Jinue

Generated by Doxygen 1.8.5

Wed Mar 20 2019 21:26:14

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:

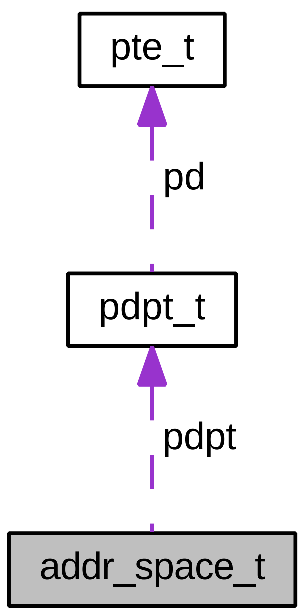
Chapter 3

Data Structure Documentation

3.1 addr_space_t Struct Reference

`#include <hal/types.h>`

Collaboration diagram for addr_space_t:



Data Fields

- **uint32_t cr3**
- union {
  kern_paddr_t pd
  pdpt_t $*$ **pdpt**
  } **top_level**

3.1.1 Detailed Description

Definition at line 80 of file types.h.

3.1.2 Field Documentation

3.1.2.1 uint32_t addr_space_t::cr3

Definition at line 81 of file types.h.

Referenced by vm_boot_init(), vm_pae_create_addr_space(), vm_pae_create_initial_addr_space(), vm_switch_addr_-space(), vm_x86_create_addr_space(), and vm_x86_create_initial_addr_space().

3.1.2.2 kern_paddr_t addr_space_t::pd

Definition at line 83 of file types.h.

Referenced by vm_x86_create_addr_space(), vm_x86_create_initial_addr_space(), vm_x86_destroy_addr_space(), and vm_x86_lookup_page_directory().

3.1.2.3 pdpt_t* addr_space_t::pdpt

Definition at line 84 of file types.h.

Referenced by vm_pae_create_addr_space(), vm_pae_create_initial_addr_space(), vm_pae_destroy_addr_space(), vm_pae_lookup_page_directory(), and vm_pae_unmap_low_alias().

3.1.2.4 union { ... } addr_space_t::top_level

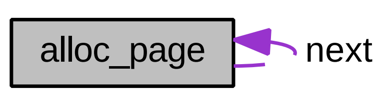
Referenced by vm_pae_create_addr_space(), vm_pae_create_initial_addr_space(), vm_pae_destroy_addr_space(), vm_pae_lookup_page_directory(), vm_pae_unmap_low_alias(), vm_x86_create_addr_space(), vm_x86_create_initial-_addr_space(), vm_x86_destroy_addr_space(), and vm_x86_lookup_page_directory().

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

- include/hal/**types.h**

3.2 alloc_page Struct Reference

Collaboration diagram for alloc_page:



Data Fields

- struct **alloc_page** * **next**

3.2.1 Detailed Description

Definition at line 38 of file page_alloc.c.

3.2.2 Field Documentation

3.2.2.1 struct **alloc_page**∗ alloc_page::next

Definition at line 39 of file page_alloc.c.

Referenced by page_alloc(), and page_free().

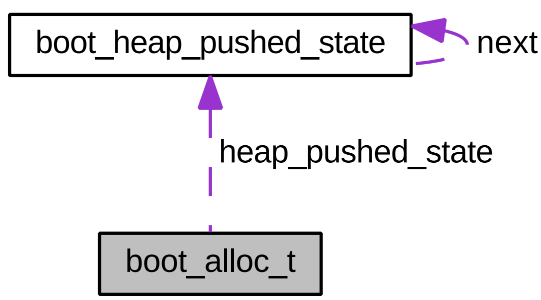
The documentation for this struct was generated from the following file:

- kernel/**page_alloc.c**

3.3 boot_alloc_t Struct Reference

`#include <types.h>`

Collaboration diagram for boot_alloc_t:



Data Fields

- void ∗ **heap_ptr**
- struct **boot_heap_pushed_state** ∗ **heap_pushed_state**
- **addr_t kernel_vm_top**
- **addr_t kernel_vm_limit**
- **kern_paddr_t kernel_paddr_top**
- **kern_paddr_t kernel_paddr_limit**
- **bool its_early**

3.3.1 Detailed Description

Definition at line 48 of file types.h.

3.3.2 Field Documentation

3.3.2.1 void∗ boot_alloc_t::heap_ptr

Definition at line 49 of file types.h.

Referenced by boot_alloc_init(), boot_heap_alloc_size(), and boot_heap_pop().

3.3.2.2 struct boot_heap_pushed_state∗ boot_alloc_t::heap_pushed_state

Definition at line 50 of file types.h.

Referenced by boot_heap_pop(), and boot_heap_push().

3.3.2.3 bool boot_alloc_t::its_early

Definition at line 55 of file types.h.

Referenced by boot_alloc_init(), boot_page_alloc_early(), boot_page_frame_alloc(), boot_vmalloc(), and vm_boot_-init().

3.3.2.4 kern_paddr_t boot_alloc_t::kernel_paddr_limit

Definition at line 54 of file types.h.

Referenced by boot_page_alloc_early(), boot_page_frame_alloc(), and mem_check_memory().

3.3.2.5 kern_paddr_t boot_alloc_t::kernel_paddr_top

Definition at line 53 of file types.h.

Referenced by boot_page_alloc_early(), boot_page_frame_alloc(), and mem_check_memory().

3.3.2.6 addr_t boot_alloc_t::kernel_vm_limit

Definition at line 52 of file types.h.

Referenced by boot_page_alloc_early(), boot_vmalloc(), and mem_check_memory().

3.3.2.7 addr_t boot_alloc_t::kernel_vm_top

Definition at line 51 of file types.h.

Referenced by boot_page_alloc_early(), boot_vmalloc(), and mem_check_memory().

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

- include/**types.h**

3.4 boot_heap_pushed_state Struct Reference

```
#include <types.h>
```

Collaboration diagram for boot_heap_pushed_state:

Data Fields

- struct **boot_heap_pushed_state** ∗ **next**

3.4.1 Detailed Description

Definition at line 44 of file types.h.

3.4.2 Field Documentation

3.4.2.1 struct boot_heap_pushed_state∗ boot_heap_pushed_state::next

Definition at line 45 of file types.h.

Referenced by boot_heap_pop(), and boot_heap_push().

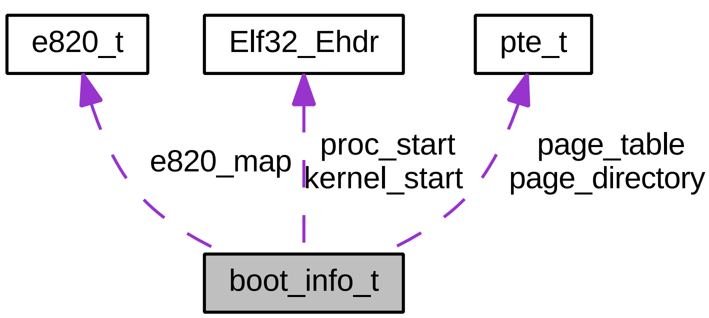
The documentation for this struct was generated from the following file:

- include/**types.h**

3.5 boot_info_t Struct Reference

```
#include <hal/types.h>
```

Collaboration diagram for boot_info_t:



Data Fields

- **Elf32_Ehdr** ∗ **kernel_start**
- **uint32_t kernel_size**
- **Elf32_Ehdr** ∗ **proc_start**

- **uint32_t proc_size**
- void ∗ **image_start**
- void ∗ **image_top**
- **uint32_t ramdisk_start**
- **uint32_t ramdisk_size**
- **uint32_t e820_entries**
- const **e820_t** ∗ **e820_map**
- void ∗ **cmdline**
- void ∗ **boot_heap**
- void ∗ **boot_end**
- **pte_t** ∗ **page_table**
- **pte_t** ∗ **page_directory**
- **uint32_t setup_signature**

3.5.1 Detailed Description

Definition at line 96 of file types.h.

3.5.2 Field Documentation

3.5.2.1 void∗ boot_info_t::boot_end

Definition at line 109 of file types.h.

Referenced by boot_info_dump(), and mem_check_memory().

3.5.2.2 void∗ boot_info_t::boot_heap

Definition at line 108 of file types.h.

Referenced by boot_info_dump(), and kmain().

3.5.2.3 void∗ boot_info_t::cmdline

Definition at line 107 of file types.h.

Referenced by kmain().

3.5.2.4 uint32_t boot_info_t::e820_entries

Definition at line 105 of file types.h.

Referenced by boot_info_dump(), dispatch_syscall(), and mem_check_memory().

3.5.2.5 const e820_t∗ boot_info_t::e820_map

Definition at line 106 of file types.h.

Referenced by boot_info_dump(), dispatch_syscall(), and mem_check_memory().

3.5.2.6 void∗ boot_info_t::image_start

Definition at line 101 of file types.h.

Referenced by boot_info_check(), boot_info_dump(), and vm_boot_init().

3.5.2.7 void∗ boot_info_t::image_top

Definition at line 102 of file types.h.

Referenced by boot_info_dump().

3.5.2.8 uint32_t boot_info_t::kernel_size

Definition at line 98 of file types.h.

Referenced by boot_info_dump(), and kmain().

3.5.2.9 Elf32_Ehdr∗ boot_info_t::kernel_start

Definition at line 97 of file types.h.

Referenced by boot_info_check(), boot_info_dump(), and dump_call_stack().

3.5.2.10 pte_t∗ boot_info_t::page_directory

Definition at line 111 of file types.h.

Referenced by boot_info_dump().

3.5.2.11 pte_t∗ boot_info_t::page_table

Definition at line 110 of file types.h.

Referenced by boot_info_dump().

3.5.2.12 uint32_t boot_info_t::proc_size

Definition at line 100 of file types.h.

Referenced by boot_info_dump().

3.5.2.13 Elf32_Ehdr∗ boot_info_t::proc_start

Definition at line 99 of file types.h.

Referenced by boot_info_dump().

3.5.2.14 uint32_t boot_info_t::ramdisk_size

Definition at line 104 of file types.h.

Referenced by kmain(), and mem_check_memory().

3.5.2.15 uint32_t boot_info_t::ramdisk_start

Definition at line 103 of file types.h.

Referenced by kmain(), and mem_check_memory().

3.5.2.16 uint32_t boot_info_t::setup_signature

Definition at line 112 of file types.h.

Referenced by boot_info_check(), and boot_info_dump().

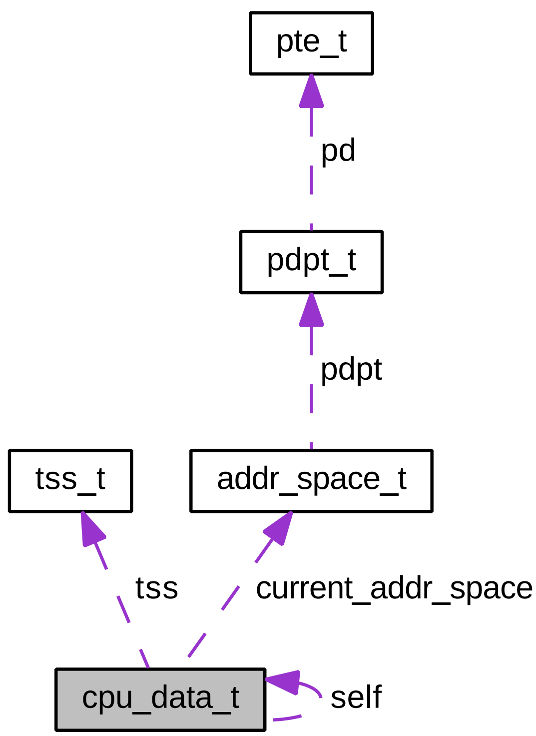
The documentation for this struct was generated from the following file:

- include/hal/**types.h**

3.6 cpu_data_t Struct Reference

`#include <hal/types.h>`

Collaboration diagram for cpu_data_t:



Data Fields

- **seg_descriptor_t gdt** [**GDT_LENGTH**]
- **tss_t tss**
- struct **cpu_data_t** ∗ **self**
- **addr_space_t** ∗ **current_addr_space**

3.6.1 Detailed Description

Definition at line 181 of file types.h.

3.6.2 Field Documentation

3.6.2.1 addr_space_t* cpu_data_t::current_addr_space

Definition at line 188 of file types.h.

Referenced by cpu_init_data(), and vm_switch_addr_space().

3.6.2.2 seg_descriptor_t cpu_data_t::gdt[GDT_LENGTH]

Definition at line 182 of file types.h.

Referenced by cpu_init_data().

3.6.2.3 struct cpu_data_t* cpu_data_t::self

Definition at line 187 of file types.h.

Referenced by cpu_init_data().

3.6.2.4 tss_t cpu_data_t::tss

Definition at line 186 of file types.h.

Referenced by cpu_init_data().

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

- include/hal/**types.h**

3.7 cpu_info_t Struct Reference

```
#include <hal/cpu.h>
```

Data Fields

- unsigned int **dcache_alignment**
- **uint32_t features**
- int **vendor**
- int **family**
- int **model**
- int **stepping**

3.7.1 Detailed Description

Definition at line 97 of file cpu.h.

3.7.2 Field Documentation

3.7.2.1 unsigned int cpu_info_t::dcache_alignment

Definition at line 98 of file cpu.h.

Referenced by cpu_detect_features(), and slab_cache_init().

3.7.2.2 int cpu_info_t::family

Definition at line 101 of file cpu.h.

Referenced by cpu_detect_features().

3.7.2.3 uint32_t cpu_info_t::features

Definition at line 99 of file cpu.h.

Referenced by cpu_detect_features().

3.7.2.4 int cpu_info_t::model

Definition at line 102 of file cpu.h.

Referenced by cpu_detect_features().

3.7.2.5 int cpu_info_t::stepping

Definition at line 103 of file cpu.h.

Referenced by cpu_detect_features().

3.7.2.6 int cpu_info_t::vendor

Definition at line 100 of file cpu.h.

Referenced by cpu_detect_features().

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

- include/hal/**cpu.h**

3.8 e820_t Struct Reference

```
#include <hal/types.h>
```

Data Fields

- **uint64_t addr**
- **uint64_t size**
- **uint32_t type**

3.8.1 Detailed Description

Definition at line 88 of file types.h.

3.8.2 Field Documentation

3.8.2.1 uint64_t e820_t::addr

Definition at line 89 of file types.h.

Referenced by dispatch_syscall(), and mem_check_memory().

3.8.2.2 uint64_t e820_t::size

Definition at line 90 of file types.h.

Referenced by dispatch_syscall().

3.8.2.3 uint32_t e820_t::type

Definition at line 91 of file types.h.

Referenced by dispatch_syscall(), and mem_check_memory().

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

- include/hal/**types.h**

3.9 Elf32_auxv_t Struct Reference

```
#include <jinue-common/elf.h>
```

Data Fields

- int **a_type**
- union {
  int32_t a_val
  } **a_un**

3.9.1 Detailed Description

Definition at line 308 of file elf.h.

3.9.2 Field Documentation

3.9.2.1 int Elf32_auxv_t::a_type

Definition at line 309 of file elf.h.

Referenced by elf_setup_stack().

3.9.2.2 union { ... } Elf32_auxv_t::a_un

Referenced by elf_setup_stack().

3.9.2.3 int32_t Elf32_auxv_t::a_val

Definition at line 311 of file elf.h.

Referenced by elf_setup_stack().

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

- include/jinue-common/**elf.h**

3.10 Elf32_Ehdr Struct Reference

```
#include <jinue-common/elf.h>
```

Data Fields

- unsigned char **e_ident** [**EI_NIDENT**]
- **Elf32_Half e_type**
- **Elf32_Half e_machine**
- **Elf32_Word e_version**
- **Elf32_Addr e_entry**
- **Elf32_Off e_phoff**
- **Elf32_Off e_shoff**
- **Elf32_Word e_flags**
- **Elf32_Half e_ehsize**
- **Elf32_Half e_phentsize**
- **Elf32_Half e_phnum**
- **Elf32_Half e_shentsize**
- **Elf32_Half e_shnum**
- **Elf32_Half e_shstrndx**

3.10.1 Detailed Description

Definition at line 258 of file elf.h.

3.10.2 Field Documentation

3.10.2.1 Elf32_Half Elf32_Ehdr::e_ehsize

Definition at line 267 of file elf.h.

3.10.2.2 Elf32_Addr Elf32_Ehdr::e_entry

Definition at line 263 of file elf.h.

Referenced by elf_check(), and elf_load().

3.10.2.3 Elf32_Word Elf32_Ehdr::e_flags

Definition at line 266 of file elf.h.

Referenced by elf_check().

3.10.2.4 unsigned char Elf32_Ehdr::e_ident[EI_NIDENT]

Definition at line 259 of file elf.h.

Referenced by elf_check().

3.10.2.5 Elf32_Half Elf32_Ehdr::e_machine

Definition at line 261 of file elf.h.

Referenced by elf_check().

3.10.2.6 Elf32_Half Elf32_Ehdr::e_phentsize

Definition at line 268 of file elf.h.

Referenced by elf_check(), and elf_load().

3.10.2.7 Elf32_Half Elf32_Ehdr::e_phnum

Definition at line 269 of file elf.h.

Referenced by elf_check(), and elf_load().

3.10.2.8 Elf32_Off Elf32_Ehdr::e_phoff

Definition at line 264 of file elf.h.

Referenced by elf_check(), and elf_load().

3.10.2.9 Elf32_Half Elf32_Ehdr::e_shentsize

Definition at line 270 of file elf.h.

3.10.2.10 Elf32_Half Elf32_Ehdr::e_shnum

Definition at line 271 of file elf.h.

Referenced by elf_lookup_symbol().

3.10.2.11 Elf32_Off Elf32_Ehdr::e_shoff

Definition at line 265 of file elf.h.

3.10.2.12 Elf32_Half Elf32_Ehdr::e_shstrndx

Definition at line 272 of file elf.h.

3.10.2.13 Elf32_Half Elf32_Ehdr::e_type

Definition at line 260 of file elf.h.

Referenced by elf_check().

3.10.2.14 Elf32_Word Elf32_Ehdr::e_version

Definition at line 262 of file elf.h.

Referenced by elf_check().

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

- include/jinue-common/**elf.h**

3.11 Elf32_Phdr Struct Reference

```
#include <jinue-common/elf.h>
```

Data Fields

- **Elf32_Word p_type**
- **Elf32_Off p_offset**
- **Elf32_Addr p_vaddr**
- **Elf32_Addr p_paddr**
- **Elf32_Word p_filesz**
- **Elf32_Word p_memsz**
- **Elf32_Word p_flags**
- **Elf32_Word p_align**

3.11.1 Detailed Description

Definition at line 275 of file elf.h.

3.11.2 Field Documentation

3.11.2.1 Elf32_Word Elf32_Phdr::p_align

Definition at line 283 of file elf.h.

3.11.2.2 Elf32_Word Elf32_Phdr::p_filesz

Definition at line 280 of file elf.h.

Referenced by elf_load().

3.11.2.3 Elf32_Word Elf32_Phdr::p_flags

Definition at line 282 of file elf.h.

3.11.2.4 Elf32_Word Elf32_Phdr::p_memsz

Definition at line 281 of file elf.h.

Referenced by elf_load().

3.11.2.5 Elf32_Off Elf32_Phdr::p_offset

Definition at line 277 of file elf.h.

3.11.2.6 Elf32_Addr Elf32_Phdr::p_paddr

Definition at line 279 of file elf.h.

3.11.2.7 Elf32_Word Elf32_Phdr::p_type

Definition at line 276 of file elf.h.

3.11.2.8 Elf32_Addr Elf32_Phdr::p_vaddr

Definition at line 278 of file elf.h.

Referenced by elf_load().

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

- include/jinue-common/**elf.h**

3.12 Elf32_Shdr Struct Reference

```
#include <jinue-common/elf.h>
```

Data Fields

- **Elf32_Word sh_name**
- **Elf32_Word sh_type**
- **Elf32_Word sh_flags**
- **Elf32_Addr sh_addr**

- **Elf32_Off sh_offset**
- **Elf32_Word sh_size**
- **Elf32_Word sh_link**
- **Elf32_Word sh_info**
- **Elf32_Word sh_addralign**
- **Elf32_Word sh_entsize**

3.12.1 Detailed Description

Definition at line 286 of file elf.h.

3.12.2 Field Documentation

3.12.2.1 Elf32_Addr Elf32_Shdr::sh_addr

Definition at line 290 of file elf.h.

3.12.2.2 Elf32_Word Elf32_Shdr::sh_addralign

Definition at line 295 of file elf.h.

3.12.2.3 Elf32_Word Elf32_Shdr::sh_entsize

Definition at line 296 of file elf.h.

Referenced by elf_lookup_symbol().

3.12.2.4 Elf32_Word Elf32_Shdr::sh_flags

Definition at line 289 of file elf.h.

3.12.2.5 Elf32_Word Elf32_Shdr::sh_info

Definition at line 294 of file elf.h.

