* if the region touches the kernel, we take only
53                   * the part above the kernel, if there is one... */
54                  if(addr + size <= kernel_region_top) {
55                      /* ... and apparently, there is none */
56                      continue;
57                  }
58
59                  fixed_addr = kernel_region_top;
60                  fixed_size -= fixed_addr - addr;
61              }
62          }
63
64          /* we must make sure the starting address is aligned on a
65           * page boundary. The size will eventually be divided
66           * by the page size, and thus need not be aligned. */
67          remainder = (unsigned int)fixed_addr % PAGE_SIZE;
68          if(remainder != 0) {
69              remainder = PAGE_SIZE - remainder;
70              if(fixed_size < remainder) {
71                  continue;
72              }
73
74              fixed_addr += remainder;
75              fixed_size -= remainder;
76          }
77
78          if(fixed_size > best_size) {
79              best_addr = fixed_addr;
80              best_size = fixed_size;
81          }
82      }
```

```
83
84     _alloc_addr = (addr_t)best_addr;
85     _alloc_size = best_size / PAGE_SIZE;
86
87     if(_alloc_size == 0) {
88         panic("no memory to allocate.");
89     }
90
91     printk("%u kilobytes (%u pages) available starting at %xh.\n",
92         _alloc_size * PAGE_SIZE / 1024,
93         _alloc_size,
94         _alloc_addr );
95 }
```

Here is the call graph for this function:



4.20.1.3 void free (addr_t *addr*)

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

Definition at line 121 of file alloc.c.

References assert, and PAGE_OFFSET_OF.

Referenced by vm_free().

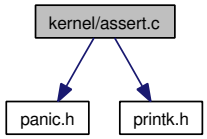
```
121                         {
123     assert( PAGE_OFFSET_OF(addr) == 0 );
124 }
```

4.21 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)

4.21.1 Function Documentation

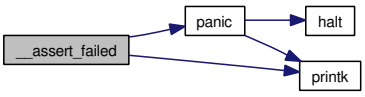
4.21.1.1 void __assert_failed (const char * *expr*, const char * *file*, unsigned int *line*, const char * *func*)

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  }
```

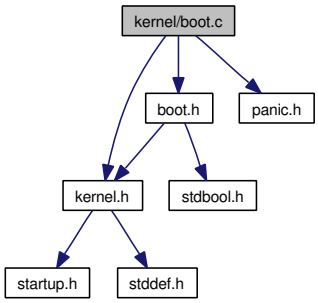
Here is the call graph for this function:



4.22 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_data**

4.22.1 Function Documentation

4.22.1.1 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 }
```

4.22.1.2 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 }
```

4.22.1.3 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 }
```

4.22.1.4 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 }
```

4.22.1.5 bool e820_is_valid (unsigned int *idx*)

Definition at line 20 of file boot.c.

Referenced by alloc_init().

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

4.22.1.6 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.

```
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 }
```

4.22.1.7 boot_t* get_boot_data (void)

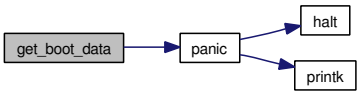
Definition at line 45 of file boot.c.

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

Referenced by kinit().

```
45                              {
46     boot_t *boot;
47
48     boot = (boot_t *)boot_data;
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:



4.22.2 Variable Documentation

4.22.2.1 addr_t boot_data

Definition at line 6 of file boot.c.

Referenced by get_boot_data().

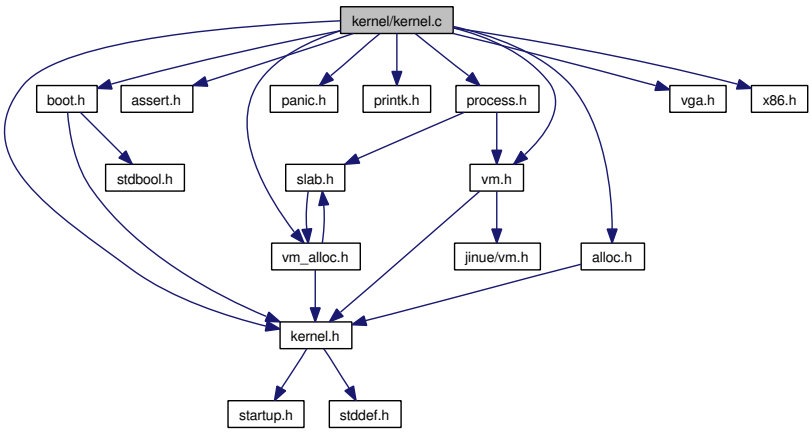
4.22.2.2 e820_t* e820_map

Definition at line 5 of file boot.c.

4.23 kernel/kernel.c File Reference

```
#include <alloc.h>
#include <assert.h>
#include <boot.h>
#include <kernel.h>
#include <panic.h>
#include <printk.h>
#include <process.h>
#include <vga.h>
#include <vm.h>
#include <vm_alloc.h>
#include <x86.h>
```

Include dependency graph for kernel.c:



Functions

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

Variables

- **size_t kernel_size**

  size of the kernel image

- **addr_t kernel_top**

  address of top of kernel image (kernel_start + kernel_size)

- **addr_t kernel_region_top**

  top of region of memory mapped 1:1 (kernel image plus some pages for data structures allocated during initialization)

- **process_t idle_process**

  process descriptor for first process (idle)

4.23.1 Function Documentation

4.23.1.1 void idle (void)

Definition at line 252 of file kernel.c.

Referenced by kernel().