3.12.2.6 Elf32_Word Elf32_Shdr::sh_link

Definition at line 293 of file elf.h.

Referenced by elf_lookup_symbol().

3.12.2.7 Elf32_Word Elf32_Shdr::sh_name

Definition at line 287 of file elf.h.

3.12.2.8 Elf32_Off Elf32_Shdr::sh_offset

Definition at line 291 of file elf.h.

Referenced by elf_lookup_symbol().

3.12.2.9 Elf32_Word Elf32_Shdr::sh_size

Definition at line 292 of file elf.h.

Referenced by elf_lookup_symbol().

3.12.2.10 Elf32_Word Elf32_Shdr::sh_type

Definition at line 288 of file elf.h.

Referenced by elf_lookup_symbol().

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

- include/jinue-common/**elf.h**

3.13 Elf32_Sym Struct Reference

```
#include <jinue-common/elf.h>
```

Data Fields

- **Elf32_Word st_name**
- **Elf32_Addr st_value**
- **Elf32_Word st_size**
- unsigned char **st_info**
- unsigned char **st_other**
- **Elf32_Half st_shndx**

3.13.1 Detailed Description

Definition at line 299 of file elf.h.

3.13.2 Field Documentation

3.13.2.1 unsigned char Elf32_Sym::st_info

Definition at line 303 of file elf.h.

Referenced by elf_lookup_symbol().

3.13.2.2 Elf32_Word Elf32_Sym::st_name

Definition at line 300 of file elf.h.

Referenced by elf_lookup_symbol().

3.13.2.3 unsigned char Elf32_Sym::st_other

Definition at line 304 of file elf.h.

3.13.2.4 Elf32_Half Elf32_Sym::st_shndx

Definition at line 305 of file elf.h.

3.13.2.5 Elf32_Word Elf32_Sym::st_size

Definition at line 302 of file elf.h.

Referenced by elf_lookup_symbol().

3.13.2.6 Elf32_Addr Elf32_Sym::st_value

Definition at line 301 of file elf.h.

Referenced by elf_lookup_symbol().

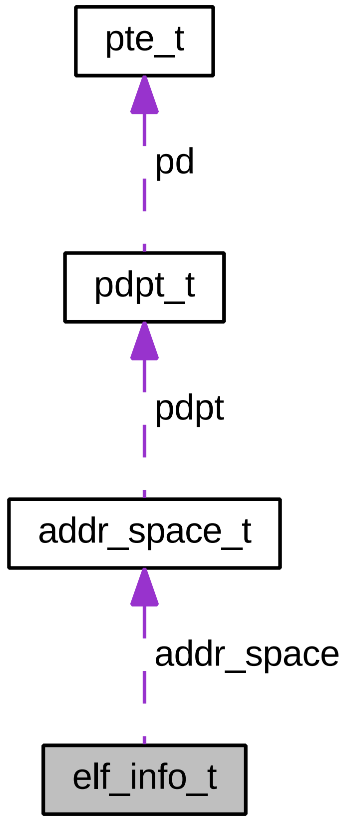
The documentation for this struct was generated from the following file:

- include/jinue-common/**elf.h**

3.14 elf_info_t Struct Reference

```
#include <jinue-common/elf.h>
```

Collaboration diagram for elf_info_t:



Data Fields

- **addr_t entry**
- **addr_t stack_addr**
- **addr_t at_phdr**
- int **at_phent**
- int **at_phnum**
- **addr_space_t * addr_space**

3.14.1 Detailed Description

Definition at line 39 of file elf.h.

3.14.2 Field Documentation

3.14.2.1 addr_space_t* elf_info_t::addr_space

Definition at line 45 of file elf.h.

Referenced by elf_load(), and elf_setup_stack().

3.14.2.2 addr_t elf_info_t::at_phdr

Definition at line 42 of file elf.h.

Referenced by elf_load(), and elf_setup_stack().

3.14.2.3 int elf_info_t::at_phent

Definition at line 43 of file elf.h.

Referenced by elf_load(), and elf_setup_stack().

3.14.2.4 int elf_info_t::at_phnum

Definition at line 44 of file elf.h.

Referenced by elf_load(), and elf_setup_stack().

3.14.2.5 addr_t elf_info_t::entry

Definition at line 40 of file elf.h.

Referenced by elf_load(), elf_setup_stack(), and kmain().

3.14.2.6 addr_t elf_info_t::stack_addr

Definition at line 41 of file elf.h.

Referenced by elf_setup_stack(), and kmain().

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

- include/jinue-common/**elf.h**

3.15 elf_symbol_t Struct Reference

```
#include <jinue-common/elf.h>
```

Data Fields

- **Elf32_Addr addr**
- const char * **name**

3.15.1 Detailed Description

Definition at line 48 of file elf.h.

3.15.2 Field Documentation

3.15.2.1 Elf32_Addr elf_symbol_t::addr

Definition at line 49 of file elf.h.

Referenced by dump_call_stack(), and elf_lookup_symbol().

3.15.2.2 const char* elf_symbol_t::name

Definition at line 50 of file elf.h.

Referenced by dump_call_stack(), and elf_lookup_symbol().

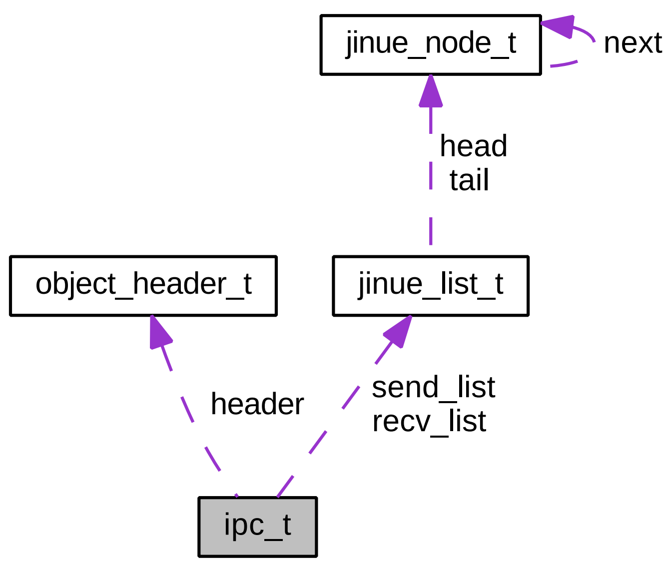
The documentation for this struct was generated from the following file:

- include/jinue-common/**elf.h**

3.16 ipc_t Struct Reference

```
#include <types.h>
```

Collaboration diagram for ipc_t:



Data Fields

- **object_header_t header**
- **jinue_list_t send_list**
- **jinue_list_t recv_list**

3.16.1 Detailed Description

Definition at line 100 of file types.h.

3.16.2 Field Documentation

3.16.2.1 object_header_t ipc_t::header

Definition at line 101 of file types.h.

Referenced by dispatch_syscall(), and ipc_object_create().

3.16.2.2 jinue_list_t ipc_t::recv_list

Definition at line 103 of file types.h.

Referenced by ipc_receive(), and ipc_send().

3.16.2.3 jinue_list_t ipc_t::send_list

Definition at line 102 of file types.h.

Referenced by ipc_receive(), and ipc_send().

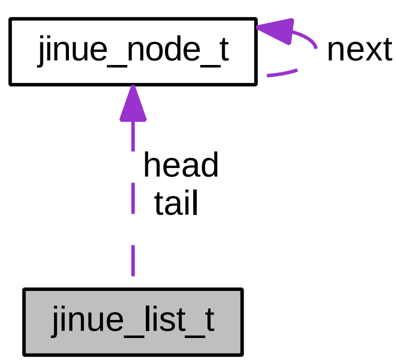
The documentation for this struct was generated from the following file:

- include/**types.h**

3.17 jinue_list_t Struct Reference

```
#include <jinue-common/list.h>
```

Collaboration diagram for jinue_list_t:



Data Fields

- **jinue_node_t** ∗ **head**
- **jinue_node_t** ∗ **tail**

3.17.1 Detailed Description

Definition at line 55 of file list.h.

3.17.2 Field Documentation

3.17.2.1 jinue_node_t* jinue_list_t::head

Definition at line 56 of file list.h.

3.17.2.2 jinue_node_t* jinue_list_t::tail

Definition at line 57 of file list.h.

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

- include/jinue-common/**list.h**

3.18 jinue_mem_entry_t Struct Reference

```
#include <jinue-common/types.h>
```

Data Fields

- **uint64_t addr**
- **uint64_t size**
- **uint32_t type**

3.18.1 Detailed Description

Definition at line 38 of file types.h.

3.18.2 Field Documentation

3.18.2.1 uint64_t jinue_mem_entry_t::addr

Definition at line 39 of file types.h.

Referenced by dispatch_syscall().

3.18.2.2 uint64_t jinue_mem_entry_t::size

Definition at line 40 of file types.h.

Referenced by dispatch_syscall().

3.18.2.3 uint32_t jinue_mem_entry_t::type

Definition at line 41 of file types.h.

Referenced by dispatch_syscall().

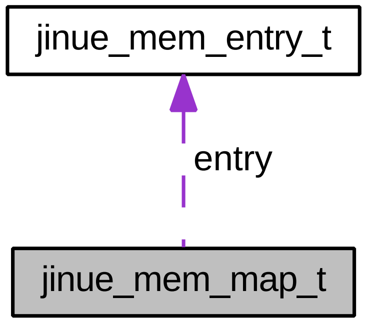
The documentation for this struct was generated from the following file:

- include/jinue-common/**types.h**

3.19 jinue_mem_map_t Struct Reference

```
#include <jinue-common/types.h>
```

Collaboration diagram for jinue_mem_map_t:



Data Fields

- **uint32_t num_entries**
- **jinue_mem_entry_t entry** [ ]

3.19.1 Detailed Description

Definition at line 44 of file types.h.

3.19.2 Field Documentation

3.19.2.1 jinue_mem_entry_t jinue_mem_map_t::entry[ ]

Definition at line 46 of file types.h.

Referenced by dispatch_syscall().

3.19.2.2 uint32_t jinue_mem_map_t::num_entries

Definition at line 45 of file types.h.

Referenced by dispatch_syscall().

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

- include/jinue-common/**types.h**

3.20 jinue_message_t Struct Reference

```
#include <jinue/ipc.h>
```

Data Fields

- **uintptr_t function**
- **uintptr_t cookie**
- **size_t buffer_size**
- **size_t data_size**
- **size_t desc_n**

3.20.1 Detailed Description

Definition at line 38 of file ipc.h.

3.20.2 Field Documentation

3.20.2.1 size_t jinue_message_t::buffer_size

Definition at line 41 of file ipc.h.

3.20.2.2 uintptr_t jinue_message_t::cookie

Definition at line 40 of file ipc.h.

3.20.2.3 size_t jinue_message_t::data_size

Definition at line 42 of file ipc.h.

3.20.2.4 size_t jinue_message_t::desc_n

Definition at line 43 of file ipc.h.

3.20.2.5 uintptr_t jinue_message_t::function

Definition at line 39 of file ipc.h.

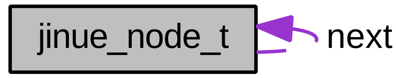
The documentation for this struct was generated from the following file:

- include/jinue/**ipc.h**

3.21 jinue_node_t Struct Reference

```
#include <jinue-common/list.h>
```

Collaboration diagram for jinue_node_t:



Data Fields

- struct **jinue_node_t** ∗ **next**

3.21.1 Detailed Description

Definition at line 38 of file list.h.

3.21.2 Field Documentation

3.21.2.1 struct jinue_node_t∗ jinue_node_t::next

Definition at line 39 of file list.h.

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

- include/jinue-common/**list.h**

3.22 jinue_reply_t Struct Reference

```
#include <jinue/ipc.h>
```

Data Fields

- **size_t data_size**
- **size_t desc_n**

3.22.1 Detailed Description

Definition at line 46 of file ipc.h.

3.22.2 Field Documentation

3.22.2.1 size_t jinue_reply_t::data_size

Definition at line 47 of file ipc.h.

3.22.2.2 size_t jinue_reply_t::desc_n

Definition at line 48 of file ipc.h.

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

- include/jinue/**ipc.h**

3.23 jinue_syscall_args_t Struct Reference

```
#include <jinue-common/syscall.h>
```

Data Fields

- **uintptr_t arg0**
- **uintptr_t arg1**
- **uintptr_t arg2**
- **uintptr_t arg3**

3.23.1 Detailed Description

Definition at line 39 of file syscall.h.

3.23.2 Field Documentation

3.23.2.1 uintptr_t jinue_syscall_args_t::arg0

Definition at line 40 of file syscall.h.

Referenced by dispatch_syscall(), ipc_receive(), and ipc_send().

3.23.2.2 uintptr_t jinue_syscall_args_t::arg1

Definition at line 41 of file syscall.h.

Referenced by dispatch_syscall(), ipc_receive(), and ipc_send().

3.23.2.3 uintptr_t jinue_syscall_args_t::arg2

Definition at line 42 of file syscall.h.

Referenced by dispatch_syscall(), ipc_receive(), ipc_reply(), and ipc_send().

3.23.2.4 uintptr_t jinue_syscall_args_t::arg3

Definition at line 43 of file syscall.h.

Referenced by dispatch_syscall(), ipc_receive(), and ipc_reply().

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

- include/jinue-common/**syscall.h**

3.24 kernel_context_t Struct Reference

```
#include <hal/types.h>
```

Data Fields

- **uint32_t edi**
- **uint32_t esi**
- **uint32_t ebx**
- **uint32_t ebp**
- **uint32_t eip**

3.24.1 Detailed Description

Definition at line 219 of file types.h.

3.24.2 Field Documentation

3.24.2.1 uint32_t kernel_context_t::ebp

Definition at line 223 of file types.h.

3.24.2.2 uint32_t kernel_context_t::ebx

Definition at line 222 of file types.h.

3.24.2.3 uint32_t kernel_context_t::edi

Definition at line 220 of file types.h.

3.24.2.4 uint32_t kernel_context_t::eip

Definition at line 224 of file types.h.

Referenced by thread_page_init().

3.24.2.5 uint32_t kernel_context_t::esi

Definition at line 221 of file types.h.

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

- include/hal/**types.h**

3.25 message_info_t Struct Reference

```
#include <types.h>
```

Data Fields

- **uintptr_t function**
- **uintptr_t cookie**
- **size_t buffer_size**
- **size_t data_size**
- **size_t desc_n**
- **size_t total_size**

3.25.1 Detailed Description

Definition at line 78 of file types.h.

3.25.2 Field Documentation

3.25.2.1 size_t message_info_t::buffer_size

Definition at line 81 of file types.h.

Referenced by ipc_reply(), and ipc_send().

3.25.2.2 uintptr_t message_info_t::cookie

Definition at line 80 of file types.h.

Referenced by ipc_send().

3.25.2.3 size_t message_info_t::data_size

Definition at line 82 of file types.h.

Referenced by ipc_receive(), ipc_reply(), and ipc_send().

3.25.2.4 size_t message_info_t::desc_n

Definition at line 83 of file types.h.

Referenced by ipc_reply(), and ipc_send().

3.25.2.5 uintptr_t message_info_t::function

Definition at line 79 of file types.h.

Referenced by ipc_send().

3.25.2.6 size_t message_info_t::total_size

Definition at line 84 of file types.h.

Referenced by ipc_receive(), and ipc_send().

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

- include/**types.h**

3.26 object_header_t Struct Reference

```
#include <types.h>
```

Data Fields

- int **type**
- int **ref_count**
- int **flags**

3.26.1 Detailed Description

Definition at line 58 of file types.h.

3.26.2 Field Documentation

3.26.2.1 int object_header_t::flags

Definition at line 61 of file types.h.

Referenced by ipc_object_create(), ipc_receive(), and ipc_send().

3.26.2.2 int object_header_t::ref_count

Definition at line 60 of file types.h.

3.26.2.3 int object_header_t::type

Definition at line 59 of file types.h.

Referenced by ipc_receive(), and ipc_send().

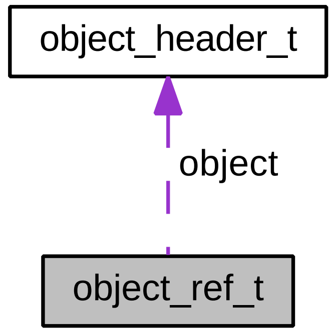
The documentation for this struct was generated from the following file:

- include/**types.h**

3.27 object_ref_t Struct Reference

`#include <types.h>`

Collaboration diagram for object_ref_t:



Data Fields

- **object_header_t ∗ object**
- **uintptr_t flags**
- **uintptr_t cookie**

3.27.1 Detailed Description

Definition at line 64 of file types.h.

3.27.2 Field Documentation

3.27.2.1 uintptr_t object_ref_t::cookie

Definition at line 67 of file types.h.

Referenced by dispatch_syscall().

3.27.2.2 uintptr_t object_ref_t::flags

Definition at line 66 of file types.h.

Referenced by dispatch_syscall().

3.27.2.3 object_header_t∗ object_ref_t::object

Definition at line 65 of file types.h.

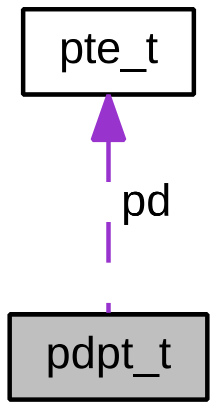
Referenced by dispatch_syscall().

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

- include/**types.h**

3.28 pdpt_t Struct Reference

Collaboration diagram for pdpt_t:



Data Fields

- **pte_t pd** [**PDPT_ENTRIES**]

3.28.1 Detailed Description

Definition at line 56 of file vm_pae.c.

3.28.2 Field Documentation

3.28.2.1 pte_t pdpt_t::pd[PDPT_ENTRIES]

Definition at line 57 of file vm_pae.c.

Referenced by vm_pae_create_addr_space(), vm_pae_create_initial_addr_space(), vm_pae_destroy_addr_space(), vm_pae_lookup_page_directory(), and vm_pae_unmap_low_alias().

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

- kernel/hal/**vm_pae.c**

3.29 pfalloc_cache_t Struct Reference

```
#include <pfalloc.h>
```

Data Fields

- **kern_paddr_t * ptr**
- **uint32_t count**

3.29.1 Detailed Description

Definition at line 42 of file pfalloc.h.

3.29.2 Field Documentation

3.29.2.1 uint32_t pfalloc_cache_t::count

Definition at line 44 of file pfalloc.h.

Referenced by init_pfalloc_cache(), pfalloc_from(), and pffree_to().

3.29.2.2 kern_paddr_t* pfalloc_cache_t::ptr

Definition at line 43 of file pfalloc.h.

Referenced by init_pfalloc_cache(), pfalloc_from(), and pffree_to().

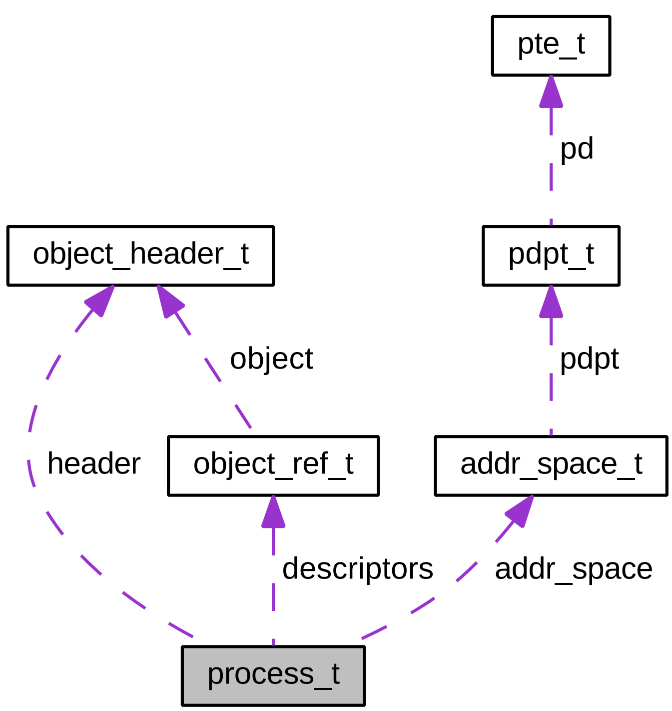
The documentation for this struct was generated from the following file:

- include/**pfalloc.h**

3.30 process_t Struct Reference

`#include <types.h>`

Collaboration diagram for process_t:



Data Fields

- **object_header_t header**
- **addr_space_t addr_space**
- **object_ref_t descriptors** [**PROCESS_MAX_DESCRIPTORS**]

3.30.1 Detailed Description

Definition at line 72 of file types.h.

3.30.2 Field Documentation

3.30.2.1 addr_space_t process_t::addr_space

Definition at line 74 of file types.h.

Referenced by kmain(), process_create(), process_create_initial(), and thread_switch().

3.30.2.2 object_ref_t process_t::descriptors[PROCESS_MAX_DESCRIPTORS]

Definition at line 75 of file types.h.

Referenced by process_get_descriptor().

3.30.2.3 object_header_t process_t::header

Definition at line 73 of file types.h.

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

- include/**types.h**

3.31 pseudo_descriptor_t Struct Reference

```
#include <hal/types.h>
```

Data Fields

- **uint16_t padding**
- **uint16_t limit**
- **addr_t addr**

3.31.1 Detailed Description

Definition at line 119 of file types.h.

3.31.2 Field Documentation

3.31.2.1 addr_t pseudo_descriptor_t::addr

Definition at line 122 of file types.h.

3.31.2.2 uint16_t pseudo_descriptor_t::limit

Definition at line 121 of file types.h.

3.31.2.3 uint16_t pseudo_descriptor_t::padding

Definition at line 120 of file types.h.

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

- include/hal/**types.h**

3.32 pte_t Struct Reference

Data Fields

- **uint64_t entry**
- **uint32_t entry**

3.32.1 Detailed Description

Definition at line 52 of file vm_pae.c.

3.32.2 Field Documentation

3.32.2.1 uint32_t pte_t::entry

Definition at line 38 of file vm_x86.c.

3.32.2.2 uint64_t pte_t::entry

Definition at line 53 of file vm_pae.c.

Referenced by vm_pae_clear_pte(), vm_pae_copy_pte(), vm_pae_destroy_addr_space(), vm_pae_get_pte_flags(), vm_pae_get_pte_paddr(), vm_pae_set_pte(), vm_pae_set_pte_flags(), vm_x86_clear_pte(), vm_x86_copy_pte(), vm-_x86_get_pte_flags(), vm_x86_get_pte_paddr(), vm_x86_set_pte(), and vm_x86_set_pte_flags().

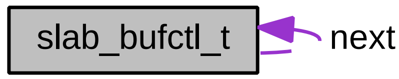
The documentation for this struct was generated from the following files:

- kernel/hal/**vm_pae.c**
- kernel/hal/**vm_x86.c**

3.33 slab_bufctl_t Struct Reference

`#include <slab.h>`

Collaboration diagram for slab_bufctl_t:



Data Fields

- struct **slab_bufctl_t** ∗ **next**

3.33.1 Detailed Description

Definition at line 84 of file slab.h.

3.33.2 Field Documentation

3.33.2.1 struct slab_bufctl_t∗ slab_bufctl_t::next

Definition at line 85 of file slab.h.

Referenced by slab_cache_alloc(), and slab_cache_free().

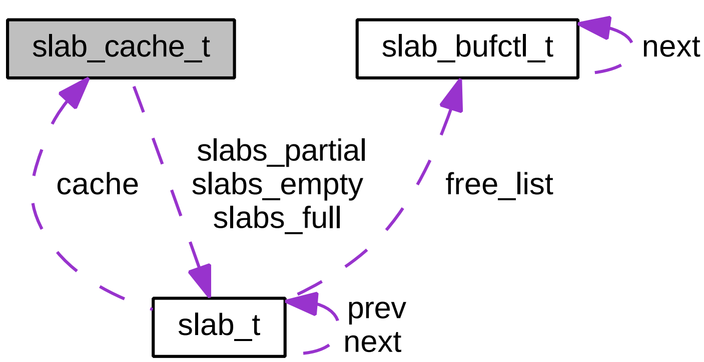
The documentation for this struct was generated from the following file:

- include/**slab.h**

3.34 slab_cache_t Struct Reference

`#include <slab.h>`

Collaboration diagram for slab_cache_t:



Data Fields

- struct **slab_t** ∗ **slabs_empty**
- struct **slab_t** ∗ **slabs_partial**
- struct **slab_t** ∗ **slabs_full**
- unsigned int **empty_count**
- **size_t obj_size**
- **size_t alloc_size**
- **size_t alignment**
- **size_t bufctl_offset**
- **size_t next_colour**
- **size_t max_colour**
- unsigned int **working_set**
- **slab_ctor_t ctor**
- **slab_ctor_t dtor**
- char ∗ **name**
- int **flags**

3.34.1 Detailed Description

Definition at line 64 of file slab.h.

3.34.2 Field Documentation

3.34.2.1 size_t slab_cache_t::alignment

Definition at line 71 of file slab.h.

Referenced by slab_cache_init().

3.34.2.2 size_t slab_cache_t::alloc_size

Definition at line 70 of file slab.h.

Referenced by slab_cache_init().

3.34.2.3 size_t slab_cache_t::bufctl_offset

Definition at line 72 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_free(), and slab_cache_init().

3.34.2.4 slab_ctor_t slab_cache_t::ctor

Definition at line 76 of file slab.h.

Referenced by slab_cache_alloc(), and slab_cache_init().

3.34.2.5 slab_ctor_t slab_cache_t::dtor

Definition at line 77 of file slab.h.

Referenced by slab_cache_free(), and slab_cache_init().

3.34.2.6 unsigned int slab_cache_t::empty_count

Definition at line 68 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_free(), slab_cache_init(), and slab_cache_reap().

3.34.2.7 int slab_cache_t::flags

Definition at line 79 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_free(), and slab_cache_init().

3.34.2.8 size_t slab_cache_t::max_colour

Definition at line 74 of file slab.h.

Referenced by slab_cache_init().

3.34.2.9 char* slab_cache_t::name

Definition at line 78 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_free(), and slab_cache_init().

3.34.2.10 size_t slab_cache_t::next_colour

Definition at line 73 of file slab.h.

Referenced by slab_cache_init().

3.34.2.11 size_t slab_cache_t::obj_size

Definition at line 69 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_free(), and slab_cache_init().

3.34.2.12 struct slab_t* slab_cache_t::slabs_empty

Definition at line 65 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_free(), slab_cache_init(), and slab_cache_reap().

3.34.2.13 struct slab_t* slab_cache_t::slabs_full

Definition at line 67 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_free(), and slab_cache_init().

3.34.2.14 struct slab_t* slab_cache_t::slabs_partial

Definition at line 66 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_free(), and slab_cache_init().

3.34.2.15 unsigned int slab_cache_t::working_set

Definition at line 75 of file slab.h.

Referenced by slab_cache_init(), slab_cache_reap(), and slab_cache_set_working_set().

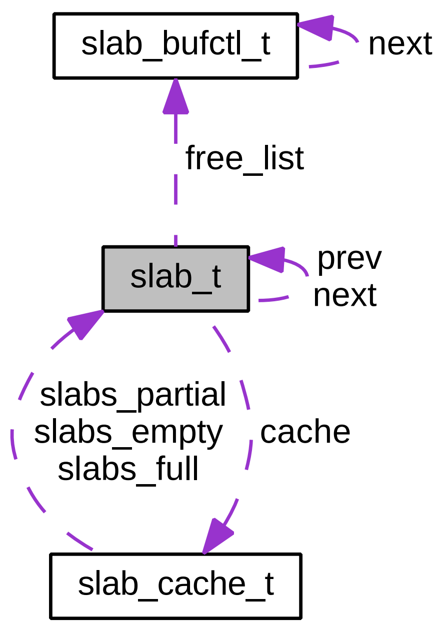
The documentation for this struct was generated from the following file:

- include/**slab.h**

3.35 slab_t Struct Reference

```
#include <slab.h>
```

Collaboration diagram for slab_t:



Data Fields

- struct **slab_t** ∗ **prev**
- struct **slab_t** ∗ **next**
- **slab_cache_t** ∗ **cache**
- unsigned int **obj_count**
- **size_t colour**
- **slab_bufctl_t** ∗ **free_list**

3.35.1 Detailed Description

Definition at line 90 of file slab.h.

3.35.2 Field Documentation

3.35.2.1 slab_cache_t* slab_t::cache

Definition at line 94 of file slab.h.

Referenced by slab_cache_free().

3.35.2.2 size_t slab_t::colour

Definition at line 97 of file slab.h.

3.35.2.3 slab_bufctl_t* slab_t::free_list

Definition at line 98 of file slab.h.

Referenced by slab_cache_alloc(), and slab_cache_free().

3.35.2.4 struct slab_t* slab_t::next

Definition at line 92 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_free(), and slab_cache_reap().

3.35.2.5 unsigned int slab_t::obj_count

Definition at line 96 of file slab.h.

Referenced by slab_cache_alloc(), and slab_cache_free().

3.35.2.6 struct slab_t* slab_t::prev

Definition at line 91 of file slab.h.

Referenced by slab_cache_alloc(), and slab_cache_free().

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

- include/**slab.h**

3.36 thread_context_t Struct Reference

```
#include <hal/types.h>
```

Data Fields

- **addr_t saved_stack_pointer**
- **addr_t local_storage_addr**
- **size_t local_storage_size**

3.36.1 Detailed Description

Definition at line 72 of file types.h.

3.36.2 Field Documentation

3.36.2.1 addr_t thread_context_t::local_storage_addr

Definition at line 76 of file types.h.

Referenced by thread_page_init().

3.36.2.2 size_t thread_context_t::local_storage_size

Definition at line 77 of file types.h.

3.36.2.3 addr_t thread_context_t::saved_stack_pointer

Definition at line 75 of file types.h.

Referenced by thread_page_init().

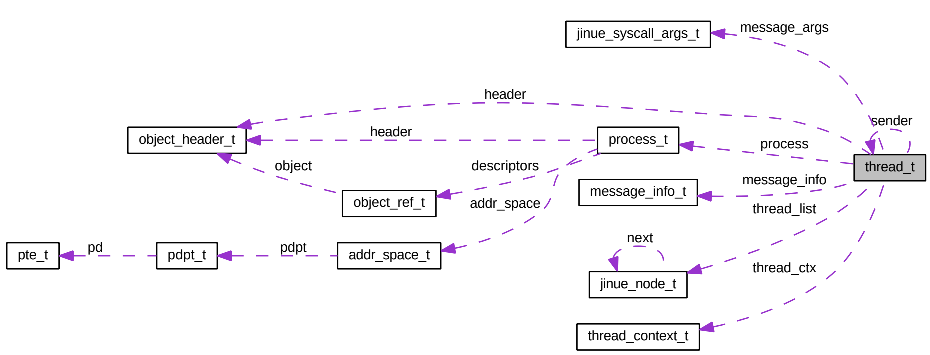
The documentation for this struct was generated from the following file:

- include/hal/**types.h**

3.37 thread_t Struct Reference

`#include <types.h>`

Collaboration diagram for thread_t:



Data Fields

- **object_header_t header**
- **thread_context_t thread_ctx**
- **jinue_node_t thread_list**
- **process_t * process**
- struct **thread_t * sender**
- **jinue_syscall_args_t * message_args**
- **message_info_t message_info**
- char **message_buffer** [**JINUE_SEND_MAX_SIZE**]

3.37.1 Detailed Description

Definition at line 87 of file types.h.

3.37.2 Field Documentation

3.37.2.1 object_header_t thread_t::header

Definition at line 88 of file types.h.

Referenced by ipc_receive(), ipc_reply(), and ipc_send().

3.37.2.2 jinue_syscall_args_t* thread_t::message_args

Definition at line 93 of file types.h.

Referenced by ipc_receive(), ipc_reply(), and ipc_send().

3.37.2.3 char thread_t::message_buffer[JINUE_SEND_MAX_SIZE]

Definition at line 95 of file types.h.

Referenced by ipc_receive(), ipc_reply(), and ipc_send().

3.37.2.4 message_info_t thread_t::message_info

Definition at line 94 of file types.h.

Referenced by ipc_receive(), ipc_reply(), and ipc_send().

3.37.2.5 process_t* thread_t::process

Definition at line 91 of file types.h.

Referenced by dispatch_syscall(), ipc_receive(), ipc_send(), and thread_switch().

3.37.2.6 struct thread_t* thread_t::sender

Definition at line 92 of file types.h.

Referenced by ipc_receive(), ipc_reply(), and ipc_send().

3.37.2.7 thread_context_t thread_t::thread_ctx

Definition at line 89 of file types.h.

Referenced by thread_page_init(), and thread_switch().

3.37.2.8 jinue_node_t thread_t::thread_list

Definition at line 90 of file types.h.

Referenced by ipc_receive(), ipc_send(), and thread_ready().

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

- include/**types.h**

3.38 trapframe_t Struct Reference

```
#include <hal/types.h>
```

Data Fields

- **uint32_t eax**
- **uint32_t ebx**
- **uint32_t esi**
- **uint32_t edi**
- **uint32_t edx**
- **uint32_t ecx**
- **uint32_t ds**
- **uint32_t es**
- **uint32_t fs**
- **uint32_t gs**
- **uint32_t errcode**
- **uint32_t ivt**
- **uint32_t ebp**
- **uint32_t eip**
- **uint32_t cs**
- **uint32_t eflags**
- **uint32_t esp**
- **uint32_t ss**

3.38.1 Detailed Description

Definition at line 193 of file types.h.

3.38.2 Field Documentation

3.38.2.1 uint32_t trapframe_t::cs

Definition at line 213 of file types.h.

Referenced by thread_page_init().

3.38.2.2 uint32_t trapframe_t::ds

Definition at line 205 of file types.h.

Referenced by thread_page_init().

3.38.2.3 uint32_t trapframe_t::eax

Definition at line 196 of file types.h.

3.38.2.4 uint32_t trapframe_t::ebp

Definition at line 211 of file types.h.

3.38.2.5 uint32_t trapframe_t::ebx

Definition at line 198 of file types.h.

3.38.2.6 uint32_t trapframe_t::ecx

Definition at line 204 of file types.h.

3.38.2.7 uint32_t trapframe_t::edi

Definition at line 202 of file types.h.

3.38.2.8 uint32_t trapframe_t::edx

Definition at line 203 of file types.h.

3.38.2.9 uint32_t trapframe_t::eflags

Definition at line 214 of file types.h.

Referenced by thread_page_init().

3.38.2.10 uint32_t trapframe_t::eip

Definition at line 212 of file types.h.

Referenced by dispatch_interrupt(), and thread_page_init().

3.38.2.11 uint32_t trapframe_t::errcode

Definition at line 209 of file types.h.

Referenced by dispatch_interrupt().

3.38.2.12 uint32_t trapframe_t::es

Definition at line 206 of file types.h.

Referenced by thread_page_init().

3.38.2.13 uint32_t trapframe_t::esi

Definition at line 200 of file types.h.

3.38.2.14 uint32_t trapframe_t::esp

Definition at line 215 of file types.h.

Referenced by thread_page_init().

3.38.2.15 uint32_t trapframe_t::fs

Definition at line 207 of file types.h.

Referenced by thread_page_init().

3.38.2.16 uint32_t trapframe_t::gs

Definition at line 208 of file types.h.

Referenced by thread_page_init().

3.38.2.17 uint32_t trapframe_t::ivt

Definition at line 210 of file types.h.

Referenced by dispatch_interrupt().

3.38.2.18 uint32_t trapframe_t::ss

Definition at line 216 of file types.h.

Referenced by thread_page_init().

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

- include/hal/**types.h**

3.39 tss_t Struct Reference

```
#include <hal/types.h>
```

Data Fields

- **uint16_t prev**
- **addr_t esp0**
- **uint16_t ss0**
- **addr_t esp1**
- **uint16_t ss1**
- **addr_t esp2**
- **uint16_t ss2**
- **uint32_t cr3**
- **uint32_t eip**
- **uint32_t eflags**
- **uint32_t eax**
- **uint32_t ecx**
- **uint32_t edx**
- **uint32_t ebx**
- **uint32_t esp**
- **uint32_t ebp**
- **uint32_t esi**
- **uint32_t edi**
- **uint16_t es**
- **uint16_t cs**
- **uint16_t ss**
- **uint16_t ds**
- **uint16_t fs**
- **uint16_t gs**
- **uint16_t ldt**
- **uint16_t debug**
- **uint16_t iomap**

3.39.1 Detailed Description

Definition at line 125 of file types.h.

3.39.2 Field Documentation

3.39.2.1 uint32_t tss_t::cr3

Definition at line 141 of file types.h.

3.39.2.2 uint16_t tss_t::cs

Definition at line 165 of file types.h.

3.39.2.3 uint16_t tss_t::debug

Definition at line 177 of file types.h.

3.39.2.4 uint16_t tss_t::ds

Definition at line 169 of file types.h.

3.39.2.5 uint32_t tss_t::eax

Definition at line 147 of file types.h.

3.39.2.6 uint32_t tss_t::ebp

Definition at line 157 of file types.h.

3.39.2.7 uint32_t tss_t::ebx

Definition at line 153 of file types.h.

3.39.2.8 uint32_t tss_t::ecx

Definition at line 149 of file types.h.

3.39.2.9 uint32_t tss_t::edi

Definition at line 161 of file types.h.

3.39.2.10 uint32_t tss_t::edx

Definition at line 151 of file types.h.

3.39.2.11 uint32_t tss_t::eflags

Definition at line 145 of file types.h.

3.39.2.12 uint32_t tss_t::eip

Definition at line 143 of file types.h.

3.39.2.13 uint16_t tss_t::es

Definition at line 163 of file types.h.

3.39.2.14 uint32_t tss_t::esi

Definition at line 159 of file types.h.

3.39.2.15 uint32_t tss_t::esp

Definition at line 155 of file types.h.

3.39.2.16 addr_t tss_t::esp0

Definition at line 129 of file types.h.

Referenced by cpu_init_data(), and thread_context_switch().

3.39.2.17 addr_t tss_t::esp1

Definition at line 133 of file types.h.

Referenced by cpu_init_data(), and thread_context_switch().

3.39.2.18 addr_t tss_t::esp2

Definition at line 137 of file types.h.

Referenced by cpu_init_data(), and thread_context_switch().

3.39.2.19 uint16_t tss_t::fs

Definition at line 171 of file types.h.

3.39.2.20 uint16_t tss_t::gs

Definition at line 173 of file types.h.

3.39.2.21 uint16_t tss_t::iomap

Definition at line 178 of file types.h.

3.39.2.22 uint16_t tss_t::ldt

Definition at line 175 of file types.h.

3.39.2.23 uint16_t tss_t::prev

Definition at line 127 of file types.h.

3.39.2.24 uint16_t tss_t::ss

Definition at line 167 of file types.h.

3.39.2.25 uint16_t tss_t::ss0

Definition at line 131 of file types.h.

Referenced by cpu_init_data().

3.39.2.26 uint16_t tss_t::ss1

Definition at line 135 of file types.h.

Referenced by cpu_init_data().

3.39.2.27 uint16_t tss_t::ss2

Definition at line 139 of file types.h.

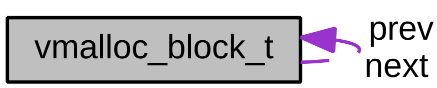
Referenced by cpu_init_data().

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

- include/hal/**types.h**

3.40 vmalloc_block_t Struct Reference

Collaboration diagram for vmalloc_block_t:



Data Fields

- **addr_t** ∗ **stack_ptr**

  stack pointer for stack of free pages in partially allocated blocks
- **addr_t** ∗ **stack_base**

  base address of free page stack
- struct **vmalloc_block_t** ∗ **prev**

  link previous block in free list
- struct **vmalloc_block_t** ∗ **next**

  link next block in free list

3.40.1 Detailed Description

Definition at line 110 of file vmalloc.c.

3.40.2 Field Documentation

3.40.2.1 struct vmalloc_block_t* vmalloc_block_t::next

link next block in free list

Definition at line 121 of file vmalloc.c.

3.40.2.2 struct vmalloc_block_t* vmalloc_block_t::prev

link previous block in free list

Definition at line 118 of file vmalloc.c.

3.40.2.3 addr_t* vmalloc_block_t::stack_base

base address of free page stack

Definition at line 115 of file vmalloc.c.

3.40.2.4 addr_t* vmalloc_block_t::stack_ptr

stack pointer for stack of free pages in partially allocated blocks

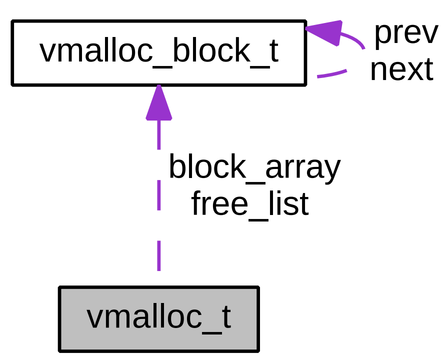
Definition at line 112 of file vmalloc.c.

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

- kernel/**vmalloc.c**

3.41 vmalloc_t Struct Reference

Collaboration diagram for vmalloc_t:



Data Fields

- **addr_t base_addr**

  start address of the first block managed by the allocator

- **addr_t start_addr**

  start address of the address space region managed by the allocator
- **addr_t end_addr**

  end address of the address space region managed by the allocator
- **addr_t init_limit**

  address up to which blocks have been initialized
- unsigned int **block_count**

  number of memory blocks managed by this allocator
- struct **vmalloc_block_t** ∗ **block_array**

  array of memory block descriptors
- struct **vmalloc_block_t** ∗ **free_list**

  list of completely or partially free blocks

3.41.1 Detailed Description

Definition at line 87 of file vmalloc.c.