```
252                 {
253     while(1) {}
254 }
```

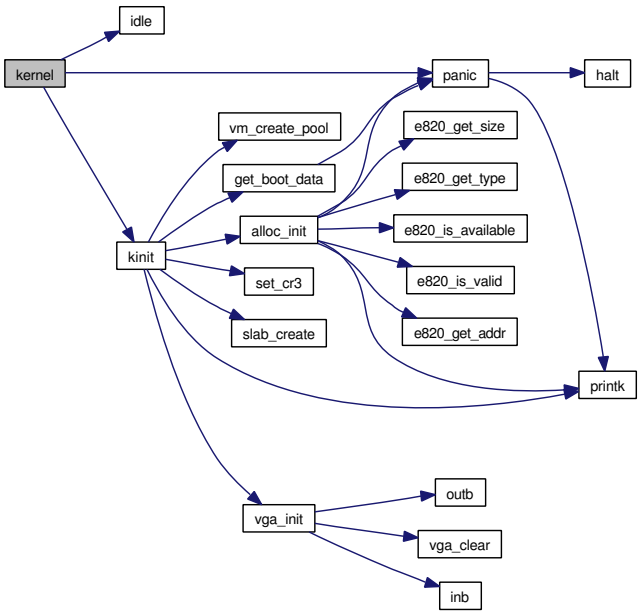
4.23.1.2 void kernel (void)

Definition at line 28 of file kernel.c.

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

```
28                  {
29     kinit();
30     idle();
31
32     panic("idle() returned.");
33 }
```

Here is the call graph for this function:



4.23.1.3 void kinit (void)

ASSERTION: we assume the kernel starts on a page boundary

ASSERTION: we assume kernel_start is aligned with a page directory entry boundary

ASSERTION: we assume kernel_start is aligned with a page directory entry boundary

TODO: remove

TODO: remove

TODO: /remove

Definition at line 35 of file kernel.c.

References alloc_init(), assert, process_t::cr3, first_process, get_boot_data(), global_pool, global_pool_cache, kernel_region_top, kernel_size, kernel_start, KLIMIT, process_t::name, process_t::next, next_pid, NULL, PAGE_DIRECTORY_ADDR, PAGE_DIRECTORY_OFFSET_OF, page_directory_template, PAGE_MASK, PAGE_OFFSET_OF, PAGE_SIZE, PAGE_TABLE_ENTRIES, PAGE_TABLE_OFFSET_OF, PAGE_TABLES_ADDR, process_t::pid, printk(), PROCESS_NAME_LENGTH,

process_slab_cache, set_cr3(), slab_create(), vga_init(), vm_create_pool(), VM_FLAG_KERNEL, VM_FLAG_PRESENT, VM_FLAG_READ_-WRITE, VM_FLAG_USER, and VM_FLAGS_PAGE_TABLE.

Referenced by kernel().

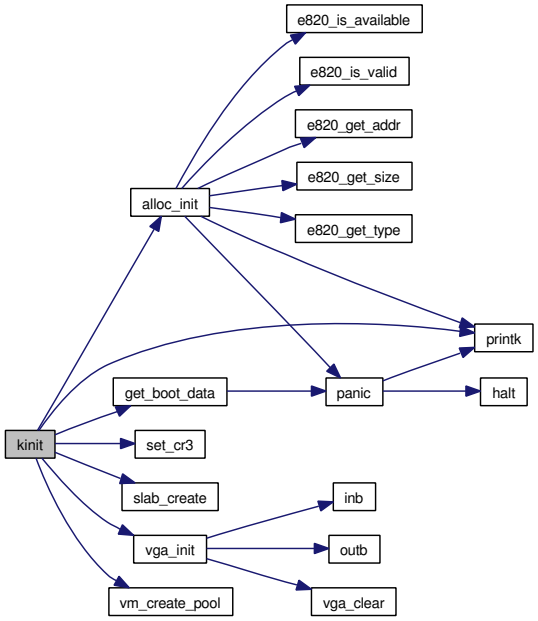
```
35                {
36     pte_t *page_table1, *page_table2, *page_directory;
37     pte_t *pte;
38     addr_t addr;
39     unsigned long idx, idy;
40     unsigned long temp;
41 
42     /* say hello */
43     vga_init();
44     printk("Kernel started.\n");
45 
47     assert( PAGE_OFFSET_OF( (unsigned int)kernel_start ) == 0 );
48 
50     assert( PAGE_TABLE_OFFSET_OF(PAGE_TABLES_ADDR) == 0 );
51     assert( PAGE_OFFSET_OF(PAGE_TABLES_ADDR) == 0 );
52 
54     assert( PAGE_TABLE_OFFSET_OF(PAGE_DIRECTORY_ADDR) == 0 );
55     assert( PAGE_OFFSET_OF(PAGE_DIRECTORY_ADDR) == 0 );
56 
57     printk("Kernel size is %u bytes.\n", kernel_size);
58 
60     printk("kernel_region_top on entry: 0x%x\n", (unsigned long)kernel_region_top );
61 
62     /* initialize data structures for caches and the global virtual page allocator */
63     slab_create(
64         &global_pool_cache,
65         &global_pool,
66         sizeof(vm_link_t),
67         VM_FLAG_KERNEL);
68 
69     vm_create_pool(&global_pool, &global_pool_cache);
70 
71     slab_create(&process_slab_cache, &global_pool,
72         sizeof(process_t), VM_FLAG_KERNEL);
73 
74     /* allocate one page for page directory template just after the
75        kernel image. Since paging is not yet activated, virtual and
76        physical address are the same. */
77     page_directory_template = (pte_t *)kernel_region_top;
78     kernel_region_top += PAGE_SIZE;
79 
80     /* allocate page tables for kernel data/code region (0..KLIMIT) and add
81        relevant entries to page directory template */
82     for(idx = 0; idx < PAGE_DIRECTORY_OFFSET_OF(KLIMIT); ++idx) {
83         page_table1 = kernel_region_top;
84         kernel_region_top += PAGE_SIZE;
85 
86         page_directory_template[idx] = (pte_t)page_table1 | VM_FLAG_PRESENT | VM_FLAG_KERNEL;
87 
88         for(idy = 0; idy < PAGE_TABLE_ENTRIES; ++idy) {
89             page_table1[idy] = 0;
```

```
90          }
91      }
92
93      while(idx < PAGE_TABLE_ENTRIES) {
94          page_directory_template[idx] = 0;
95          ++idx;
96      }
97
98      /* allocate and fill content of a page directory and two page tables
99         for the creation of the address space of the first process (idle) */
100     page_directory = kernel_region_top;
101     kernel_region_top += PAGE_SIZE;
102
103     page_table1 = kernel_region_top;
104     kernel_region_top += PAGE_SIZE;
105
106     page_table2 = kernel_region_top;
107     kernel_region_top += PAGE_SIZE;
108
109     for(idx = 0; idx < PAGE_TABLE_ENTRIES; ++idx) {
110         temp = page_directory_template[idx];
111         page_directory[idx] = temp;
112         page_table1[idx]    = temp;
113     }
114
115     page_directory[PAGE_DIRECTORY_OFFSET_OF(PAGE_TABLES_ADDR)] =
116         (pte_t)page_table1 | VM_FLAG_PRESENT | VM_FLAG_USER | VM_FLAG_READ_WRITE;
117
118     page_directory[PAGE_DIRECTORY_OFFSET_OF(PAGE_DIRECTORY_ADDR)] =
119         (pte_t)page_table2 | VM_FLAG_PRESENT | VM_FLAG_USER | VM_FLAG_READ_WRITE;
120
121     page_table1[PAGE_DIRECTORY_OFFSET_OF(PAGE_TABLES_ADDR)] =
122         (pte_t)page_table1 | VM_FLAG_PRESENT | VM_FLAGS_PAGE_TABLE;
123
124     page_table1[PAGE_DIRECTORY_OFFSET_OF(PAGE_DIRECTORY_ADDR)] =
125         (pte_t)page_table2 | VM_FLAG_PRESENT | VM_FLAGS_PAGE_TABLE;
126
127     page_table2[0] = (pte_t)page_directory | VM_FLAG_PRESENT | VM_FLAGS_PAGE_TABLE;
128     for(idx = 1; idx < PAGE_TABLE_ENTRIES; ++idx) {
129         page_table2[idx] = 0;
130     }
131
132     /* create process descriptor for first process */
133     idle_process.pid = 0;
134     next_pid = 1;
135
136     idle_process.next = NULL;
137     first_process = &idle_process;
138
139     idle_process.cr3 = (addr_t)page_directory;
140
141     idle_process.name[0] = 'i';
142     idle_process.name[1] = 'd';
143     idle_process.name[2] = 'l';
144     idle_process.name[3] = 'e';
145     for(idx = 4; idx < PROCESS_NAME_LENGTH; ++idx) {
146         idle_process.name[idx] = 0;
```

```
147     }
148
149     /* perform 1:1 mapping of kernel image and data
150
151        note: page tables for memory region (0..KLIMIT) are contiguous
152              in memory */
153     page_table1 =
154         (pte_t *)page_directory[ PAGE_DIRECTORY_OFFSET_OF(kernel_start) ];
155     page_table1 = (pte_t *)( (unsigned int)page_table1 & ~PAGE_MASK  );
156
157     pte =
158         (pte_t *)&page_table1[ PAGE_TABLE_OFFSET_OF(kernel_start) ];
159
160     for(addr = kernel_start; addr < kernel_region_top; addr += PAGE_SIZE) {
161         *pte = (pte_t)addr | VM_FLAG_PRESENT | VM_FLAG_KERNEL;
162         ++pte;
163     }
164
166     printk("boot data: 0x%x\n", (unsigned long)get_boot_data() );
167
168     printk("page directory (0x%x):\n", (unsigned long)page_directory);
169     printk("  0x%x 0x%x 0x%x 0x%x 0x%x 0x%x 0x%x\n",
170         (unsigned long)page_directory[0],
171         (unsigned long)page_directory[1],
172         (unsigned long)page_directory[2],
173         (unsigned long)page_directory[3],
174         (unsigned long)page_directory[4],
175         (unsigned long)page_directory[5],
176         (unsigned long)page_directory[6]
177         );
178
179     if(PAGE_DIRECTORY_OFFSET_OF(kernel_start) != 0) {
180         printk("OOPS: PAGE_DIRECTORY_OFFSET_OF(kernel_start) != 0 (%u)\n",
181             PAGE_DIRECTORY_OFFSET_OF(kernel_start));
182     }
183
184     if(PAGE_TABLE_OFFSET_OF(kernel_start) != 256) {
185         printk("PAGE_TABLE_OFFSET_OF(kernel_start) != 256 (%u)\n",
186             PAGE_TABLE_OFFSET_OF(kernel_start));
187     }
188
189     page_table1 =
190         (pte_t *)page_directory[0];
191     page_table1 = (pte_t *)( (unsigned int)page_table1 & ~PAGE_MASK  );
192     pte = (pte_t *)&page_table1[250];
193     printk("Page table 0 (0x%x) offset 250 (0x%x):\n", (unsigned long)page_table1, (unsigned long) pte);
194
195     for(idx = 0; idx < 42; ++idx) {
196         if(idx % 7 == 0) {
197             printk("  0x%x ", (unsigned long)pte[idx]);
198         }
199         else if(idx % 7 == 6) {
200             printk("0x%x\n", (unsigned long)pte[idx]);
201         }
202         else {
203             printk("0x%x ", (unsigned long)pte[idx]);
204         }
```