3.41.2 Field Documentation

3.41.2.1 addr_t vmalloc_t::base_addr

start address of the first block managed by the allocator

Definition at line 89 of file vmalloc.c.

3.41.2.2 struct vmalloc_block_t∗ vmalloc_t::block_array

array of memory block descriptors

Definition at line 104 of file vmalloc.c.

3.41.2.3 unsigned int vmalloc_t::block_count

number of memory blocks managed by this allocator

Definition at line 101 of file vmalloc.c.

3.41.2.4 addr_t vmalloc_t::end_addr

end address of the address space region managed by the allocator

Definition at line 95 of file vmalloc.c.

3.41.2.5 struct vmalloc_block_t∗ vmalloc_t::free_list

list of completely or partially free blocks

Definition at line 107 of file vmalloc.c.

3.41.2.6 addr_t vmalloc_t::init_limit

address up to which blocks have been initialized

Definition at line 98 of file vmalloc.c.

3.41.2.7 addr_t vmalloc_t::start_addr

start address of the address space region managed by the allocator

Definition at line 92 of file vmalloc.c.

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

- kernel/**vmalloc.c**

3.42 x86_cpuid_regs_t Struct Reference

```
#include <hal/x86.h>
```

Data Fields

- **uint32_t eax**
- **uint32_t ebx**
- **uint32_t ecx**
- **uint32_t edx**

3.42.1 Detailed Description

Definition at line 38 of file x86.h.

3.42.2 Field Documentation

3.42.2.1 uint32_t x86_cpuid_regs_t::eax

Definition at line 39 of file x86.h.

Referenced by cpu_detect_features().

3.42.2.2 uint32_t x86_cpuid_regs_t::ebx

Definition at line 40 of file x86.h.

Referenced by cpu_detect_features().

3.42.2.3 uint32_t x86_cpuid_regs_t::ecx

Definition at line 41 of file x86.h.

Referenced by cpu_detect_features().

3.42.2.4 uint32_t x86_cpuid_regs_t::edx

Definition at line 42 of file x86.h.

Referenced by cpu_detect_features().

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

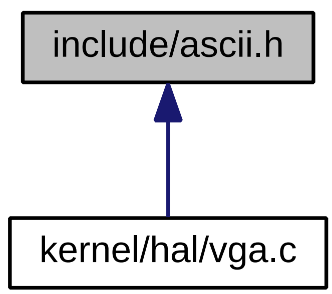
- include/hal/**x86.h**

Chapter 4

File Documentation

4.1 include/ascii.h File Reference

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



Macros

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

4.1.1 Macro Definition Documentation

4.1.1.1 #define CHAR_BS 0x08

Definition at line 35 of file ascii.h.

4.1.1.2 #define CHAR_CR 0x0d

Definition at line 41 of file ascii.h.

4.1.1.3 #define CHAR_HT 0x09

Definition at line 37 of file ascii.h.

4.1.1.4 #define CHAR_LF 0x0a

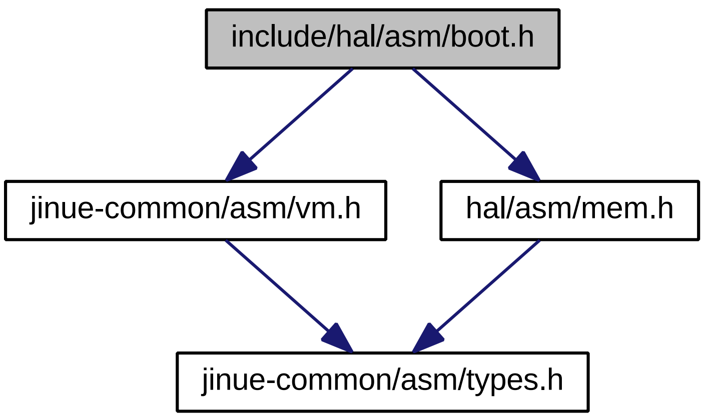
Definition at line 39 of file ascii.h.

4.2 include/boot.h File Reference

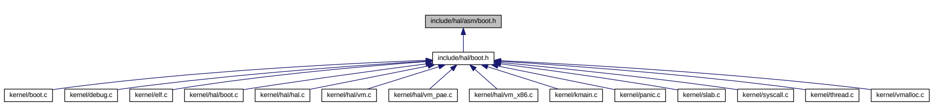
4.3 include/hal/asm/boot.h File Reference

```
#include <jinue-common/asm/vm.h>
#include <hal/asm/mem.h>
```

Include dependency graph for boot.h:



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



Macros

- #define **BOOT_E820_ENTRIES** 0x1e8
- #define **BOOT_SETUP_SECTS** 0x1f1
- #define **BOOT_SYSIZE** 0x1f4
- #define **BOOT_SIGNATURE** 0x1fe
- #define **BOOT_MAGIC** 0xaa55
- #define **BOOT_SETUP** 0x200
- #define **BOOT_SETUP_HEADER** 0x202
- #define **BOOT_SETUP_MAGIC** 0x53726448 /∗ "HdrS", reversed ∗/

- #define **BOOT_RAMDISK_IMAGE** 0x218
- #define **BOOT_RAMDISK_SIZE** 0x21C
- #define **BOOT_CMD_LINE_PTR** 0x228
- #define **BOOT_E820_MAP** 0x2d0
- #define **BOOT_E820_MAP_END** 0xd00
- #define **BOOT_E820_MAP_SIZE** (**BOOT_E820_MAP_END** - **BOOT_E820_MAP**)
- #define **BOOT_SETUP32_ADDR** **MEM_ZONE_DMA16_START**
- #define **BOOT_SETUP32_SIZE** **PAGE_SIZE**
- #define **BOOT_DATA_STRUCT** **BOOT_E820_ENTRIES**
- #define **BOOT_STACK_HEAP_SIZE** (4 ∗ **PAGE_SIZE**)

4.3.1 Macro Definition Documentation

4.3.1.1 #define BOOT_CMD_LINE_PTR 0x228

Definition at line 59 of file boot.h.

4.3.1.2 #define BOOT_DATA_STRUCT BOOT_E820_ENTRIES

Definition at line 71 of file boot.h.

4.3.1.3 #define BOOT_E820_ENTRIES 0x1e8

Definition at line 39 of file boot.h.

4.3.1.4 #define BOOT_E820_MAP 0x2d0

Definition at line 61 of file boot.h.

4.3.1.5 #define BOOT_E820_MAP_END 0xd00

Definition at line 63 of file boot.h.

4.3.1.6 #define BOOT_E820_MAP_SIZE (BOOT_E820_MAP_END - BOOT_E820_MAP)

Definition at line 65 of file boot.h.

4.3.1.7 #define BOOT_MAGIC 0xaa55

Definition at line 47 of file boot.h.

4.3.1.8 #define BOOT_RAMDISK_IMAGE 0x218

Definition at line 55 of file boot.h.

4.3.1.9 #define BOOT_RAMDISK_SIZE 0x21C

Definition at line 57 of file boot.h.

4.3.1.10 #define BOOT_SETUP 0x200

Definition at line 49 of file boot.h.

4.3.1.11 #define BOOT_SETUP32_ADDR MEM_ZONE_DMA16_START

Definition at line 67 of file boot.h.

4.3.1.12 #define BOOT_SETUP32_SIZE PAGE_SIZE

Definition at line 69 of file boot.h.

4.3.1.13 #define BOOT_SETUP_HEADER 0x202

Definition at line 51 of file boot.h.

4.3.1.14 #define BOOT_SETUP_MAGIC 0x53726448 /* "HdrS", reversed */

Definition at line 53 of file boot.h.

Referenced by boot_info_check().

4.3.1.15 #define BOOT_SETUP_SECTS 0x1f1

Definition at line 41 of file boot.h.

4.3.1.16 #define BOOT_SIGNATURE 0x1fe

Definition at line 45 of file boot.h.

4.3.1.17 #define BOOT_STACK_HEAP_SIZE (4 * PAGE_SIZE)

Definition at line 73 of file boot.h.

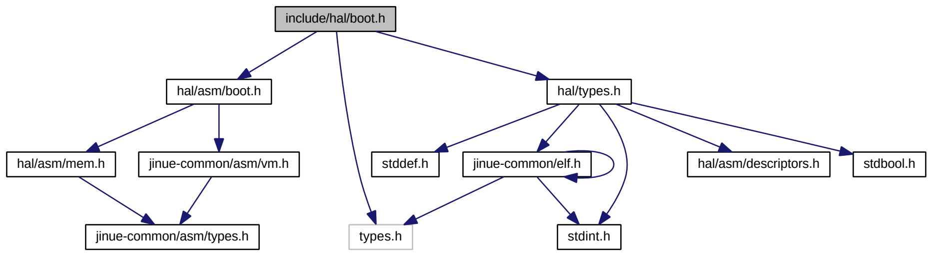
4.3.1.18 #define BOOT_SYSIZE 0x1f4

Definition at line 43 of file boot.h.

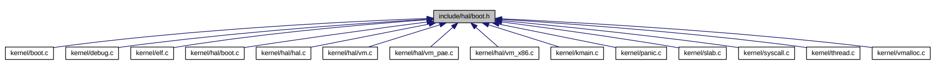
4.4 include/hal/boot.h File Reference

```
#include <types.h>
#include <hal/asm/boot.h>
#include <hal/types.h>
```

Include dependency graph for boot.h:



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



Macros

- #define **boot_heap_alloc**(boot_alloc, t, align) ((t ∗)**boot_heap_alloc_size**(boot_alloc, sizeof(t), align))
    Allocate an object on the boot heap.

Functions

- void **boot_alloc_init** (**boot_alloc_t** ∗boot_alloc, void ∗heap_ptr)
    Initialize the boot allocator.
- void ∗ **boot_heap_alloc_size** (**boot_alloc_t** ∗boot_alloc, **size_t** size, **size_t** align)
    Allocate an object on the boot heap.
- void **boot_heap_push** (**boot_alloc_t** ∗boot_alloc)
    Push the current state of the boot allocator heap.
- void **boot_heap_pop** (**boot_alloc_t** ∗boot_alloc)
    Pop the last pushed boot allocator heap.
- **addr_t boot_page_alloc_early** (**boot_alloc_t** ∗boot_alloc)
    Early page allocation.
- **kern_paddr_t boot_page_frame_alloc** (**boot_alloc_t** ∗boot_alloc)
    Allocate a page frame, that is, a page of physical memory.
- **addr_t boot_vmalloc** (**boot_alloc_t** ∗boot_alloc)
    Allocate a page of address space.
- **addr_t boot_page_alloc** (**boot_alloc_t** ∗boot_alloc)
    Allocate a page in the allocations region of the kernel address space.
- **addr_t boot_page_alloc_image** (**boot_alloc_t** ∗boot_alloc)
    Allocate a page in the image region of the kernel address space.
- **bool boot_info_check** (**bool** panic_on_failure)
- const **boot_info_t** ∗ **get_boot_info** (void)
- void **boot_info_dump** (void)

4.4.1 Macro Definition Documentation

4.4.1.1 #define boot_heap_alloc(*boot_alloc*, *t*, *align*) ((t ∗)**boot_heap_alloc_size**(boot_alloc, sizeof(t), align))

Allocate an object on the boot heap.

This macro is a wrapper for boot_heap_alloc_size that takes a type as the second argument instead of an object size.

Parameters

boot_alloc	the boot allocator state
t	the type of object to allocate
align	the required start address alignment of the object, zero for no constraint

Returns

the allocated object

Definition at line 51 of file boot.h.

Referenced by boot_heap_push(), hal_init(), and vm_pae_create_initial_addr_space().

4.4.2 Function Documentation

4.4.2.1 void boot_alloc_init (**boot_alloc_t** ∗ *boot_alloc*, void ∗ *heap_ptr*)

Initialize the boot allocator.

The boot allocator is used for heap and page allocation during kernel initialization. After this function is called, the boot heap is ready to use (see **boot_heap_alloc()** (p. 66)). However, the page and page frame allocators require additional initialization by the machine-dependent code before they can be used.

Parameters

boot_alloc	the allocator state initialized by this function
heap_ptr	the current top of the boot heap

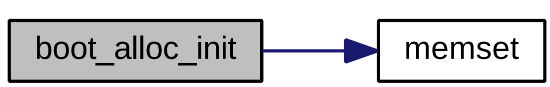
Definition at line 54 of file boot.c.

References boot_alloc_t::heap_ptr, boot_alloc_t::its_early, and memset().

Referenced by kmain().

```
54                                                          {
55     memset(boot_alloc, 0, sizeof(boot_alloc_t));
56     boot_alloc->heap_ptr    = heap_ptr;
57     boot_alloc->its_early   = true;
58     /* TODO handle heap limit. */
59 }
```

Here is the call graph for this function:



4.4.2.2 void∗ boot_heap_alloc_size (boot_alloc_t ∗ *boot_alloc*, size_t *size*, size_t *align*)

Allocate an object on the boot heap.

Callers do not call this function directly but instead use the **boot_heap_alloc()** (p. 66) macro that takes a type as the second argument instead of an object size.

Parameters

boot_alloc	the boot allocator state
size	the size of the object to allocate, in bytes
align	the required start address alignment of the object, zero for no constraint

Returns

the allocated object

Definition at line 73 of file boot.c.

References ALIGN_END_PTR, and boot_alloc_t::heap_ptr.

```
73                                                                                   {
74      if(align != 0) {
75          boot_alloc->heap_ptr = ALIGN_END_PTR(boot_alloc->heap_ptr, align);
76      }
77
78      void *object             = boot_alloc->heap_ptr;
79      boot_alloc->heap_ptr     = (char *)boot_alloc->heap_ptr + size;
80
81      return object;
82  }
```

4.4.2.3 void boot_heap_pop (boot_alloc_t ∗ *boot_alloc*)

Pop the last pushed boot allocator heap.

This function frees all heap allocations performed since the matching call to **boot_heap_push()** (p. 68).

Parameters

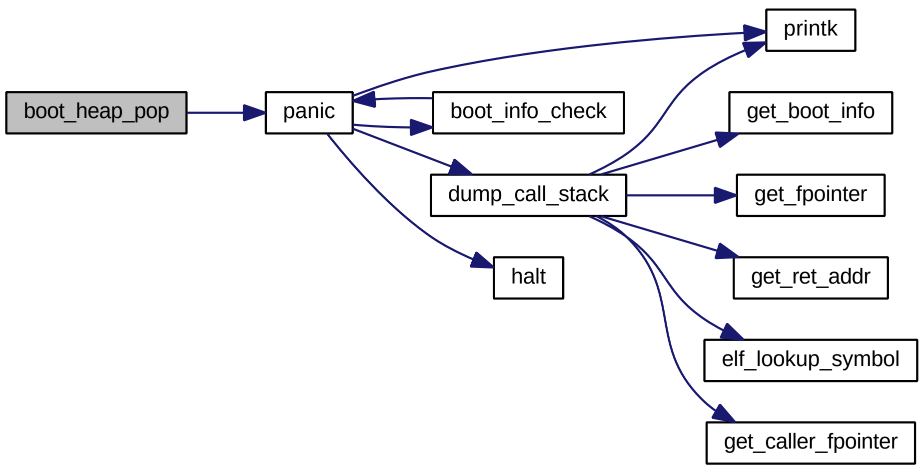
boot_alloc	the boot allocator state

Definition at line 112 of file boot.c.

References boot_alloc_t::heap_ptr, boot_alloc_t::heap_pushed_state, boot_heap_pushed_state::next, NULL, and panic().

```
112                                                {
113      if(boot_alloc->heap_pushed_state == NULL) {
114          panic("No more boot heap pushed state to pop.");
115      }
116
117      boot_alloc->heap_ptr             = boot_alloc->heap_pushed_state;
118      boot_alloc->heap_pushed_state    = boot_alloc->heap_pushed_state->next;
119  }
```

Here is the call graph for this function:



4.4.2.4 void boot_heap_push (boot_alloc_t ∗ *boot_alloc*)

Push the current state of the boot allocator heap.

All heap allocations performed after calling this function are freed by the matching call to **boot_heap_pop()** (p. 67). This function can be called multiple times before calling **boot_heap_pop()** (p. 67). Heap states pushed by this function are popped by **boot_heap_pop()** (p. 67)() in the reverse order they were pushed.

Parameters

boot_alloc	the boot allocator state

Definition at line 95 of file boot.c.

References boot_heap_alloc, boot_alloc_t::heap_pushed_state, and boot_heap_pushed_state::next.

```
95                                              {
96      struct boot_heap_pushed_state *pushed_state =
97              boot_heap_alloc(boot_alloc, struct boot_heap_pushed_state, 0);
98
99      pushed_state->next                = boot_alloc->heap_pushed_state;
100      boot_alloc->heap_pushed_state    = pushed_state;
101 }
```

4.4.2.5 bool boot_info_check (bool *panic_on_failure*)

Definition at line 42 of file boot.c.

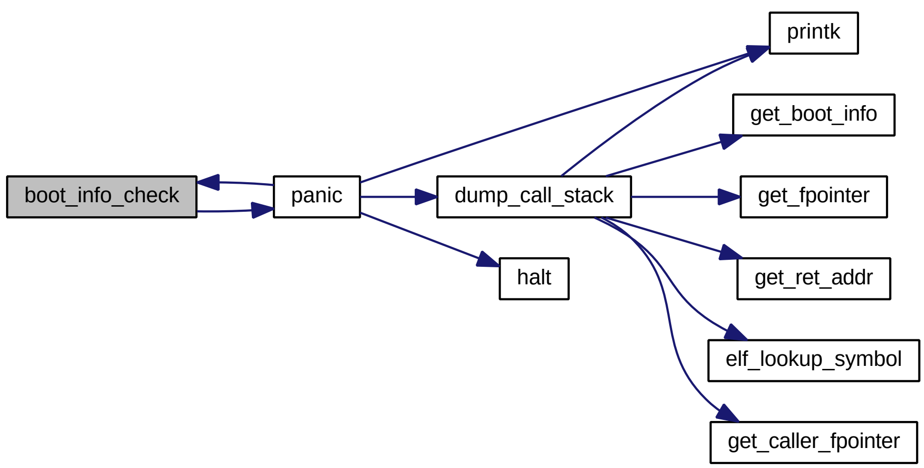
References BOOT_SETUP_MAGIC, boot_info_t::image_start, boot_info_t::kernel_start, NULL, page_offset_of, panic(), and boot_info_t::setup_signature.

Referenced by kmain(), and panic().

```
42                                            {
43      const char *error_description = NULL;
44
45      /* This data structure is accessed early during the boot process, before
46       * paging is enabled. What this means is that, if boot_info is NULL and we
47       * dereference it, it does *not* cause a page fault or any other CPU
```

```
     * exception. */
    if(boot_info == NULL) {
        error_description = "Boot information structure pointer is NULL.";
    }
    else if(boot_info->setup_signature != BOOT_SETUP_MAGIC) {
        error_description = "Bad setup header signature.";
    }
    else if(page_offset_of(boot_info->image_start) != 0) {
        error_description = "Bad image alignment.";
    }
    else if(page_offset_of(boot_info->kernel_start) != 0) {
        error_description = "Bad kernel alignment.";
    }
    else {
        return true;
    }

    if(panic_on_failure) {
        panic(error_description);
    }

    return false;
}
```

Here is the call graph for this function:



4.4.2.6 void boot_info_dump (void)

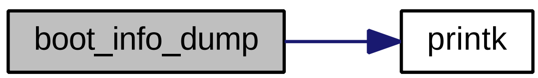
Definition at line 76 of file boot.c.

References boot_info_t::boot_end, boot_info_t::boot_heap, boot_info_t::e820_entries, boot_info_t::e820_map, boot-_info_t::image_start, boot_info_t::image_top, boot_info_t::kernel_size, boot_info_t::kernel_start, boot_info_t::page_-directory, boot_info_t::page_table, printk(), boot_info_t::proc_size, boot_info_t::proc_start, and boot_info_t::setup_-signature.

```
76                        {
77     printk("Boot information structure:\n");
78     printk("    kernel_start    %x  %u\n", boot_info->kernel_start    , boot_info->
    kernel_start    );
79     printk("    kernel_size     %x  %u\n", boot_info->kernel_size     , boot_info->
    kernel_size     );
80     printk("    proc_start      %x  %u\n", boot_info->proc_start      , boot_info->
    proc_start      );
81     printk("    proc_size       %x  %u\n", boot_info->proc_size       , boot_info->
    proc_size       );
82     printk("    image_start     %x  %u\n", boot_info->image_start     , boot_info->
    image_start     );
```

```
    printk("    image_top       %x  %u\n", boot_info->image_top      , boot_info->image_top       );
    printk("    e820_entries    %x  %u\n", boot_info->e820_entries   , boot_info->e820_entries    );
    printk("    e820_map        %x  %u\n", boot_info->e820_map       , boot_info->e820_map        );
    printk("    boot_heap       %x  %u\n", boot_info->boot_heap      , boot_info->boot_heap       );
    printk("    boot_end        %x  %u\n", boot_info->boot_end       , boot_info->boot_end        );
    printk("    page_table      %x  %u\n", boot_info->page_table     , boot_info->page_table      );
    printk("    page_directory  %x  %u\n", boot_info->page_directory , boot_info->page_directory  );
    printk("    setup_signature %x  %u\n", boot_info->setup_signature, boot_info->setup_signature );
}
```

Here is the call graph for this function:



4.4.2.7 addr_t boot_page_alloc (boot_alloc_t * *boot_alloc*)

Allocate a page in the allocations region of the kernel address space.

The physical memory is allocated just after the kernel image and other initialization-time allocations by calling **boot_-page_frame_alloc()** (p. 73) whereas the address space page is allocated in the allocations region by calling **vmalloc()** (p. 275).

If either of these two conditions is met, you must use boot_pgalloc_image() instead of this function: 1) The address space page allocator has not not yet been initialized by calling **vmalloc_init()** (p. 275); or 2) It is necessary to allocate multiple contiguous pages.

Parameters

boot_alloc	the boot allocator state

Returns

address of allocated page

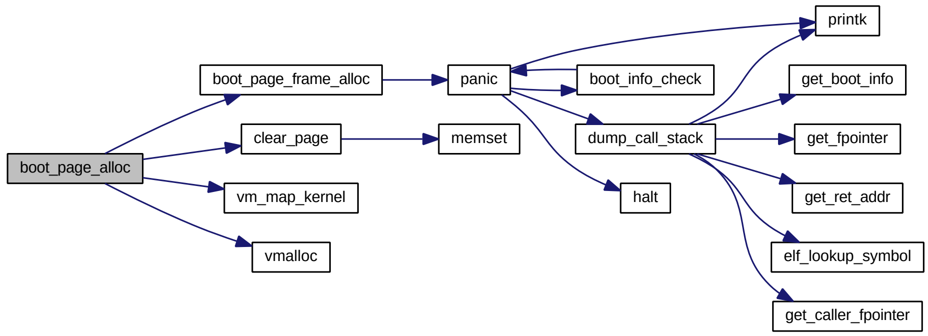
Definition at line 282 of file boot.c.

References boot_page_frame_alloc(), clear_page(), VM_FLAG_READ_WRITE, vm_map_kernel(), and vmalloc().

Referenced by hal_init(), slab_cache_init(), and thread_create_boot().

```
282                                                 {
283     kern_paddr_t paddr  = boot_page_frame_alloc(boot_alloc);
284     addr_t vaddr        = vmalloc();
285 
286     vm_map_kernel(vaddr, paddr, VM_FLAG_READ_WRITE);
287 
288     /* This newly allocated page may have data left from a previous boot which
289      * may contain sensitive information. Let's clear it. */
290     clear_page(vaddr);
291 
292     return vaddr;
293 }
```

Here is the call graph for this function:



4.4.2.8 addr_t boot_page_alloc_early (boot_alloc_t * *boot_alloc*)

Early page allocation.

When the kernel is first entered, the setup code has set up temporary page tables that map a contiguous region of physical memory (RAM) that contains the kernel image at KLIMIT. The setup code itself allocates a few pages, notably for the temporary page tables and for the boot stack and heap. These pages are allocated sequentially just after the kernel image.

This function allocates pages sequentially following the kernel image and the setup code allocations. It is meant to be called early in the initialization process, while the temporary page tables set up by the setup code are still being used, which means before the kernel switches to the initial address space it sets up.

Because the page tables set up by the setup code are being used, there is a fixed relation between the virtual address of the pages allocated by this function and the physical address of the underlying page frames. This relation is expressed by the **EARLY_PTR_TO_PHYS_ADDR()** (p. 142) macro.

This function must not be called once the kernel has switched away from the page tables set up by the setup code to the initial address space it has set up itself. This function checks for this and triggers a kernel panic if it happens.

Parameters

boot_alloc	the boot allocator state

Returns

address of allocated page

Definition at line 150 of file boot.c.

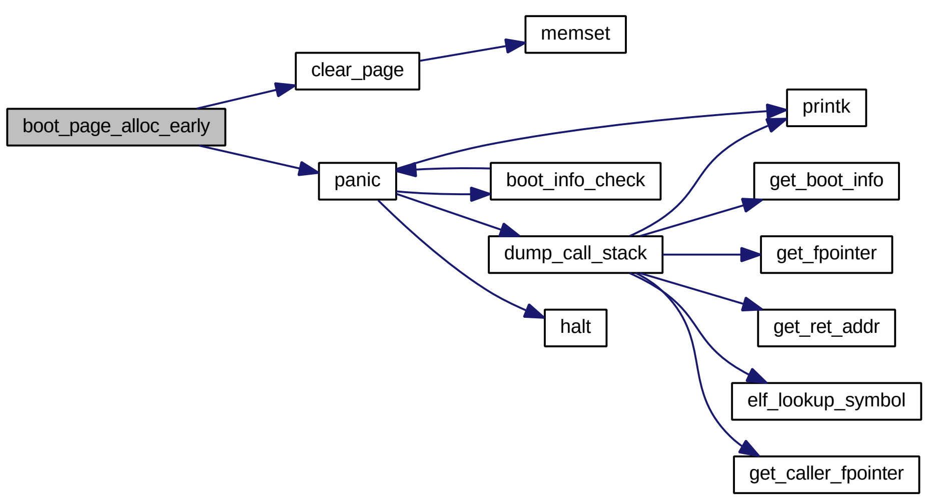
References clear_page(), EARLY_PTR_TO_PHYS_ADDR, boot_alloc_t::its_early, boot_alloc_t::kernel_paddr_limit, boot_alloc_t::kernel_paddr_top, boot_alloc_t::kernel_vm_limit, boot_alloc_t::kernel_vm_top, NULL, PAGE_MASK, P-AGE_SIZE, and panic().

Referenced by vm_init_initial_page_directory(), vm_pae_create_initial_addr_space(), and vm_x86_create_initial_addr-_space().

```
150                                                  {
151     /* Preconditions */
152     if(! boot_alloc->its_early) {
153         panic("boot_pgalloc_early() called too late");
154     }
155
156     if(boot_alloc->kernel_vm_top == NULL) {
```

```
        panic("boot_pgalloc_early(): allocator is uninitialized");
    }

    if(((uintptr_t)boot_alloc->kernel_vm_top & PAGE_MASK) != 0) {
        panic("boot_pgalloc_early(): bad kernel region top VM address alignment");
    }

    if(boot_alloc->kernel_paddr_top != EARLY_PTR_TO_PHYS_ADDR(boot_alloc->kernel_vm_top)) {
        panic("boot_pgalloc_early(): inconsistent allocator state");
    }

    /* address of allocated page */
    addr_t allocated_page = boot_alloc->kernel_vm_top;

    /* Update allocator state.
     *
     * In this early allocator function that is called while the temporary page
     * tables set up by the setup code are still being used, there is a fixed
     * relationship between virtual and physical addresses. */
    boot_alloc->kernel_vm_top      = allocated_page + PAGE_SIZE;
    boot_alloc->kernel_paddr_top   = boot_alloc->kernel_paddr_top + PAGE_SIZE;

    /* Check updated state against allocation limits. */
    if(boot_alloc->kernel_vm_top > boot_alloc->kernel_vm_limit) {
        panic("vmalloc_early(): kernel address space exhausted");
    }

    if(boot_alloc->kernel_paddr_top > boot_alloc->kernel_paddr_limit) {
        panic("vmalloc_early(): available memory exhausted");
    }

    /* This newly allocated page may have data left from a previous boot which
     * may contain sensitive information. Let's clear it. */
    clear_page(allocated_page);

    /* Post-condition */
    if(boot_alloc->kernel_paddr_top != EARLY_PTR_TO_PHYS_ADDR(boot_alloc->kernel_vm_top)) {
        panic("boot_pgalloc_early(): inconsistent allocator state on return");
    }

    return allocated_page;
}
```

Here is the call graph for this function:



4.4.2.9 addr_t boot_page_alloc_image (boot_alloc_t * *boot_alloc*)

Allocate a page in the image region of the kernel address space.

Since the size of the image region is limited, use boot_pgalloc() instead of this function whenever possible.

The difference between this function and boot_pgalloc() is that the address space page is allocated by this function using **boot_vmalloc()** (p. 74) instead of **vmalloc()** (p. 275). Pages allocated by subsequent calls to this function are allocated sequentially.

Parameters

boot_alloc	the boot allocator state

Returns

address of allocated page

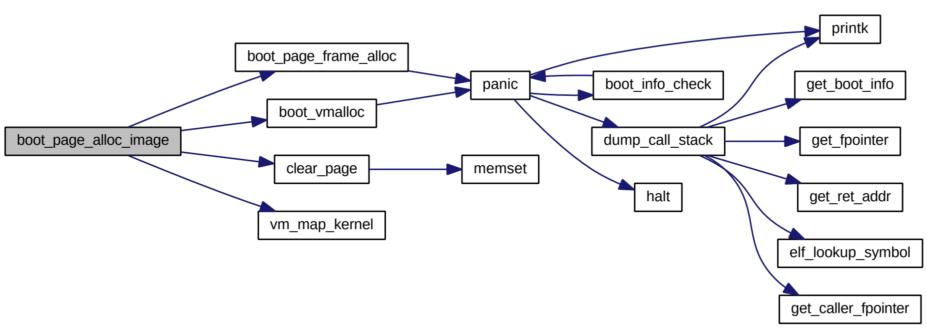
Definition at line 310 of file boot.c.

References boot_page_frame_alloc(), boot_vmalloc(), clear_page(), VM_FLAG_READ_WRITE, and vm_map_kernel().

Referenced by hal_init().

```
310                                                        {
311     kern_paddr_t paddr  = boot_page_frame_alloc(boot_alloc);
312     addr_t vaddr        = boot_vmalloc(boot_alloc);
313
314     vm_map_kernel(vaddr, paddr, VM_FLAG_READ_WRITE);
315
316     /* This newly allocated page may have data left from a previous boot which
317      * may contain sensitive information. Let's clear it. */
318     clear_page(vaddr);
319
320     return vaddr;
321 }
```

Here is the call graph for this function:



4.4.2.10 kern_paddr_t boot_page_frame_alloc (boot_alloc_t ∗ *boot_alloc*)

Allocate a page frame, that is, a page of physical memory.

The allocated page frame is not mapped anywhere. For a mapped page, call boot_pgalloc() or boot_pgalloc_image() instead;

Parameters

boot_alloc	the boot allocator state

Returns

physical address of allocated page frame

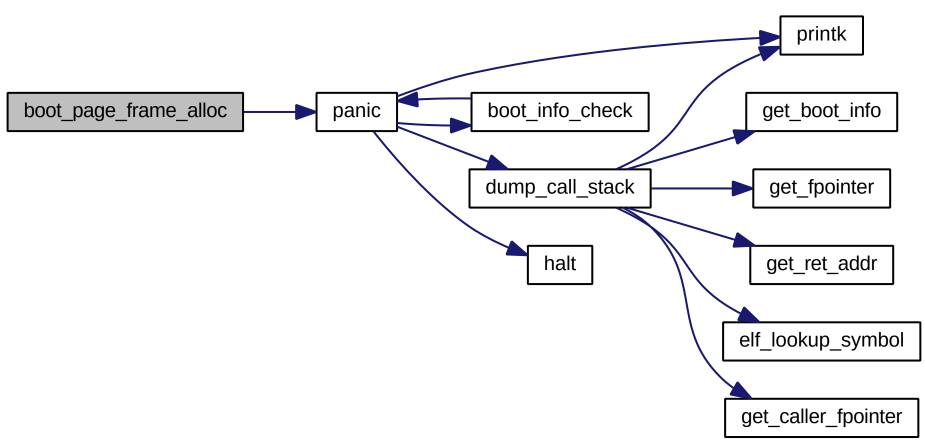
Definition at line 210 of file boot.c.

References boot_alloc_t::its_early, boot_alloc_t::kernel_paddr_limit, boot_alloc_t::kernel_paddr_top, PAGE_SIZE, and panic().

Referenced by boot_page_alloc(), boot_page_alloc_image(), elf_load(), and elf_setup_stack().

```
210                                                              {
211     if(boot_alloc->its_early) {
212         panic("boot_pfalloc() called too soon");
213     }
214 
215     /* address of allocated page  */
216     kern_paddr_t paddr = boot_alloc->kernel_paddr_top;
217 
218     /* Update allocator state. */
219     boot_alloc->kernel_paddr_top = paddr + PAGE_SIZE;
220 
221     /* Check bounds. */
222     if(boot_alloc->kernel_paddr_top > boot_alloc->kernel_paddr_limit) {
223         panic("pfalloc_boot(): available memory exhausted");
224     }
225 
226     return paddr;
227 }
```

Here is the call graph for this function:



4.4.2.11 addr_t boot_vmalloc (boot_alloc_t ∗ *boot_alloc*)

Allocate a page of address space.

No memory is mapped to the allocated page. The page is allocated from the image region of the kernel address space, just after the kernel image and other initialization-time page allocations. This function allocates pages sequentially.

Parameters

boot_alloc	the boot allocator state

Returns

address of allocated page

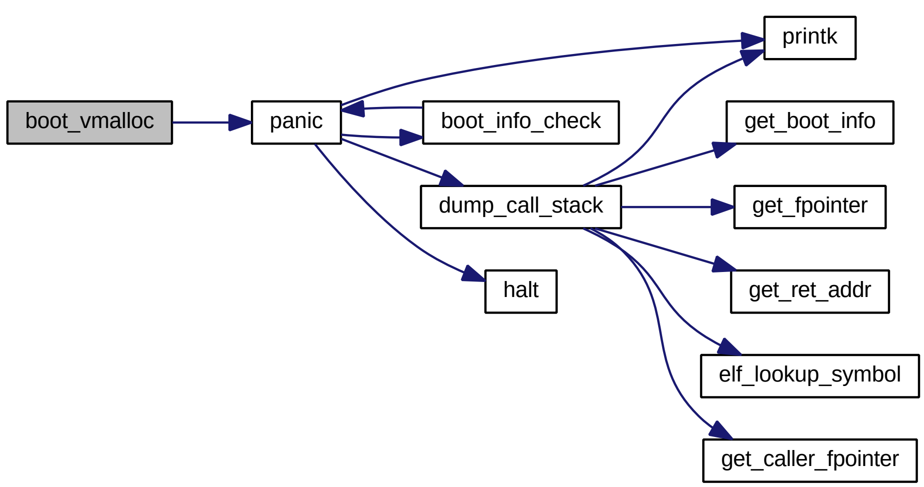
Definition at line 241 of file boot.c.

References boot_alloc_t::its_early, boot_alloc_t::kernel_vm_limit, boot_alloc_t::kernel_vm_top, NULL, PAGE_SIZE, and panic().

Referenced by boot_page_alloc_image(), and vm_boot_init().

```
241                                                {
242     if(boot_alloc->its_early) {
243         panic("boot_vmalloc() called too soon");
244     }
245
246     if(boot_alloc->kernel_vm_top == NULL) {
247         panic("boot_pgalloc_early(): allocator is uninitialized");
248     }
249
250     /* address of allocated page */
251     addr_t page = boot_alloc->kernel_vm_top;
252
253     /* Update allocator state. */
254     boot_alloc->kernel_vm_top = page + PAGE_SIZE;
255
256     /* Check bounds. */
257     if(boot_alloc->kernel_vm_top > boot_alloc->kernel_vm_limit) {
258         panic("vmalloc_boot(): kernel address space exhausted");
259     }
260
261     return page;
262 }
```

Here is the call graph for this function:



4.4.2.12 const boot_info_t∗ get_boot_info (void)

Definition at line 72 of file boot.c.

References boot_info.

Referenced by dispatch_syscall(), dump_call_stack(), and kmain().

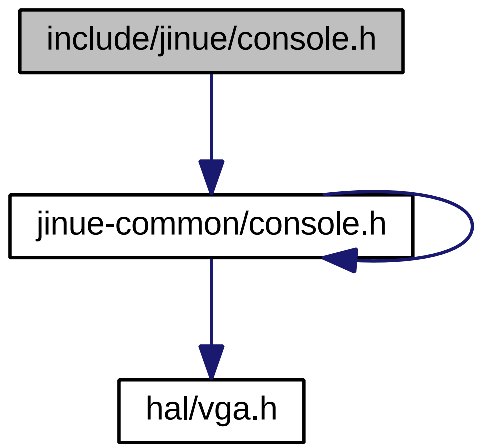
```
                                            {
       return boot_info;
   }
```

4.5 include/console.h File Reference

4.6 include/jinue/console.h File Reference

`#include <jinue-common/console.h>`

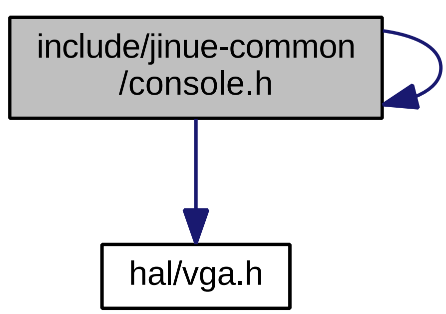
Include dependency graph for console.h:



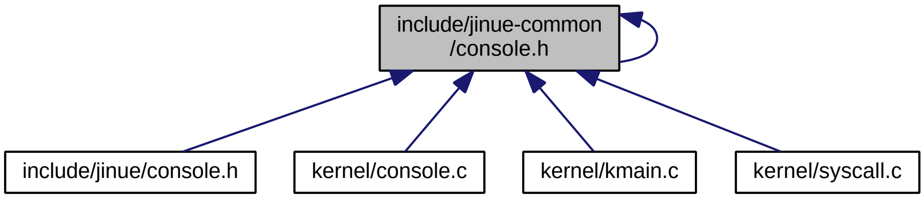
4.7 include/jinue-common/console.h File Reference

`#include <jinue-common/console.h>`
`#include <hal/vga.h>`

Include dependency graph for console.h:



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



Macros

- #define **CONSOLE_SERIAL_IOPORT SERIAL_COM1_IOPORT**
- #define **CONSOLE_SERIAL_BAUD_RATE** 115200
- #define **CONSOLE_DEFAULT_COLOR** 0x0a /∗ VGA_COLOR_BRIGHTGREEN ∗/

Functions

- void **console_init** (void)
- void **console_printn** (const char ∗message, unsigned int n, int colour)
- void **console_putc** (char c, int colour)
- void **console_print** (const char ∗message, int colour)

4.7.1 Macro Definition Documentation

4.7.1.1 #define CONSOLE_DEFAULT_COLOR 0x0a /∗ VGA_COLOR_BRIGHTGREEN ∗/

Definition at line 35 of file console.h.

Referenced by dispatch_syscall().

4.7.1.2 #define CONSOLE_SERIAL_BAUD_RATE 115200

Definition at line 40 of file console.h.

Referenced by console_init().

4.7.1.3 #define CONSOLE_SERIAL_IOPORT SERIAL_COM1_IOPORT

Definition at line 38 of file console.h.

Referenced by console_init(), console_printn(), and console_putc().

4.7.2 Function Documentation

4.7.2.1 void console_init (void)

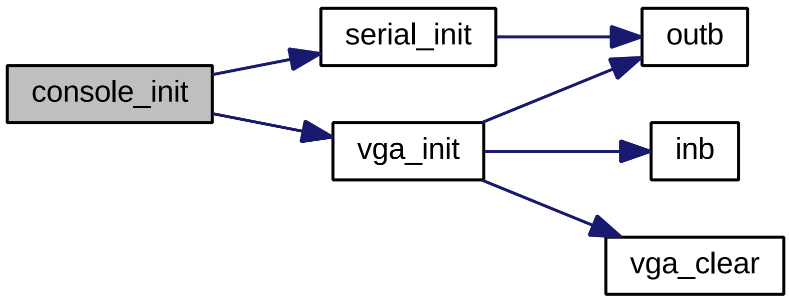
Definition at line 38 of file console.c.

References CONSOLE_SERIAL_BAUD_RATE, CONSOLE_SERIAL_IOPORT, serial_init(), and vga_init().

Referenced by kmain().

```
38                        {
39     vga_init();
40     serial_init(CONSOLE_SERIAL_IOPORT, CONSOLE_SERIAL_BAUD_RATE);
41 }
```

Here is the call graph for this function:



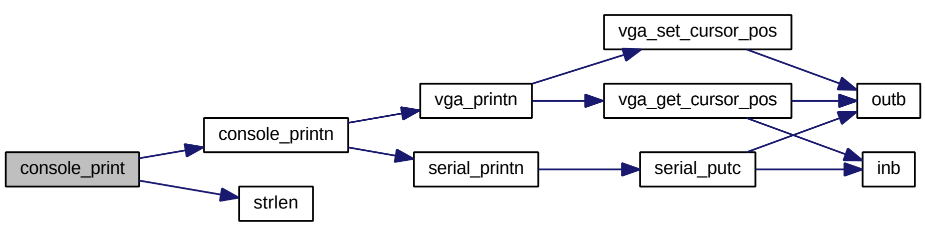
4.7.2.2 void console_print (const char ∗ *message*, int *colour*)

Definition at line 53 of file console.c.