```
205     }
206
207     page_table1 =
208         (pte_t *)page_directory[4];
209     page_table1 = (pte_t *)( (unsigned int)page_table1 & ~PAGE_MASK  );
210     printk("page table 4 (0x%x):\n", (unsigned long)page_table1);
211     printk("  0x%x 0x%x 0x%x 0x%x 0x%x 0x%x 0x%x\n",
212         (unsigned long)page_table1[0],
213         (unsigned long)page_table1[1],
214         (unsigned long)page_table1[2],
215         (unsigned long)page_table1[3],
216         (unsigned long)page_table1[4],
217         (unsigned long)page_table1[5],
218         (unsigned long)page_table1[6]
219         );
220
221     page_table1 =
222         (pte_t *)page_directory[5];
223     page_table1 = (pte_t *)( (unsigned int)page_table1 & ~PAGE_MASK  );
224     printk("page table 5 (0x%x):\n", (unsigned long)page_table1);
225     printk("  0x%x 0x%x 0x%x 0x%x 0x%x 0x%x 0x%x\n",
226         (unsigned long)page_table1[0],
227         (unsigned long)page_table1[1],
228         (unsigned long)page_table1[2],
229         (unsigned long)page_table1[3],
230         (unsigned long)page_table1[4],
231         (unsigned long)page_table1[5],
232         (unsigned long)page_table1[6]
233         );
234
241     /* activate paging */
242     set_cr3( (unsigned long)page_directory );
243
244     /*temp = get_cr0();
245     temp |= (1 << X86_FLAG_PG);
246     set_cr0x(temp);*/
247
248     /* initialize page frame allocator */
249     alloc_init();
250 }
```

Here is the call graph for this function:



4.23.2 Variable Documentation

4.23.2.1 process_t idle_process

process descriptor for first process (idle)

Definition at line 25 of file kernel.c.

4.23.2.2 addr_t kernel_region_top

top of region of memory mapped 1:1 (kernel image plus some pages for data structures allocated during initialization)

Definition at line 22 of file kernel.c.

Referenced by alloc_init(), and kinit().

4.23.2.3 size_t kernel_size

size of the kernel image

Definition at line 15 of file kernel.c.

Referenced by kinit().

4.23.2.4 addr_t kernel_top

address of top of kernel image (kernel_start + kernel_size)

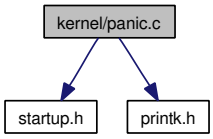
Definition at line 18 of file kernel.c.

4.24 kernel/panic.c File Reference

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

```
#include <printk.h>
```

Include dependency graph for panic.c:



Functions

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

4.24.1 Function Documentation

4.24.1.1 void panic (const char * *message*)

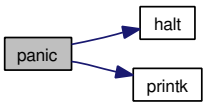
Definition at line 4 of file panic.c.

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 }
```

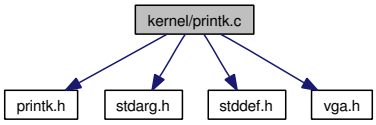
Here is the call graph for this function:



4.25 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 **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)

4.25.1 Function Documentation

4.25.1.1 void print_hex_b (unsigned char *byte*)

Definition at line 107 of file printk.c.

References print_hex_nibble().

```
107                                     {
108     print_hex_nibble( (char)byte );
109     print_hex_nibble( (char)(byte>>4) );
110 }
```

Here is the call graph for this function:

4.25.1.2 void print_hex_l (unsigned long *dword*)

Definition at line 120 of file printk.c.

References print_hex_nibble().

```
120                                     {
121     int off;
122
123     for(off=32-4; off>=0; off-=4) {
124         print_hex_nibble( (char)(dword>>off) );
125     }
126 }
```

Here is the call graph for this function:

4.25.1.3 void print_hex_nibble (unsigned char *byte*)

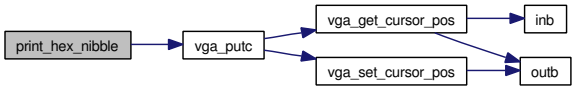
Definition at line 93 of file printk.c.

References vga_putc().

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

```
93                                      {
94     char c;
95
96     c = byte & 0xf;
97     if(c < 10) {
98         c += '0';
99     }
100     else {
101         c+= ('a' - 10);
102     }
103
104     vga_putc(c);
105 }
```

Here is the call graph for this function:



4.25.1.4 void print_hex_q (unsigned long long *qword*)

Definition at line 128 of file printk.c.

References print_hex_nibble().

```
128                                             {
129     int off;
130
131     for(off=64-4; off>=0; off-=4) {
132         print_hex_nibble( (char)(qword>>off) );
133     }
134 }
```

Here is the call graph for this function:

4.25.1.5 void print_hex_w (unsigned short *word*)

Definition at line 112 of file printk.c.

References print_hex_nibble().

```
112                                         {
113     int off;
114
115     for(off=16-4; off>=0; off-=4) {
116         print_hex_nibble( (char)(word>>off) );
117     }
118 }
```

Here is the call graph for this function:

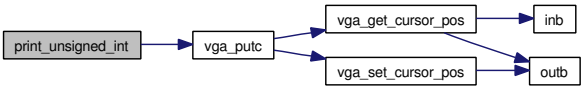
4.25.1.6 void print_unsigned_int (unsigned int *n*)

Definition at line 69 of file printk.c.

References vga_putc().

```
69                                          {
70      unsigned int flag = 0;
71      unsigned int pwr;
72      unsigned int digit;
73      char c;
74
75      if(n == 0) {
76          vga_putc('0');
77          return;
78      }
79
80      for(pwr = 1000 * 1000 * 1000; pwr > 0; pwr /= 10) {
81          digit = n / pwr;
82
83          if(digit != 0 || flag) {
84              c = (char)digit + '0';
85              vga_putc(c);
86
87              flag = 1;
88              n -= digit * pwr;
89          }
90      }
91 }
```

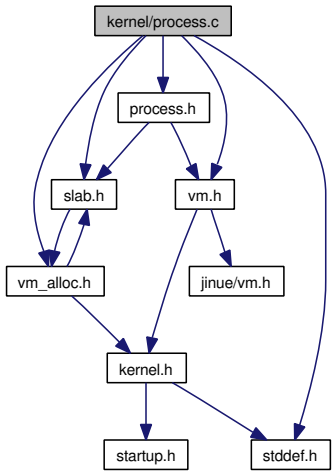
Here is the call graph for this function:



4.26 kernel/process.c File Reference

```
#include <process.h>
#include <slab.h>
#include <stddef.h>
#include <vm.h>
#include <vm_alloc.h>
```

Include dependency graph for process.c:



Functions

- **process_t** * **process_create** (void)
- void **process_destroy** (**process_t** *p)
- void **process_destroy_by_pid** (**pid_t** pid)
- **process_t** * **process_find_by_pid** (**pid_t** pid)

Variables

- **pid_t next_pid**

  PID for next process creation.

- **process_t** * **first_process**

  head of process descriptors linked list

- **slab_cache_t process_slab_cache**

  slab cache for allocation of process descriptors

- **pte_t * page_directory_template**

  template for the creation of a new page directory

4.26.1 Function Documentation

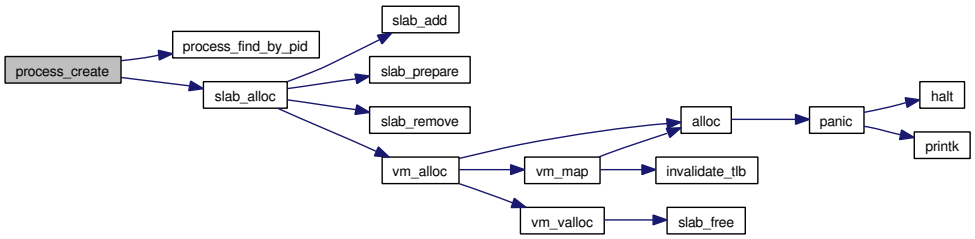
4.26.1.1 process_t* process_create (void)

Definition at line 20 of file process.c.

References next_pid, NULL, process_t::pid, process_find_by_pid(), and slab_alloc().

```
20                                {
21     process_t *p = slab_alloc(&process_slab_cache);
22
23     while( process_find_by_pid(next_pid) != NULL ) {
24         ++next_pid;
25     }
26
27     p->pid = next_pid++;
28
29     /* TODO: actual implementation */
30     return p;
31 }
```

Here is the call graph for this function:



4.26.1.2 void process_destroy (process_t * p)

Definition at line 33 of file process.c.

Referenced by process_destroy_by_pid().

```
33                                  {
34     /* TODO: actual implementation */
35 }
```

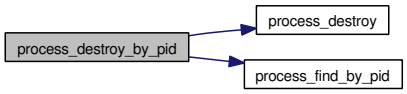
4.26.1.3 void process_destroy_by_pid (pid_t *pid*)

Definition at line 37 of file process.c.

References process_destroy(), and process_find_by_pid().

```
37                                      {
38     process_destroy( process_find_by_pid(pid) );
39 }
```

Here is the call graph for this function:



4.26.1.4 process_t* process_find_by_pid (pid_t *pid*)

Definition at line 41 of file process.c.

References process_t::next, NULL, and process_t::pid.

Referenced by process_create(), and process_destroy_by_pid().

```
41                                          {
42     process_t *p;
43
44     p = first_process;
45
46     while(p != NULL) {
47         if(p->pid == pid) {
48             return p;
49         }
50
51         p = p->next;
52     }
53
54     return NULL;
55 }
```

4.26.2 Variable Documentation

4.26.2.1 process_t* first_process

head of process descriptors linked list

Definition at line 11 of file process.c.

Referenced by kinit().

4.26.2.2 pid_t next_pid

PID for next process creation.

Definition at line 8 of file process.c.

Referenced by kinit(), and process_create().

4.26.2.3 pte_t* page_directory_template

template for the creation of a new page directory

Definition at line 17 of file process.c.

Referenced by kinit().

4.26.2.4 slab_cache_t process_slab_cache

slab cache for allocation of process descriptors

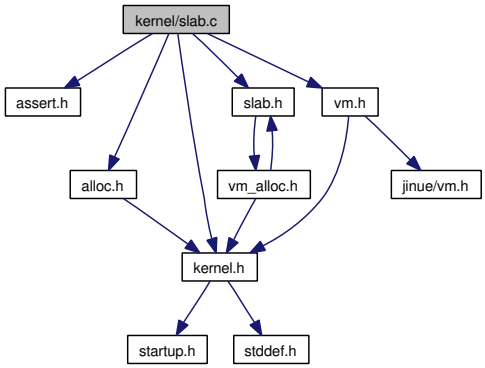
Definition at line 14 of file process.c.

Referenced by kinit().

4.27 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, **vm_alloc_t** *pool, **size_t** obj_size, 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.

4.27.1 Function Documentation

4.27.1.1 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 136 of file slab.c.

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

Referenced by slab_alloc(), and vm_vfree_block().

```
136                                                         {
137     slab->next = *head;
138     slab->prev = NULL;
139
140     (*head)->prev = slab;
141     *head = slab;
142 }
```

4.27.1.2 addr_t slab_alloc (slab_cache_t * *cache*)

TODO: handle the NULL pointer

Definition at line 27 of file slab.c.

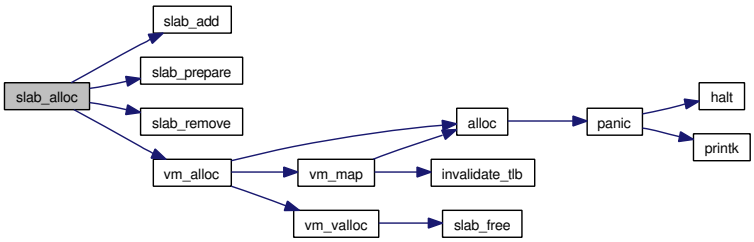
References slab_header_t::available, slab_cache_t::empty, slab_header_t::free_list, slab_cache_t::full, NULL, slab_cache_t::partial, slab_add(), slab_prepare(), slab_remove(), vm_alloc(), slab_cache_t::vm_allocator, and slab_cache_t::vm_flags.