References console_printn(), and strlen().

```
53                                              {
54     console_printn(message, strlen(message), colour);
55 }
```

Here is the call graph for this function:



4.7.2.3 void console_printn (const char ∗ *message*, unsigned int *n*, int *colour*)

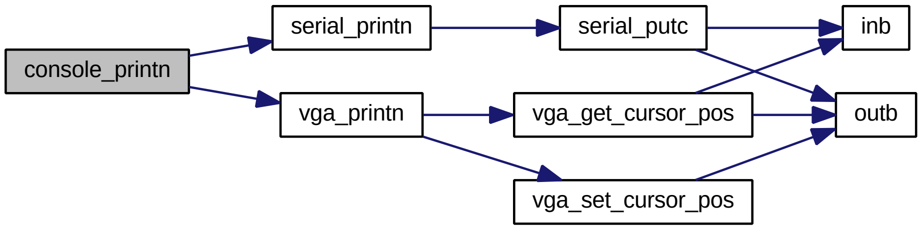
Definition at line 43 of file console.c.

References CONSOLE_SERIAL_IOPORT, serial_printn(), and vga_printn().

Referenced by console_print(), and dispatch_syscall().

```
43                                                               {
44     vga_printn(message, n, colour);
45     serial_printn(CONSOLE_SERIAL_IOPORT, message, n);
46 }
```

Here is the call graph for this function:



4.7.2.4 void console_putc (char *c*, int *colour*)

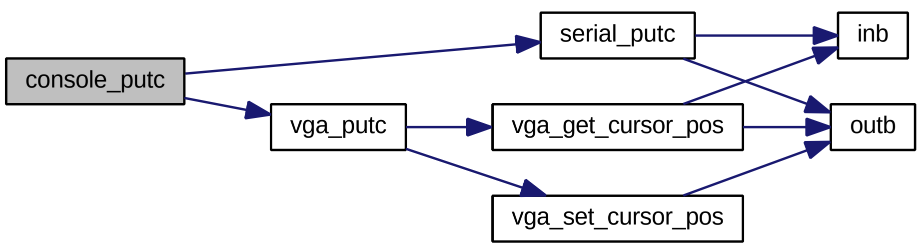
Definition at line 48 of file console.c.

References CONSOLE_SERIAL_IOPORT, serial_putc(), and vga_putc().

Referenced by dispatch_syscall().

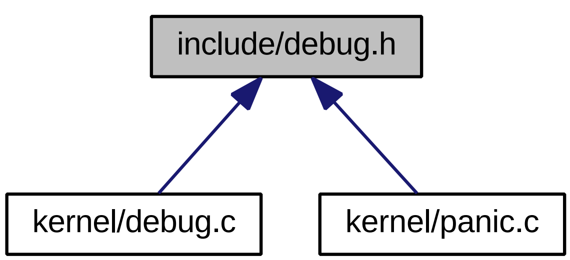
```
48                                      {
49      vga_putc(c, colour);
50      serial_putc(CONSOLE_SERIAL_IOPORT, c);
51 }
```

Here is the call graph for this function:



4.8 include/debug.h File Reference

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



Functions

- void **dump_call_stack** (void)

4.8.1 Function Documentation

4.8.1.1 void dump_call_stack (void)

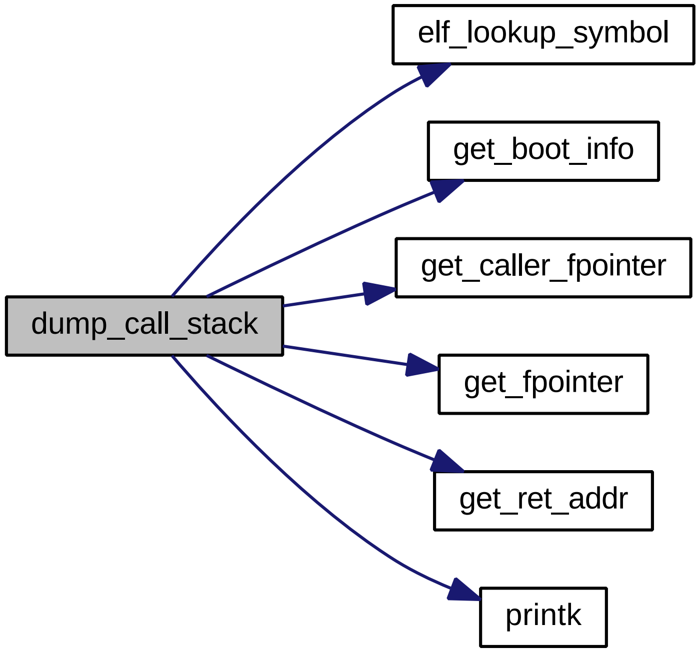
Definition at line 41 of file debug.c.

References elf_symbol_t::addr, boot_info, elf_lookup_symbol(), get_boot_info(), get_caller_fpointer(), get_fpointer(), get_ret_addr(), boot_info_t::kernel_start, elf_symbol_t::name, NULL, printk(), and STT_FUNCTION.

Referenced by panic().

```
41                          {
42     addr_t               fptr;
43 
44     const boot_info_t *boot_info = get_boot_info();
45 
46     printk("Call stack dump:\n");
47 
48     fptr = get_fpointer();
49 
50     while(fptr != NULL) {
51         addr_t return_addr = get_ret_addr(fptr);
52         if(return_addr == NULL) {
53             break;
54         }
55 
56         /* assume e8 xx xx xx xx for call instruction encoding */
57         return_addr -= 5;
58 
59         elf_symbol_t symbol;
60         int retval = elf_lookup_symbol(
61                 boot_info->kernel_start,
62                 (Elf32_Addr)return_addr,
63                 STT_FUNCTION,
64                 &symbol);
65 
66         if(retval < 0) {
67             printk("\t0x%x (unknown)\n", return_addr);
68         }
69         else {
70             const char *name = symbol.name;
71 
72             if(name == NULL) {
73                 name = "[unknown]";
74             }
75 
76             printk(
77                     "\t0x%x (%s+%u)\n",
78                     return_addr,
79                     name,
80                     return_addr - symbol.addr);
81         }
82 
83         fptr = get_caller_fpointer(fptr);
84     }
85 }
```

Here is the call graph for this function:

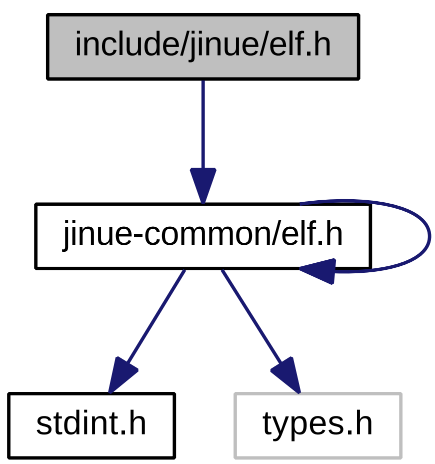


4.9 include/elf.h File Reference

4.10 include/jinue/elf.h File Reference

`#include <jinue-common/elf.h>`

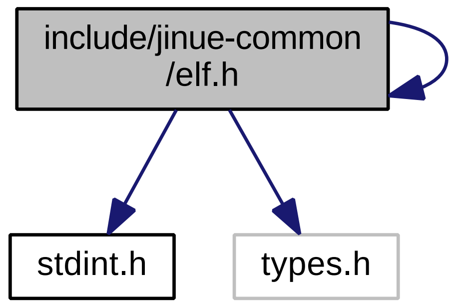
Include dependency graph for elf.h:



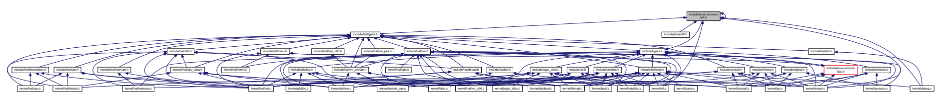
4.11 include/jinue-common/elf.h File Reference

```
#include <stdint.h>
#include <jinue-common/elf.h>
#include <types.h>
```

Include dependency graph for elf.h:



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



Data Structures

- struct **elf_info_t**
- struct **elf_symbol_t**
- struct **Elf32_Ehdr**
- struct **Elf32_Phdr**
- struct **Elf32_Shdr**
- struct **Elf32_Sym**
- struct **Elf32_auxv_t**

Macros

- #define **EI_MAG0** 0

  Index of file identification - byte 0.
- #define **EI_MAG1** 1

  Index of file identification - byte 1.
- #define **EI_MAG2** 2

  Index of file identification - byte 2.
- #define **EI_MAG3** 3

  Index of file identification - byte 3.
- #define **EI_CLASS** 4

  File class.
- #define **EI_DATA** 5

  Data encoding.

- #define **EI_VERSION** 6
  File version.
- #define **EI_PAD** 7
  Start of padding bytes.
- #define **EI_NIDENT** 16
  size of e_ident[]
- #define **ELF_MAGIC0** 0x7f
  File identification - byte 0 (0x7f)
- #define **ELF_MAGIC1** 'E'
  File identification - byte 1 ('E')
- #define **ELF_MAGIC2** 'L'
  File identification - byte 2 ('L')
- #define **ELF_MAGIC3** 'F'
  File identification - byte 3 ('F')
- #define **EM_NONE** 0
  No machine.
- #define **EM_SPARC** 2
  SPARC.
- #define **EM_386** 3
  Intel 80386.
- #define **EM_MIPS** 8
  MIPS RS3000.
- #define **EM_SPARC32PLUS** 18
  Enhanced instruction set SPARC.
- #define **EM_ARM** 40
  32-bit ARM
- #define **EM_X86_64** 62
  AMD64/X86-64.
- #define **EM_OPENRISC** 92
  OpenRISC 32-bit embedded processor.
- #define **EM_ALTERA_NIOS2** 113
  Altera Nios 2 32-bit soft processor.
- #define **EM_AARCH64** 183
  64-bit AARCH64 ARM
- #define **EM_MICROBLAZE** 189
  Xilinx MicroBlaze 32-bit soft processor.
- #define **ET_NONE** 0
  No file type.
- #define **ET_REL** 1
  Relocatable file.
- #define **ET_EXEC** 2
  Executable file.
- #define **ET_DYN** 3
  Shared object file.
- #define **ET_CORE** 4
  Core file.
- #define **ELFCLASSNONE** 0

Invalid class.

- #define **ELFCLASS32** 1

  32-bit objects
- #define **ELFCLASS64** 2

  64-bit objects
- #define **ELFDATANONE** 0

  Invalid data encoding.
- #define **ELFDATA2LSB** 1

  Little-endian.
- #define **ELFDATA2MSB** 2

  Big-endian.
- #define **PT_NULL** 0

  Unused entry.
- #define **PT_LOAD** 1

  Loadable segment.
- #define **PT_DYNAMIC** 2

  Dynamic linking information.
- #define **PT_INTERP** 3

  Path to program interpreter.
- #define **PT_NOTE** 4

  Location and size of notes.
- #define **PT_SHLIB** 5

  Unspecified semantics.
- #define **PT_PHDR** 6

  Program header table.
- #define **SHT_NULL** 0

  Inactive section.
- #define **SHT_PROGBITS** 1

  Program data.
- #define **SHT_SYMTAB** 2

  Symbol table.
- #define **SHT_STRTAB** 3

  String table.
- #define **SHT_RELA** 4

  Relocations with addends.
- #define **SHT_HASH** 5

  Symbol hash table.
- #define **SHT_DYNAMIC** 6

  Information for dynamic linking.
- #define **SHT_NOTE** 7

  Notes section.
- #define **SHT_NOBITS** 8

  Section without data (.bss)
- #define **SHT_REL** 9

  Relocations without addends.
- #define **SHT_SHLIB** 10

  Reserved, unspecified semantic, not ABI compliant.

- #define **SHT_DYNSYM** 11

  Dynamic symbols table.
- #define **STB_LOCAL** 0

  Local binding.
- #define **STB_GLOBAL** 1

  Global binding.
- #define **STB_WEAK** 2

  Weak binding.
- #define **STT_NOTYPE** 0

  Unspecified type.
- #define **STT_OBJECT** 1

  Data object.
- #define **STT_FUNCTION** 2

  Function or other executable code.
- #define **STT_SECTION** 3

  Section symbol.
- #define **STT_FILE** 4

  Source file.
- #define **ELF32_ST_BIND**(i) ((i) >> 4)
- #define **ELF32_ST_TYPE**(i) ((i) & 0xf)
- #define **STN_UNDEF** 0

  Undefined symbol index.
- #define **PF_R** (1 << 2)
- #define **PF_W** (1 << 1)
- #define **PF_X** (1 << 0)
- #define **AT_NULL** 0

  Last entry.
- #define **AT_IGNORE** 1

  Ignore entry.
- #define **AT_EXECFD** 2

  Program file descriptor.
- #define **AT_PHDR** 3

  Program headers address.
- #define **AT_PHENT** 4

  Size of program header entry.
- #define **AT_PHNUM** 5

  Number of program header entries.
- #define **AT_PAGESZ** 6

  Page size.
- #define **AT_BASE** 7

  Base address.
- #define **AT_FLAGS** 8

  Flags.
- #define **AT_ENTRY** 9

  Program entry point.
- #define **AT_DCACHEBSIZE** 10

  Data cache block size.

- #define **AT_ICACHEBSIZE** 11

  Instruction cache block size.
- #define **AT_UCACHEBSIZE** 12

  Unified cache block size.
- #define **AT_STACKBASE** 13

  Stack base address for main thread.
- #define **AT_HWCAP** 16

  Machine-dependent processor feature flags.
- #define **AT_HWCAP2** 26

  More machine-dependent processor feature flags.
- #define **AT_SYSINFO_EHDR** 33

  Address of vDSO.

Typedefs

- typedef **uint32_t Elf32_Addr**
- typedef **uint16_t Elf32_Half**
- typedef **uint32_t Elf32_Off**
- typedef **int32_t Elf32_Sword**
- typedef **uint32_t Elf32_Word**
- typedef **Elf32_auxv_t auxv_t**

Functions

- void **elf_check** (**Elf32_Ehdr** ∗elf)
- void **elf_load** (**elf_info_t** ∗info, **Elf32_Ehdr** ∗elf, **addr_space_t** ∗addr_space, **boot_alloc_t** ∗boot_alloc)
- void **elf_setup_stack** (**elf_info_t** ∗info, **boot_alloc_t** ∗boot_alloc)
- int **elf_lookup_symbol** (const **Elf32_Ehdr** ∗elf_header, **Elf32_Addr** addr, int type, **elf_symbol_t** ∗result)

4.11.1 Macro Definition Documentation

4.11.1.1 #define AT_BASE 7

Base address.

Definition at line 339 of file elf.h.

4.11.1.2 #define AT_DCACHEBSIZE 10

Data cache block size.

Definition at line 348 of file elf.h.

4.11.1.3 #define AT_ENTRY 9

Program entry point.

Definition at line 345 of file elf.h.

Referenced by elf_setup_stack().

4.11.1.4 #define AT_EXECFD 2

Program file descriptor.

Definition at line 324 of file elf.h.

4.11.1.5 #define AT_FLAGS 8

Flags.

Definition at line 342 of file elf.h.

4.11.1.6 #define AT_HWCAP 16

Machine-dependent processor feature flags.

Definition at line 360 of file elf.h.

4.11.1.7 #define AT_HWCAP2 26

More machine-dependent processor feature flags.

Definition at line 363 of file elf.h.

4.11.1.8 #define AT_ICACHEBSIZE 11

Instruction cache block size.

Definition at line 351 of file elf.h.

4.11.1.9 #define AT_IGNORE 1

Ignore entry.

Definition at line 321 of file elf.h.

4.11.1.10 #define AT_NULL 0

Last entry.

Definition at line 318 of file elf.h.

Referenced by elf_setup_stack().

4.11.1.11 #define AT_PAGESZ 6

Page size.

Definition at line 336 of file elf.h.

Referenced by elf_setup_stack().

4.11.1.12 #define AT_PHDR 3

Program headers address.

Definition at line 327 of file elf.h.

Referenced by elf_setup_stack().

4.11.1.13 #define AT_PHENT 4

Size of program header entry.

Definition at line 330 of file elf.h.

Referenced by elf_setup_stack().

4.11.1.14 #define AT_PHNUM 5

Number of program header entries.

Definition at line 333 of file elf.h.

Referenced by elf_setup_stack().

4.11.1.15 #define AT_STACKBASE 13

Stack base address for main thread.

Definition at line 357 of file elf.h.

Referenced by elf_setup_stack().

4.11.1.16 #define AT_SYSINFO_EHDR 33

Address of vDSO.

Definition at line 366 of file elf.h.

4.11.1.17 #define AT_UCACHEBSIZE 12

Unified cache block size.

Definition at line 354 of file elf.h.

4.11.1.18 #define EI_CLASS 4

File class.

Definition at line 50 of file elf.h.

Referenced by elf_check().

4.11.1.19 #define EI_DATA 5

Data encoding.

Definition at line 53 of file elf.h.

Referenced by elf_check().

4.11.1.20 #define EI_MAG0 0

Index of file identification - byte 0.

Definition at line 38 of file elf.h.

Referenced by elf_check().

4.11.1.21 #define EI_MAG1 1

Index of file identification - byte 1.

Definition at line 41 of file elf.h.

Referenced by elf_check().

4.11.1.22 #define EI_MAG2 2

Index of file identification - byte 2.

Definition at line 44 of file elf.h.

Referenced by elf_check().

4.11.1.23 #define EI_MAG3 3

Index of file identification - byte 3.

Definition at line 47 of file elf.h.

Referenced by elf_check().

4.11.1.24 #define EI_NIDENT 16

size of e_ident[]

Definition at line 62 of file elf.h.

4.11.1.25 #define EI_PAD 7

Start of padding bytes.

Definition at line 59 of file elf.h.

4.11.1.26 #define EI_VERSION 6

File version.

Definition at line 56 of file elf.h.

Referenced by elf_check().

4.11.1.27 #define ELF32_ST_BIND(*i*) ((i) >> 4)

Definition at line 233 of file elf.h.

4.11.1.28 #define ELF32_ST_TYPE(*i*) ((i) & 0xf)

Definition at line 235 of file elf.h.

Referenced by elf_lookup_symbol().

4.11.1.29 #define ELF_MAGIC0 0x7f

File identification - byte 0 (0x7f)

Definition at line 66 of file elf.h.

Referenced by elf_check().

4.11.1.30 #define ELF_MAGIC1 'E'

File identification - byte 1 ('E')

Definition at line 69 of file elf.h.

Referenced by elf_check().

4.11.1.31 #define ELF_MAGIC2 'L'

File identification - byte 2 ('L')

Definition at line 72 of file elf.h.

Referenced by elf_check().

4.11.1.32 #define ELF_MAGIC3 'F'

File identification - byte 3 ('F')

Definition at line 75 of file elf.h.

Referenced by elf_check().

4.11.1.33 #define ELFCLASS32 1

32-bit objects

Definition at line 132 of file elf.h.

Referenced by elf_check().

4.11.1.34 #define ELFCLASS64 2

64-bit objects

Definition at line 135 of file elf.h.

4.11.1.35 #define ELFCLASSNONE 0

Invalid class.

Definition at line 129 of file elf.h.

4.11.1.36 #define ELFDATA2LSB 1

Little-endian.

Definition at line 142 of file elf.h.

Referenced by elf_check().

4.11.1.37 #define ELFDATA2MSB 2

Big-endian.

Definition at line 145 of file elf.h.

4.11.1.38 #define ELFDATANONE 0

Invalid data encoding.

Definition at line 139 of file elf.h.

4.11.1.39 #define EM_386 3

Intel 80386.

Definition at line 85 of file elf.h.

Referenced by elf_check().

4.11.1.40 #define EM_AARCH64 183

64-bit AARCH64 ARM

Definition at line 106 of file elf.h.

4.11.1.41 #define EM_ALTERA_NIOS2 113

Altera Nios 2 32-bit soft processor.

Definition at line 103 of file elf.h.

4.11.1.42 #define EM_ARM 40

32-bit ARM

Definition at line 94 of file elf.h.

4.11.1.43 #define EM_MICROBLAZE 189

Xilinx MicroBlaze 32-bit soft processor.

Definition at line 109 of file elf.h.

4.11.1.44 #define EM_MIPS 8

MIPS RS3000.

Definition at line 88 of file elf.h.

4.11.1.45 #define EM_NONE 0

No machine.

Definition at line 79 of file elf.h.

4.11.1.46 #define EM_OPENRISC 92

OpenRISC 32-bit embedded processor.

Definition at line 100 of file elf.h.

4.11.1.47 #define EM_SPARC 2

SPARC.

Definition at line 82 of file elf.h.

4.11.1.48 #define EM_SPARC32PLUS 18

Enhanced instruction set SPARC.

Definition at line 91 of file elf.h.

4.11.1.49 #define EM_X86_64 62

AMD64/X86-64.

Definition at line 97 of file elf.h.

4.11.1.50 #define ET_CORE 4

Core file.

Definition at line 125 of file elf.h.

4.11.1.51 #define ET_DYN 3

Shared object file.

Definition at line 122 of file elf.h.

4.11.1.52 #define ET_EXEC 2

Executable file.

Definition at line 119 of file elf.h.

Referenced by elf_check().

4.11.1.53 #define ET_NONE 0

No file type.

Definition at line 113 of file elf.h.

4.11.1.54 #define ET_REL 1

Relocatable file.

Definition at line 116 of file elf.h.

4.11.1.55 #define PF_R (1 << 2)

Definition at line 242 of file elf.h.

4.11.1.56 #define PF_W (1 << 1)

Definition at line 244 of file elf.h.

Referenced by elf_load().

4.11.1.57 #define PF_X (1 << 0)

Definition at line 246 of file elf.h.

4.11.1.58 #define PT_DYNAMIC 2

Dynamic linking information.

Definition at line 155 of file elf.h.

4.11.1.59 #define PT_INTERP 3

Path to program interpreter.

Definition at line 158 of file elf.h.

4.11.1.60 #define PT_LOAD 1

Loadable segment.

Definition at line 152 of file elf.h.

Referenced by elf_load().

4.11.1.61 #define PT_NOTE 4

Location and size of notes.

Definition at line 161 of file elf.h.

4.11.1.62 #define PT_NULL 0

Unused entry.

Definition at line 149 of file elf.h.

4.11.1.63 #define PT_PHDR 6

Program header table.

Definition at line 167 of file elf.h.

4.11.1.64 #define PT_SHLIB 5

Unspecified semantics.

Definition at line 164 of file elf.h.

4.11.1.65 #define SHT_DYNAMIC 6

Information for dynamic linking.

Definition at line 189 of file elf.h.

4.11.1.66 #define SHT_DYNSYM 11

Dynamic symbols table.

Definition at line 204 of file elf.h.

4.11.1.67 #define SHT_HASH 5

Symbol hash table.

Definition at line 186 of file elf.h.

4.11.1.68 #define SHT_NOBITS 8

Section without data (.bss)

Definition at line 195 of file elf.h.

4.11.1.69 #define SHT_NOTE 7

Notes section.

Definition at line 192 of file elf.h.

4.11.1.70 #define SHT_NULL 0

Inactive section.

Definition at line 171 of file elf.h.

4.11.1.71 #define SHT_PROGBITS 1

Program data.

Definition at line 174 of file elf.h.

4.11.1.72 #define SHT_REL 9

Relocations without addends.

Definition at line 198 of file elf.h.

4.11.1.73 #define SHT_RELA 4

Relocations with addends.

Definition at line 183 of file elf.h.

4.11.1.74 #define SHT_SHLIB 10

Reserved, unspecified semantic, not ABI compliant.

Definition at line 201 of file elf.h.

4.11.1.75 #define SHT_STRTAB 3

String table.

Definition at line 180 of file elf.h.

4.11.1.76 #define SHT_SYMTAB 2

Symbol table.

Definition at line 177 of file elf.h.

Referenced by elf_lookup_symbol().

4.11.1.77 #define STB_GLOBAL 1

Global binding.

Definition at line 211 of file elf.h.

4.11.1.78 #define STB_LOCAL 0

Local binding.

Definition at line 208 of file elf.h.

4.11.1.79 #define STB_WEAK 2

Weak binding.

Definition at line 214 of file elf.h.

4.11.1.80 #define STN_UNDEF 0

Undefined symbol index.

Definition at line 239 of file elf.h.

4.11.1.81 #define STT_FILE 4

Source file.

Definition at line 230 of file elf.h.

4.11.1.82 #define STT_FUNCTION 2

Function or other executable code.

Definition at line 224 of file elf.h.

Referenced by dump_call_stack().

4.11.1.83 #define STT_NOTYPE 0

Unspecified type.

Definition at line 218 of file elf.h.

4.11.1.84 #define STT_OBJECT 1

Data object.

Definition at line 221 of file elf.h.

4.11.1.85 #define STT_SECTION 3

Section symbol.

Definition at line 227 of file elf.h.

4.11.2 Typedef Documentation

4.11.2.1 typedef Elf32_auxv_t auxv_t

Definition at line 315 of file elf.h.

4.11.2.2 typedef uint32_t Elf32_Addr

Definition at line 248 of file elf.h.

4.11.2.3 typedef uint16_t Elf32_Half

Definition at line 250 of file elf.h.

4.11.2.4 typedef uint32_t Elf32_Off

Definition at line 252 of file elf.h.

4.11.2.5 typedef int32_t Elf32_Sword

Definition at line 254 of file elf.h.

4.11.2.6 typedef uint32_t Elf32_Word

Definition at line 256 of file elf.h.

4.11.3 Function Documentation

4.11.3.1 void elf_check (Elf32_Ehdr ∗ *elf*)

Definition at line 43 of file elf.c.

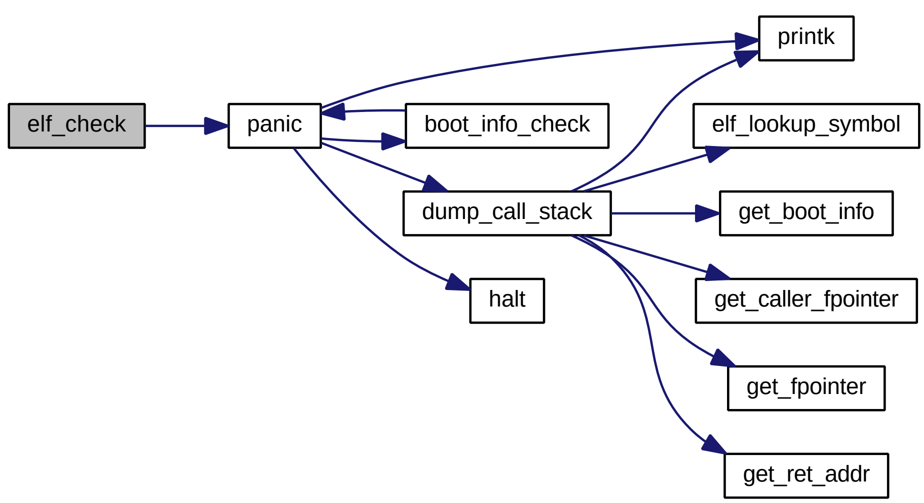
References Elf32_Ehdr::e_entry, Elf32_Ehdr::e_flags, Elf32_Ehdr::e_ident, Elf32_Ehdr::e_machine, Elf32_Ehdr::e_-phentsize, Elf32_Ehdr::e_phnum, Elf32_Ehdr::e_phoff, Elf32_Ehdr::e_type, Elf32_Ehdr::e_version, EI_CLASS, EI_D-ATA, EI_MAG0, EI_MAG1, EI_MAG2, EI_MAG3, EI_VERSION, ELF_MAGIC0, ELF_MAGIC1, ELF_MAGIC2, ELF_M-AGIC3, ELFCLASS32, ELFDATA2LSB, EM_386, ET_EXEC, and panic().

Referenced by elf_load().

```
43                                  {
44     /* check: valid ELF binary magic number */
45     if(     elf->e_ident[EI_MAG0] != ELF_MAGIC0 ||
46             elf->e_ident[EI_MAG1] != ELF_MAGIC1 ||
47             elf->e_ident[EI_MAG2] != ELF_MAGIC2 ||
48             elf->e_ident[EI_MAG3] != ELF_MAGIC3 ) {
49         panic("Not an ELF binary");
50     }
51 
52     /* check: 32-bit objects */
53     if(elf->e_ident[EI_CLASS] != ELFCLASS32) {
54         panic("Bad file class");
55     }
56 
57     /* check: endianess */
58     if(elf->e_ident[EI_DATA] != ELFDATA2LSB) {
```

```
        panic("Bad endianess");
    }

    /* check: version */
    if(elf->e_version != 1 || elf->e_ident[EI_VERSION] != 1) {
        panic("Not ELF version 1");
    }

    /* check: machine */
    if(elf->e_machine != EM_386) {
        panic("This process manager binary does not target the x86 architecture");
    }

    /* check: the 32-bit Intel architecture defines no flags */
    if(elf->e_flags != 0) {
        panic("Invalid flags specified");
    }

    /* check: file type is executable */
    if(elf->e_type != ET_EXEC) {
        panic("process manager binary is not an an executable");
    }

    /* check: must have a program header */
    if(elf->e_phoff == 0 || elf->e_phnum == 0) {
        panic("No program headers");
    }

    /* check: must have an entry point */
    if(elf->e_entry == 0) {
        panic("No entry point for process manager");
    }

    /* check: program header entry size */
    if(elf->e_phentsize != sizeof(Elf32_Phdr)) {
        panic("Unsupported program header size");
    }
}
```

Here is the call graph for this function:



4.11.3.2 void elf_load (**elf_info_t** ∗ *info*, **Elf32_Ehdr** ∗ *elf*, **addr_space_t** ∗ *addr_space*, **boot_alloc_t** ∗ *boot_alloc*)

TODO: add exec flag once PAE is enabled

TODO add exec flag once PAE is enabled TODO lookup actual address of page frame

Definition at line 98 of file elf.c.

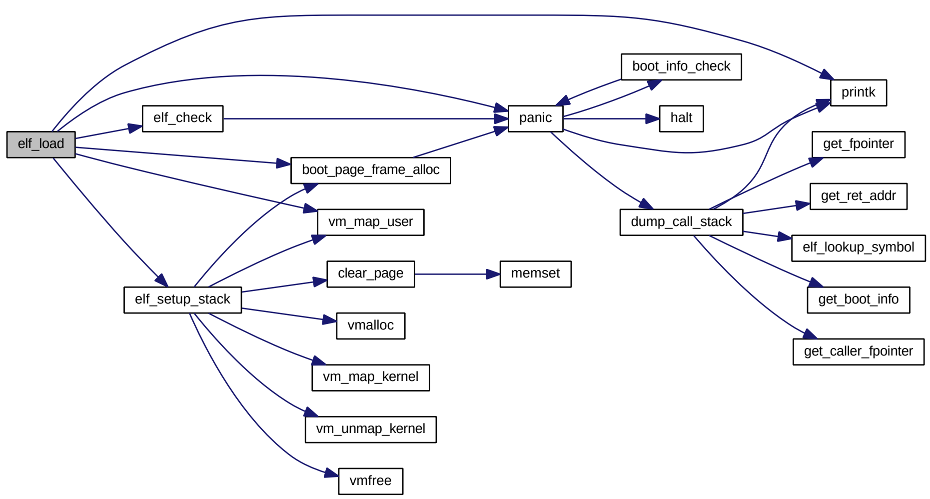
References elf_info_t::addr_space, ALIGN_END_PTR, ALIGN_START_PTR, elf_info_t::at_phdr, elf_info_t::at_phent, elf_info_t::at_phnum, boot_page_frame_alloc(), Elf32_Ehdr::e_entry, Elf32_Ehdr::e_phentsize, Elf32_Ehdr::e_phnum, Elf32_Ehdr::e_phoff, EARLY_PTR_TO_PHYS_ADDR, elf_check(), elf_setup_stack(), elf_info_t::entry, Elf32_Phdr::p_filesz, Elf32_Phdr::p_memsz, Elf32_Phdr::p_vaddr, PAGE_SIZE, panic(), PF_W, printk(), PT_LOAD, VM_FLAG_READ_ONLY, VM_FLAG_READ_WRITE, and vm_map_user().

Referenced by kmain().

```
                                     {

    unsigned int idx;

    /* check that ELF binary is valid */
    elf_check(elf);

    /* get the program header table */
    Elf32_Phdr * phdr   = (Elf32_Phdr *)((char *)elf + elf->e_phoff);

    info->at_phdr       = (addr_t)phdr;
    info->at_phnum      = elf->e_phnum;
    info->at_phent      = elf->e_phentsize;
    info->addr_space    = addr_space;
    info->entry         = (addr_t)elf->e_entry;

    for(idx = 0; idx < elf->e_phnum; ++idx) {
        if(phdr[idx].p_type != PT_LOAD) {
            continue;
        }

        /* check that the segment is not in the region reserved for kernel use */
        if(! user_buffer_check((void *)phdr[idx].p_vaddr, phdr[idx].p_memsz)) {
            panic("process manager memory layout -- address of segment too low");
        }

        /* set start and end addresses for mapping and copying */
        char *file_ptr  = (char *)elf + phdr[idx].p_offset;
        char *vptr      = (char *)phdr[idx].p_vaddr;
        char *vend      = vptr  + phdr[idx].p_memsz;  /* limit for padding */
        char *vfend     = vptr  + phdr[idx].p_filesz; /* limit for copy */

        /* align on page boundaries, be inclusive,
           note that vfend is not aligned        */
        file_ptr        = ALIGN_START_PTR(file_ptr, PAGE_SIZE);
        vptr            = ALIGN_START_PTR(vptr,     PAGE_SIZE);
        vend            = ALIGN_END_PTR(vend,      PAGE_SIZE);

        /* copy if we have to */
        if( (phdr[idx].p_flags & PF_W) || (phdr[idx].p_filesz != phdr[idx].p_memsz) ) {
            while(vptr < vend) {
                unsigned long    flags;
                char            *stop;

                /* start of this page and next page */
                char *vnext     = vptr + PAGE_SIZE;

                /* set flags */
                if(phdr[idx].p_flags & PF_W) {
                    flags = VM_FLAG_READ_WRITE;
                }
                else  {
                    flags = VM_FLAG_READ_ONLY;
                }

                /* allocate and map the new page */
                kern_paddr_t page = boot_page_frame_alloc(boot_alloc);
                vm_map_user(addr_space, (addr_t)vptr, page, flags);

                /* copy */
                if(vnext > vfend) {
                    stop = vfend;
                }
                else {
                    stop = vnext;
                }

                while(vptr < stop) {
                    *(vptr++) = *(file_ptr++);
                }
```

```
                /* pad */
                while(vptr < vnext) {
                    *(vptr++) = 0;
                }
            }
        }
        else {
            while(vptr < vend) {
                /* perform mapping */
                vm_map_user(addr_space, (addr_t)vptr, EARLY_PTR_TO_PHYS_ADDR(file_ptr), VM_FLAG_READ_ONLY);

                vptr     += PAGE_SIZE;
                file_ptr += PAGE_SIZE;
            }
        }
    }

    elf_setup_stack(info, boot_alloc);

    printk("ELF binary loaded.\n");
}
```

Here is the call graph for this function:



4.11.3.3 int elf_lookup_symbol (const Elf32_Ehdr * *elf_header*, Elf32_Addr *addr*, int *type*, elf_symbol_t * *result*)

Definition at line 272 of file elf.c.

References elf_symbol_t::addr, Elf32_Ehdr::e_shnum, ELF32_ST_TYPE, elf_symbol_t::name, NULL, Elf32_Shdr::sh_entsize, Elf32_Shdr::sh_link, Elf32_Shdr::sh_offset, Elf32_Shdr::sh_size, Elf32_Shdr::sh_type, SHT_SYMTAB, Elf32_Sym::st_info, Elf32_Sym::st_name, Elf32_Sym::st_size, and Elf32_Sym::st_value.

Referenced by dump_call_stack().

```
                                     {

    int     idx;
    size_t  symbol_entry_size;
    size_t  symbol_table_size;

    const char *elf_file         = elf_file_bytes(elf_header);
    const char *symbols_table    = NULL;
    const char *string_table     = NULL;

    for(idx = 0; idx < elf_header->e_shnum; ++idx) {
```

```
        const Elf32_Shdr *section_header = elf_get_section_header(elf_header, idx);

        if(section_header->sh_type == SHT_SYMTAB) {
            symbols_table       = &elf_file[section_header->sh_offset];
            symbol_entry_size   = section_header->sh_entsize;
            symbol_table_size   = section_header->sh_size;

            const Elf32_Shdr *string_section_header = elf_get_section_header(
                    elf_header,
                    section_header->sh_link);

            string_table = &elf_file[string_section_header->sh_offset];

            break;
        }
    }

    if(symbols_table == NULL) {
        /* no symbol table */
        return -1;
    }

    const char *symbol = symbols_table;

    while(symbol < symbols_table + symbol_table_size) {
        const Elf32_Sym *symbol_header = (const Elf32_Sym *)symbol;

        if(ELF32_ST_TYPE(symbol_header->st_info) == type) {
            Elf32_Addr lookup_addr  = (Elf32_Addr)addr;
            Elf32_Addr start        = symbol_header->st_value;
            Elf32_Addr end          = start + symbol_header->st_size;

            if(lookup_addr >= start && lookup_addr < end) {
                result->addr = symbol_header->st_value;
                result->name = &string_table[symbol_header->st_name];

                return 0;
            }
        }

        symbol += symbol_entry_size;
    }

    /* not found */
    return -1;
}
```

4.11.3.4 void elf_setup_stack (elf_info_t ∗ *info,* boot_alloc_t ∗ *boot_alloc*)

TODO: check for overlap of stack with loaded segments

Definition at line 198 of file elf.c.

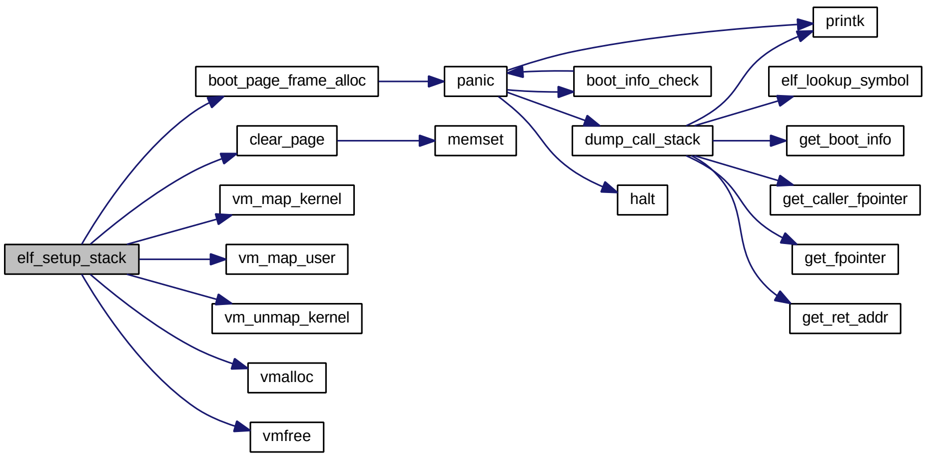
References Elf32_auxv_t::a_type, Elf32_auxv_t::a_un, Elf32_auxv_t::a_val, elf_info_t::addr_space, AT_ENTRY, AT_-NULL, AT_PAGESZ, elf_info_t::at_phdr, AT_PHDR, elf_info_t::at_phent, AT_PHENT, elf_info_t::at_phnum, AT_PHN-UM, AT_STACKBASE, boot_page_frame_alloc(), clear_page(), elf_info_t::entry, PAGE_SIZE, elf_info_t::stack_addr, STACK_BASE, STACK_START, VM_FLAG_READ_WRITE, vm_map_kernel(), vm_map_user(), vm_unmap_kernel(), vmalloc(), and vmfree().