Referenced by process_create(), and vm_vfree_block().

```
27                                          {
28     slab_header_t *slab;
29     addr_t addr;
30
31     /* use a partial slab if one is available... */
32     slab = cache->partial;
33     if(slab != NULL) {
34         addr = slab->free_list;
35         slab->free_list = *(addr_t *)addr;
36
```

```
37          /* maybe the slab is now full */
38          if(--slab->available == 0) {
39              slab_remove(&cache->partial, slab);
40              slab_add(&cache->full, slab);
41          }
42
43          return addr;
44      }
45
46      /* ... otherwise, use an empty slab ... */
47      slab = cache->empty;
48      if(slab != NULL) {
49          /* the slab is no longer empty */
50          slab_remove(&cache->empty, slab);
51          slab_add(&cache->partial, slab);
52
53          addr = slab->free_list;
54          slab->free_list = *(addr_t *)addr;
55
56          /* maybe the slab is now full */
57          if(--slab->available == 0) {
58              slab_remove(&cache->partial, slab);
59              slab_add(&cache->full, slab);
60          }
61
62          return addr;
63      }
64
65      /* ... and, as last resort, allocate a slab */
67      slab = (slab_header_t *)vm_alloc(cache->vm_allocator, cache->vm_flags);
68      slab_prepare(cache, (addr_t)slab);
69
70      /* this slab is not empty since we are allocating an object from it */
71      slab_add(&cache->partial, slab);
72
73      addr = slab->free_list;
74      slab->free_list = *(addr_t *)addr;
75
76      /* maybe the slab is now full */
77      if(--slab->available == 0) {
78          slab_remove(&cache->partial, slab);
79          slab_add(&cache->full, slab);
80      }
81
82      return addr;
83  }
```

Here is the call graph for this function:



4.27.1.3 void slab_create (slab_cache_t * *cache*, vm_alloc_t * *pool*, size_t *obj_size*, unsigned long *flags*)

Definition at line 7 of file slab.c.

References slab_cache_t::empty, slab_cache_t::full, NULL, slab_cache_t::obj_size, PAGE_SIZE, slab_cache_t::partial, slab_cache_t::per_slab, slab_cache_t::vm_allocator, and slab_cache_t::vm_flags.

Referenced by kinit().

```
11                          {
12
13
14      cache->obj_size = obj_size;
15      cache->per_slab = ( PAGE_SIZE - sizeof(slab_header_t) ) / obj_size;
16      cache->empty = NULL;
17      cache->partial = NULL;
18      cache->full = NULL;
19      cache->vm_flags = flags;
20      cache->vm_allocator = pool;
21  }
```

4.27.1.4 void slab_destroy (slab_cache_t * *cache*)

Definition at line 23 of file slab.c.

```
23                                      {
24      /* TODO: implement slab_destroy */
25  }
```

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

Definition at line 85 of file slab.c.

Referenced by vm_valloc().

```
85                                                  {
86 }
```

4.27.1.6 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 93 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().

```
93                                                      {
94     unsigned int cx;
95     size_t obj_size;
96     count_t per_slab;
97     slab_header_t *slab;
98     addr_t *ptr;
99     addr_t next;
100
102     assert( PAGE_OFFSET_OF(page) == 0 );
103
105     assert( cache->per_slab > 0 );
106
108     assert( (*PDE_OF(page) & ~PAGE_MASK) != NULL &&  (*PDE_OF(page) & VM_FLAG_PRESENT) != 0 );
109     assert( (*PTE_OF(page) & ~PAGE_MASK) != NULL &&  (*PTE_OF(page) & VM_FLAG_PRESENT) != 0 );
110
111     obj_size = cache->obj_size;
112     per_slab = cache->per_slab;
113
114     /* initialize slab header */
115     slab = (slab_header_t *)page;
116     slab->available = per_slab;
117     slab->free_list = page + sizeof(slab_header_t);
```

```
118
119     /* create free list */
120     ptr = (addr_t *)slab->free_list;
121
122     for(cx = 0; cx < per_slab - 1; ++cx) {
123         next = ptr + obj_size;
124         *ptr = next;
125         ptr = (addr_t *)next;
126     }
127
128     *ptr = NULL;
129 }
```

4.27.1.7 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 149 of file slab.c.

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

Referenced by slab_alloc().

```
149                                                         {
150     if(slab->next != NULL) {
151         slab->next->prev = slab->prev;
152     }
153
154     if(slab->prev != NULL) {
155         slab->prev->next = slab->next;
156     }
157     else {
158         *head = slab->next;
159     }
160 }
```

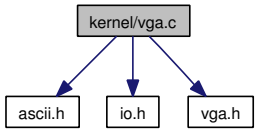
4.28 kernel/vga.c File Reference

```
#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)

4.28.1 Function Documentation

4.28.1.1 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 }
```

4.28.1.2 vga_pos_t vga_get_cursor_pos (void)

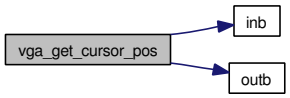
Definition at line 50 of file vga.c.

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:



4.28.1.3 void vga_init (void)

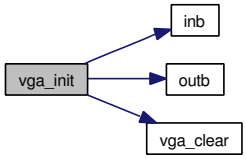
Definition at line 7 of file vga.c.

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:



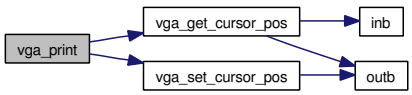
4.28.1.4 void vga_print (const char * *message*)

Definition at line 72 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

```
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:



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

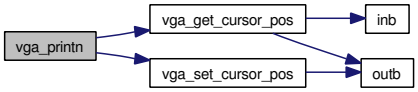
Definition at line 83 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

```
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:



4.28.1.6 void vga_putc (char *c*)

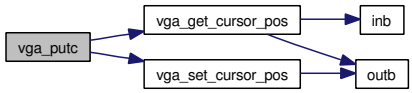
Definition at line 96 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by print_hex_nibble(), and print_unsigned_int().

```
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:



4.28.1.7 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 }
```

4.28.1.8 void vga_set_cursor_pos (vga_pos_t *pos*)

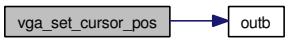
Definition at line 61 of file vga.c.

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 }
```

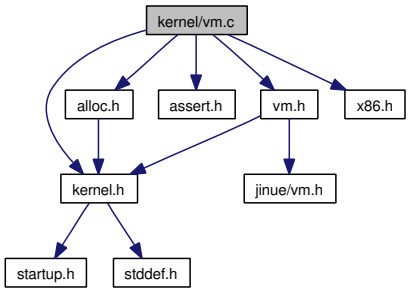
Here is the call graph for this function:



4.29 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.

4.29.1 Function Documentation

4.29.1.1 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

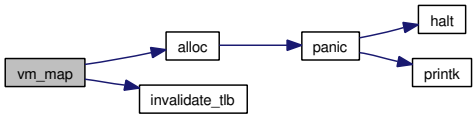
Definition at line 14 of file vm.c.

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_READ_WRITE, VM_FLAG_USER, and VM_FLAGS_PAGE_TABLE.

Referenced by vm_alloc(), and vm_vfree_block().

```
14                                                                  {
15     pte_t *pte, *pde;
16     addr_t page_table;
17     int idx;
18
20     assert( PAGE_OFFSET_OF(vaddr) == 0 );
21
23     assert( PAGE_OFFSET_OF(paddr) == 0 );
24
25     /* get page directory entry */
26     pde = PDE_OF(vaddr);
27
28     /* check if page table must be created */
29     if( !(*pde & VM_FLAG_PRESENT) ) {
30         /* allocate a new page table */
31         page_table = alloc(PAGE_SIZE);
32
33         /* map page table in the region of memory reserved for that purpose */
34         pte = PAGE_TABLE_PTE_OF(vaddr);
35         *pte = (pte_t)page_table | VM_FLAGS_PAGE_TABLE | VM_FLAG_PRESENT;
36
37         /* obtain virtual address of new page table */
38         pte = PAGE_TABLE_OF(vaddr);
39
40         /* invalidate TLB entry for new page table */
41         invalidate_tlb( (addr_t)pte );
42
43         /* zero content of page table */
44         for(idx = 0; idx < PAGE_TABLE_ENTRIES; ++idx) {
45             pte[idx] = 0;
46         }
47
48         /* link to page table from page directory */
49         *pde = (pte_t)page_table | VM_FLAG_USER | VM_FLAG_READ_WRITE | VM_FLAG_PRESENT;
50     }
51
52     /* perform the actual mapping */
53     pte = PTE_OF(vaddr);
54     *pte = (pte_t)paddr | flags | VM_FLAG_PRESENT;
55
56     /* invalidate TLB entry for newly mapped page */
57     invalidate_tlb(vaddr);
58 }
```

Here is the call graph for this function:



4.29.1.2 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 64 of file vm.c.

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

Referenced by vm_free().

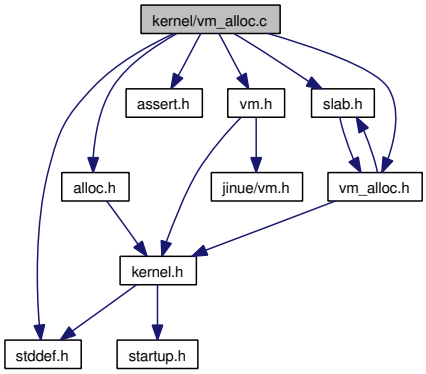
```
64                              {
65     pte_t *pte;
66
68     assert( PAGE_OFFSET_OF(addr) == 0 );
69
70     pte = PTE_OF(addr);
71     *pte = NULL;
72
73     /* TODO: is this really necessary? */
74     invalidate_tlb(addr);
75 }
```

Here is the call graph for this function:

4.30 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

- void **vm_create_pool** (**vm_alloc_t** *pool, **slab_cache_t** *cache)
- **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. 46)).*

Variables

- **vm_alloc_t global_pool**
- **slab_cache_t global_pool_cache**

4.30.1 Function Documentation

4.30.1.1 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. 46))

TODO: handle the NULL pointer

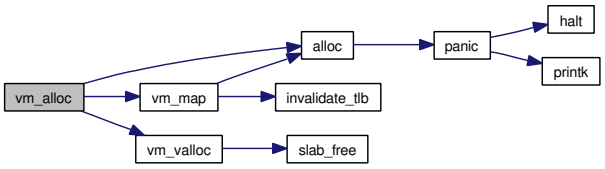
Definition at line 146 of file vm_alloc.c.

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

Referenced by slab_alloc().

```
146                                                 {
147     addr_t paddr, vaddr;
148
151     vaddr = vm_valloc(pool);
152     paddr = alloc(PAGE_SIZE);
153     vm_map(vaddr, paddr, flags);
154
155     return vaddr;
156 }
```

Here is the call graph for this function:



4.30.1.2 void vm_create_pool (vm_alloc_t * *pool*, slab_cache_t * *cache*)

Definition at line 13 of file vm_alloc.c.

References vm_alloc_t::cache, vm_alloc_t::head, NULL, and vm_alloc_t::size.

Referenced by kinit().

```
13                                                          {
14     pool->size = 0;
15     pool->head = NULL;
16     pool->cache = cache;
17 }
```

4.30.1.3 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. 46)).

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

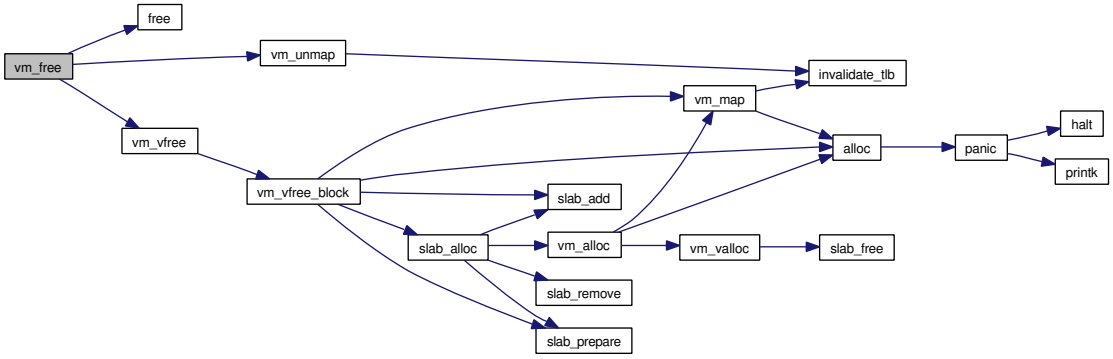
Definition at line 165 of file vm_alloc.c.

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

```
165                                         {
```

```
166     addr_t paddr;
167
169     assert( addr != (addr_t)NULL );
170
171     paddr = (addr_t)(*PTE_OF(addr) | ~PAGE_MASK);
172
173     vm_unmap(addr);
174     vm_vfree(pool, addr);
175     free(paddr);
176 }
```

Here is the call graph for this function:



4.30.1.4 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. 17) 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 28 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, and slab_free().

Referenced by vm_alloc().

```
28                                      {
29     addr_t addr;
30     vm_link_t *head;
31     size_t size;
32 
33     head = pool->head;
34 
35     /* no page available */
36     if(head == (addr_t)NULL) {
37         return (addr_t)NULL;
38     }
39 
40     addr = head->addr;
41     size = head->size - PAGE_SIZE;
42 
44     assert( PAGE_OFFSET_OF(size) == 0 );
45 
46     /* if block is made of only one page, we remove it from the free list */
47     if(size == 0) {
48         pool->head = head->next;
49         slab_free(pool->cache, head);
50     }
51     else {
52         head->size = size;
53         head->addr += PAGE_SIZE;
54     }
55 
57     assert( PAGE_OFFSET_OF(addr) == 0 );
58 
59     return addr;
60 }
```

Here is the call graph for this function:

4.30.1.5 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. 94) (and not **vm_alloc()** (p. 92)).

Parameters:

pool data structure managing the relevant virtual memory region

addr address of virtual page which must be freed

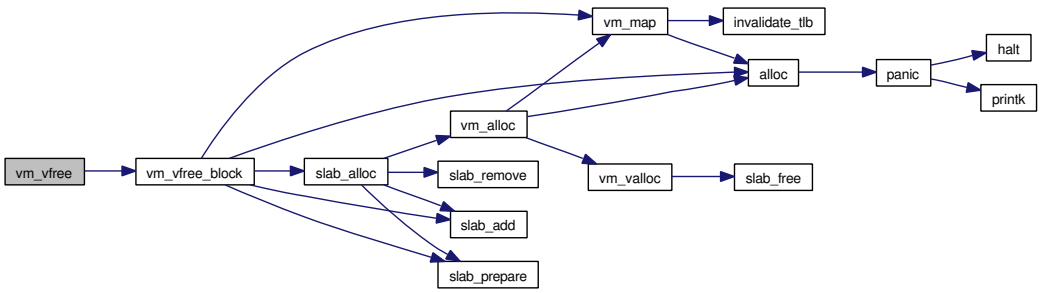
Definition at line 70 of file vm_alloc.c.

References PAGE_SIZE, and vm_vfree_block().

Referenced by vm_free().

```
70                                          {
71     vm_vfree_block(pool, addr, PAGE_SIZE);
72 }
```

Here is the call graph for this function:



4.30.1.6 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 82 of file vm_alloc.c.

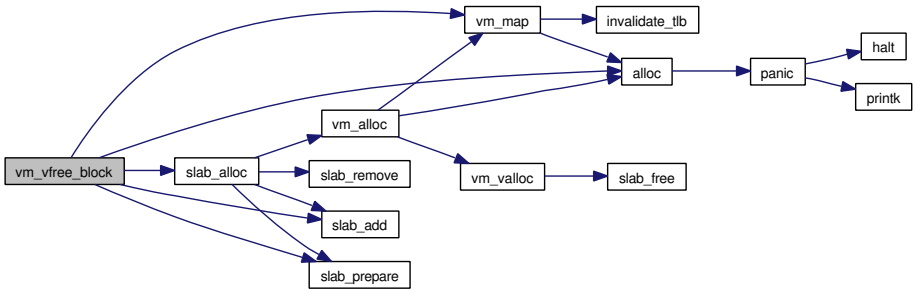
References vm_link_t::addr, alloc(), assert, vm_alloc_t::cache, slab_cache_t::empty, vm_alloc_t::head, vm_link_t::next, NULL, PAGE_OFFSET_OF, PAGE_SIZE, slab_cache_t::partial, vm_link_t::size, slab_add(), slab_alloc(), slab_prepare(), slab_cache_t::vm_allocator, VM_FLAG_KERNEL, and vm_map().

Referenced by vm_vfree().

```
82                                                                  {
83     addr_t phys_page;
84     vm_link_t *link;
85
87     assert( PAGE_OFFSET_OF(addr) == 0 );
88
90     assert( PAGE_OFFSET_OF(size) == 0 );
91
93     assert( addr != (addr_t)NULL );
94
95     /*  The virtual address space allocator needs a slab cache from which to
96         allocate data structures for its free list. Also, each slab cache needs
97         a virtual address space allocator to allocate slabs when needed.
98
99         There can be a mutual dependency between the virtual address space
100         allocator and the slab cache. This is not a problem in general, but a
101         special bootstrapping procedure is needed for initialization of the
102         virtual address space allocator in that case. The virtual address space
103         allocator will actually "donate" a virtual page (backed by physical ram)
104         to the cache for use as a slab.
105
106         This case is handled here
107      */
108     if(pool->head == NULL) {
109         if(pool->cache->vm_allocator == pool) {
110             if(pool->cache->empty == NULL && pool->cache->partial == NULL) {
111                 /* allocate a physical page for slab */
112                 phys_page = alloc(PAGE_SIZE);
113
114                 /* map page */
115                 vm_map(addr, phys_page, VM_FLAG_KERNEL);
116
117                 /* prepare the slab and add it to cache empty list */
118                 slab_prepare(pool->cache, addr);
119                 slab_add(&pool->cache->empty, addr);
120
121                 size -= PAGE_SIZE;
122
123                 /* if the block contained only one page, we have nothing left
124                    to free */
125                 if(size == 0) {
126                     return;
127                 }
128
129                 addr += PAGE_SIZE;
130             }
```

```
131         }
132     }
133
134     link = (vm_link_t *)slab_alloc(pool->cache);
135     link->size = size;
136     link->addr = addr;
137
138     link->next = pool->head;
139     pool->head = link;
140 }
```

Here is the call graph for this function:



4.30.2 Variable Documentation

4.30.2.1 vm_alloc_t global_pool

Definition at line 8 of file vm_alloc.c.

Referenced by kinit().

4.30.2.2 slab_cache_t global_pool_cache

Definition at line 10 of file vm_alloc.c.

Referenced by kinit().

Index