Referenced by elf_load().

```
198                                                                  {
199     kern_paddr_t page;
200     addr_t vpage;
201 
204     /* initial stack allocation */
205     for(vpage = (addr_t)STACK_START; vpage < (addr_t)STACK_BASE; vpage +=
      PAGE_SIZE) {
206         page  = boot_page_frame_alloc(boot_alloc);
207         vm_map_user(info->addr_space, vpage, page, VM_FLAG_READ_WRITE);
208 
209         /* This newly allocated page may have data left from a previous boot which
210          * may contain sensitive information. Let's clear it. */
```

```
        clear_page(vpage);
    }

    /* At this point, page has the address of the stack's top-most page frame,
     * which is the one in which we are about to copy the auxiliary vectors. Map
     * it temporarily in this address space so we can write to it. */
    addr_t top_page = vmalloc();
    vm_map_kernel(top_page, page, VM_FLAG_READ_WRITE);

    /* start at the top */
    uint32_t *sp = (uint32_t *)(top_page + PAGE_SIZE);

    /* Program name string: "proc", null-terminated */
    *(--sp) = 0;
    *(--sp) = 0x636f7270;

    char *argv0 = (char *)STACK_BASE - 2 * sizeof(uint32_t);

    /* auxiliary vectors */
    Elf32_auxv_t *auxvp = (Elf32_auxv_t *)sp - 7;

    auxvp[0].a_type     = AT_PHDR;
    auxvp[0].a_un.a_val = (int32_t)info->at_phdr;

    auxvp[1].a_type     = AT_PHENT;
    auxvp[1].a_un.a_val = (int32_t)info->at_phent;

    auxvp[2].a_type     = AT_PHNUM;
    auxvp[2].a_un.a_val = (int32_t)info->at_phnum;

    auxvp[3].a_type     = AT_PAGESZ;
    auxvp[3].a_un.a_val = PAGE_SIZE;

    auxvp[4].a_type     = AT_ENTRY;
    auxvp[4].a_un.a_val = (int32_t)info->entry;

    auxvp[5].a_type     = AT_STACKBASE;
    auxvp[5].a_un.a_val = STACK_BASE;

    auxvp[6].a_type     = AT_NULL;
    auxvp[6].a_un.a_val = 0;

    sp = (uint32_t *)auxvp;

    /* empty environment variables */
    *(--sp) = 0;

    /* argv with only program name */
    *(--sp) = 0;
    *(--sp) = (uint32_t)argv0;

    /* argc */
    *(--sp) = 1;

    info->stack_addr = (addr_t)STACK_BASE - PAGE_SIZE + ((addr_t)sp - top_page);

    /* unmap and free temporary page */
    vm_unmap_kernel(top_page);
    vmfree(top_page);
}
```

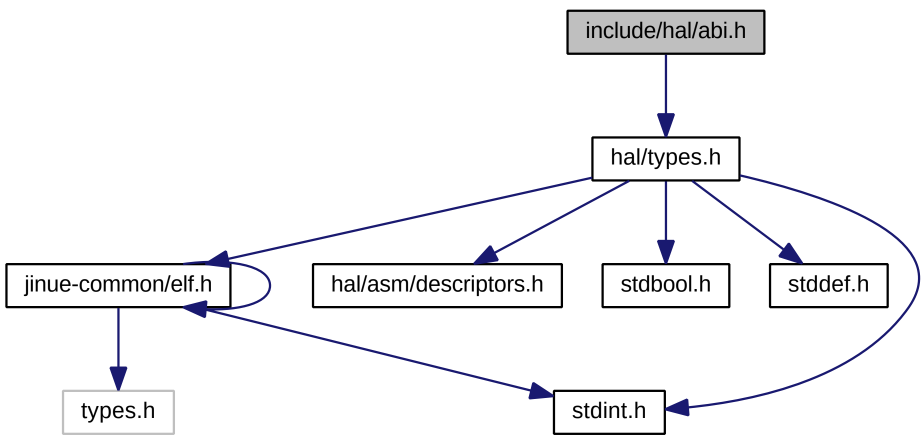
Here is the call graph for this function:



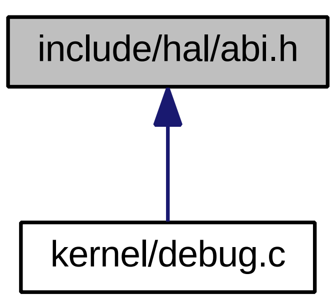
4.12 include/hal/abi.h File Reference

`#include <hal/types.h>`

Include dependency graph for abi.h:



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



Functions

- **addr_t get_fpointer** (void)
- **addr_t get_caller_fpointer** (**addr_t** fptr)
- **addr_t get_ret_addr** (**addr_t** fptr)
- **addr_t get_program_counter** (void)

4.12.1 Function Documentation

4.12.1.1 addr_t get_caller_fpointer (addr_t *fptr*)

Referenced by dump_call_stack().

4.12.1.2 addr_t get_fpointer (void)

Referenced by dump_call_stack().

4.12.1.3 addr_t get_program_counter (void)

4.12.1.4 addr_t get_ret_addr (addr_t *fptr*)

Referenced by dump_call_stack().

4.13 include/hal/asm/descriptors.h File Reference

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

Macros

- #define **SEG_SELECTOR**(index, rpl) (((index) << 3) | ((rpl) & 0x3))
- #define **RPL_KERNEL** 0
- #define **RPL_USER** 3
- #define **GDT_NULL** 0

  GDT entry for the null descriptor.
- #define **GDT_KERNEL_CODE** 1

  GDT entry for kernel code segment.
- #define **GDT_KERNEL_DATA** 2

  GDT entry for kernel data segment.
- #define **GDT_USER_CODE** 3

  GDT entry for user code segment.
- #define **GDT_USER_DATA** 4

  GDT entry for user data segment.
- #define **GDT_TSS** 5

GDT entry for task-state segment (TSS)

- #define **GDT_PER_CPU_DATA** 6

  GDT entry for per-cpu data (includes the TSS)
- #define **GDT_USER_TLS_DATA** 7

  GDT entry for thread-local storage.
- #define **GDT_LENGTH** 8

  number of descriptors in GDT
- #define **SEG_FLAGS_OFFSET** 40

  offset of descriptor type in descriptor
- #define **TSS_LIMIT** 104

  size of the task-state segment (TSS)
- #define **SEG_FLAG_PRESENT** (1<<7)

  segment is present
- #define **SEG_FLAG_SYSTEM** 0

  system segment (i.e.
- #define **SEG_FLAG_NOSYSTEM** (1<<4)

  code/data/stack segment
- #define **SEG_FLAG_32BIT** (1<<14)

  32-bit segment
- #define **SEG_FLAG_16BIT** 0

  16-bit segment
- #define **SEG_FLAG_32BIT_GATE** (1<<3)

  32-bit gate
- #define **SEG_FLAG_16BIT_GATE** 0

  16-bit gate
- #define **SEG_FLAG_BUSY** (1<<1)

  task is busy (for TSS descriptor)
- #define **SEG_FLAG_IN_PAGES** (1<<15)

  limit has page granularity
- #define **SEG_FLAG_IN_BYTES** 0

  limit has byte granularity
- #define **SEG_FLAG_KERNEL** 0

  kernel/supervisor segment (privilege level 0)
- #define **SEG_FLAG_USER** (3<<5)

  user segment (privilege level 3)
- #define **SEG_FLAG_NORMAL** (**SEG_FLAG_32BIT** | **SEG_FLAG_IN_PAGES** | **SEG_FLAG_NOSYSTEM** | **SEG_FLAG_PRESENT**)

  commonly used segment flags
- #define **SEG_FLAG_NORMAL_GATE** (**SEG_FLAG_32BIT_GATE** | **SEG_FLAG_SYSTEM** | **SEG_FLAG_PRESENT**)

  commonly used gate flags
- #define **SEG_FLAG_TSS** (**SEG_FLAG_IN_BYTES** | **SEG_FLAG_SYSTEM** | **SEG_FLAG_PRESENT**)

  commonly used flags for task-state segment
- #define **SEG_TYPE_READ_ONLY** 0

  read-only data segment
- #define **SEG_TYPE_DATA** 2

  read/write data segment

- #define **SEG_TYPE_TASK_GATE** 5

  task gate
- #define **SEG_TYPE_INTERRUPT_GATE** 6

  interrupt gate
- #define **SEG_TYPE_TRAP_GATE** 7

  trap gate
- #define **SEG_TYPE_TSS** 9

  task-state segment (TSS)
- #define **SEG_TYPE_CODE** 10

  code segment
- #define **SEG_TYPE_CALL_GATE** 12

  call gate

4.13.1 Macro Definition Documentation

4.13.1.1 #define GDT_KERNEL_CODE 1

GDT entry for kernel code segment.

Definition at line 46 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.2 #define GDT_KERNEL_DATA 2

GDT entry for kernel data segment.

Definition at line 49 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.3 #define GDT_LENGTH 8

number of descriptors in GDT

Definition at line 67 of file descriptors.h.

4.13.1.4 #define GDT_NULL 0

GDT entry for the null descriptor.

Definition at line 43 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.5 #define GDT_PER_CPU_DATA 6

GDT entry for per-cpu data (includes the TSS)

Definition at line 61 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.6 #define GDT_TSS 5

GDT entry for task-state segment (TSS)

Definition at line 58 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.7 #define GDT_USER_CODE 3

GDT entry for user code segment.

Definition at line 52 of file descriptors.h.

Referenced by cpu_init_data(), and thread_page_init().

4.13.1.8 #define GDT_USER_DATA 4

GDT entry for user data segment.

Definition at line 55 of file descriptors.h.

Referenced by cpu_init_data(), and thread_page_init().

4.13.1.9 #define GDT_USER_TLS_DATA 7

GDT entry for thread-local storage.

Definition at line 64 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.10 #define RPL_KERNEL 0

Definition at line 38 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.11 #define RPL_USER 3

Definition at line 40 of file descriptors.h.

Referenced by thread_page_init().

4.13.1.12 #define SEG_FLAG_16BIT 0

16-bit segment

Definition at line 88 of file descriptors.h.

4.13.1.13 #define SEG_FLAG_16BIT_GATE 0

16-bit gate

Definition at line 94 of file descriptors.h.

4.13.1.14 #define SEG_FLAG_32BIT (1<<14)

32-bit segment

Definition at line 85 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.15 #define SEG_FLAG_32BIT_GATE (1<<3)

32-bit gate

Definition at line 91 of file descriptors.h.

4.13.1.16 #define SEG_FLAG_BUSY (1<<1)

task is busy (for TSS descriptor)

Definition at line 97 of file descriptors.h.

4.13.1.17 #define SEG_FLAG_IN_BYTES 0

limit has byte granularity

Definition at line 103 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.18 #define SEG_FLAG_IN_PAGES (1<<15)

limit has page granularity

Definition at line 100 of file descriptors.h.

4.13.1.19 #define SEG_FLAG_KERNEL 0

kernel/supervisor segment (privilege level 0)

Definition at line 106 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.20 #define SEG_FLAG_NORMAL (SEG_FLAG_32BIT | SEG_FLAG_IN_PAGES | SEG_FLAG_NOSYSTEM | SEG_FLAG_PRESENT)

commonly used segment flags

Definition at line 112 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.21 #define SEG_FLAG_NORMAL_GATE (SEG_FLAG_32BIT_GATE | SEG_FLAG_SYSTEM | SEG_FLAG_PRESENT)

commonly used gate flags

Definition at line 116 of file descriptors.h.

4.13.1.22 #define SEG_FLAG_NOSYSTEM (1<<4)

code/data/stack segment

Definition at line 82 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.23 #define SEG_FLAG_PRESENT (1<<7)

segment is present

Definition at line 76 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.24 #define SEG_FLAG_SYSTEM 0

system segment (i.e.

call-gate, etc.)

Definition at line 79 of file descriptors.h.

4.13.1.25 #define SEG_FLAG_TSS (SEG_FLAG_IN_BYTES | SEG_FLAG_SYSTEM | SEG_FLAG_PRESENT)

commonly used flags for task-state segment

Definition at line 120 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.26 #define SEG_FLAG_USER (3<<5)

user segment (privilege level 3)

Definition at line 109 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.27 #define SEG_FLAGS_OFFSET 40

offset of descriptor type in descriptor

Definition at line 70 of file descriptors.h.

4.13.1.28 #define SEG_SELECTOR(*index*, *rpl*) (((index) << 3) | ((rpl) & 0x3))

Definition at line 35 of file descriptors.h.

Referenced by cpu_init_data(), and thread_page_init().

4.13.1.29 #define SEG_TYPE_CALL_GATE 12

call gate

Definition at line 146 of file descriptors.h.

4.13.1.30 #define SEG_TYPE_CODE 10

code segment

Definition at line 143 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.31 #define SEG_TYPE_DATA 2

read/write data segment

Definition at line 128 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.32 #define SEG_TYPE_INTERRUPT_GATE 6

interrupt gate

Definition at line 134 of file descriptors.h.

4.13.1.33 #define SEG_TYPE_READ_ONLY 0

read-only data segment

Definition at line 125 of file descriptors.h.

4.13.1.34 #define SEG_TYPE_TASK_GATE 5

task gate

Definition at line 131 of file descriptors.h.

4.13.1.35 #define SEG_TYPE_TRAP_GATE 7

trap gate

Definition at line 137 of file descriptors.h.

4.13.1.36 #define SEG_TYPE_TSS 9

task-state segment (TSS)

Definition at line 140 of file descriptors.h.

Referenced by cpu_init_data().

4.13.1.37 #define TSS_LIMIT 104

size of the task-state segment (TSS)

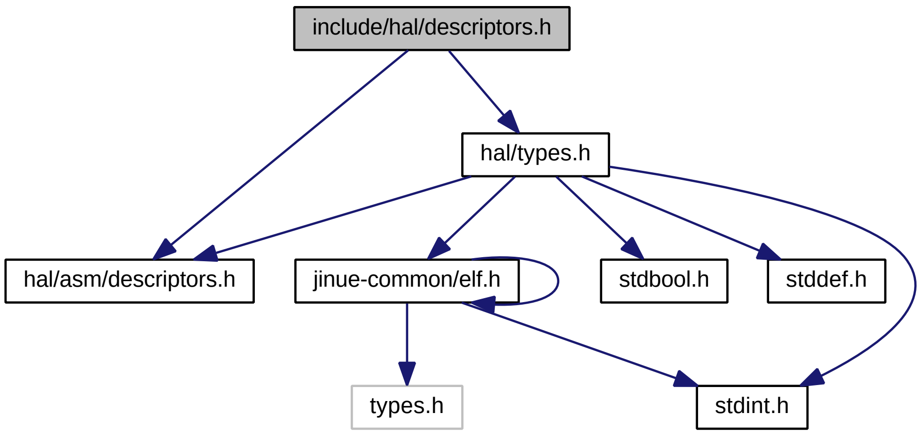
Definition at line 73 of file descriptors.h.

Referenced by cpu_init_data().

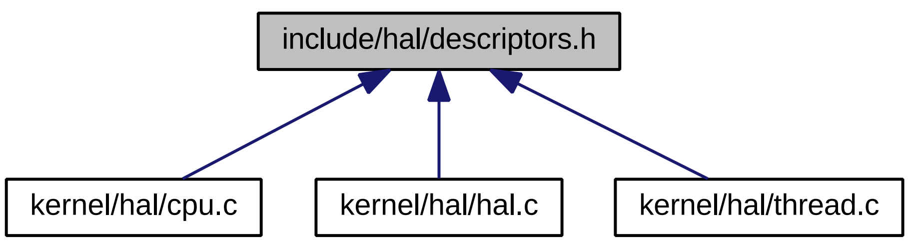
4.14 include/hal/descriptors.h File Reference

```
#include <hal/asm/descriptors.h>
#include <hal/types.h>
```

Include dependency graph for descriptors.h:



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



Macros

- #define **PACK_DESCRIPTOR**(val, mask, shamt1, shamt2) ((((**uint64_t**)(**uintptr_t**)(val) >> shamt1) & mask) << shamt2)
- #define **SEG_DESCRIPTOR**(base, limit, type)
- #define **GATE_DESCRIPTOR**(segment, offset, type, param_count)

4.14.1 Macro Definition Documentation

4.14.1.1 #define GATE_DESCRIPTOR(*segment*, *offset*, *type*, *param_count*)

Value:

```
(       PACK_DESCRIPTOR((type),        0xff,    0,  SEG_FLAGS_OFFSET) \
    | PACK_DESCRIPTOR((param_count), 0xf,     0,  32) \
    | PACK_DESCRIPTOR((segment),     0xffff,  0,  16) \
    | PACK_DESCRIPTOR((offset),      0xffff, 16,  48) \
    | PACK_DESCRIPTOR((offset),      0xffff,  0,   0) \
)
```

Definition at line 52 of file descriptors.h.

4.14.1.2 #define PACK_DESCRIPTOR(*val*, *mask*, *shamt1*, *shamt2*) ((((uint64_t)(uintptr_t)(val) >> shamt1) & mask) << shamt2)

Definition at line 40 of file descriptors.h.

4.14.1.3 #define SEG_DESCRIPTOR(*base*, *limit*, *type*)

Value:

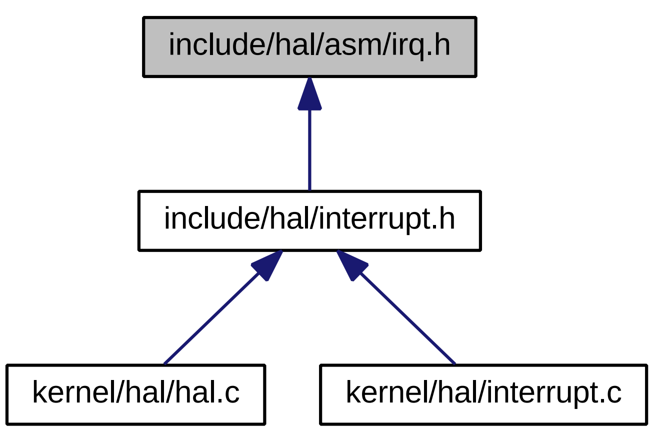
```
(       PACK_DESCRIPTOR((type),  0xf0ff,  0, SEG_FLAGS_OFFSET) \
    | PACK_DESCRIPTOR((base),  0xff,   24, 56) \
    | PACK_DESCRIPTOR((base),  0xff,   16, 32) \
    | PACK_DESCRIPTOR((base),  0xffff,  0, 16) \
    | PACK_DESCRIPTOR((limit), 0xf,    16, 48) \
    | PACK_DESCRIPTOR((limit), 0xffff,  0,  0) \
)
```

Definition at line 43 of file descriptors.h.

Referenced by cpu_init_data().

4.15 include/hal/asm/irq.h File Reference

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



Macros

- #define **IDT_VECTOR_COUNT** 256
- #define **IDT_LAST_EXCEPTION** 31
- #define **IDT_PIC8259_BASE** (**IDT_LAST_EXCEPTION** + 1)
- #define **EXCEPTION_DIV_ZERO** 0

  Divide Error.
- #define **EXCEPTION_NMI** 2

  NMI Interrupt.
- #define **EXCEPTION_BREAK** 3

  Breakpoint.
- #define **EXCEPTION_OVERFLOW** 4

  Overflow.
- #define **EXCEPTION_BOUND** 5

  BOUND Range Exceeded.
- #define **EXCEPTION_INVALID_OP** 6

  Invalid Opcode (Undefined Opcode)
- #define **EXCEPTION_NO_COPROC** 7

  Device Not Available (No Math Coprocessor)
- #define **EXCEPTION_DOUBLE_FAULT** 8

  Double Fault.
- #define **EXCEPTION_INVALID_TSS** 10

  Invalid TSS.
- #define **EXCEPTION_SEGMENT_NOT_PRESENT** 11

  Segment Not Present.
- #define **EXCEPTION_STACK_SEGMENT** 12

  Stack-Segment Fault.
- #define **EXCEPTION_GENERAL_PROTECTION** 13

  General Protection.
- #define **EXCEPTION_PAGE_FAULT** 14

  Page Fault.
- #define **EXCEPTION_MATH** 16

  x87 FPU Floating-Point Error (Math Fault)
- #define **EXCEPTION_ALIGNMENT** 17

  Alignment Check.
- #define **EXCEPTION_MACHINE_CHECK** 18

  Machine Check.
- #define **EXCEPTION_SIMD** 19

  SIMD Floating-Point Exception.
- #define **HAS_ERRCODE**(x) ((x) == **EXCEPTION_DOUBLE_FAULT** || (x) == **EXCEPTION_ALIGNMENT** || ((x) >= **EXCEPTION_INVALID_TSS** && (x) <= **EXCEPTION_PAGE_FAULT**))

4.15.1 Macro Definition Documentation

4.15.1.1 #define EXCEPTION_ALIGNMENT 17

Alignment Check.

Definition at line 84 of file irq.h.

4.15.1.2 #define EXCEPTION_BOUND 5

BOUND Range Exceeded.

Definition at line 54 of file irq.h.

4.15.1.3 #define EXCEPTION_BREAK 3

Breakpoint.

Definition at line 48 of file irq.h.

4.15.1.4 #define EXCEPTION_DIV_ZERO 0

Divide Error.

Definition at line 42 of file irq.h.

4.15.1.5 #define EXCEPTION_DOUBLE_FAULT 8

Double Fault.

Definition at line 63 of file irq.h.

4.15.1.6 #define EXCEPTION_GENERAL_PROTECTION 13

General Protection.

Definition at line 75 of file irq.h.

4.15.1.7 #define EXCEPTION_INVALID_OP 6

Invalid Opcode (Undefined Opcode)

Definition at line 57 of file irq.h.

4.15.1.8 #define EXCEPTION_INVALID_TSS 10

Invalid TSS.

Definition at line 66 of file irq.h.

4.15.1.9 #define EXCEPTION_MACHINE_CHECK 18

Machine Check.

Definition at line 87 of file irq.h.

4.15.1.10 #define EXCEPTION_MATH 16

x87 FPU Floating-Point Error (Math Fault)

Definition at line 81 of file irq.h.

4.15.1.11 #define EXCEPTION_NMI 2

NMI Interrupt.

Definition at line 45 of file irq.h.

4.15.1.12 #define EXCEPTION_NO_COPROC 7

Device Not Available (No Math Coprocessor)

Definition at line 60 of file irq.h.

4.15.1.13 #define EXCEPTION_OVERFLOW 4

Overflow.

Definition at line 51 of file irq.h.

4.15.1.14 #define EXCEPTION_PAGE_FAULT 14

Page Fault.

Definition at line 78 of file irq.h.

4.15.1.15 #define EXCEPTION_SEGMENT_NOT_PRESENT 11

Segment Not Present.

Definition at line 69 of file irq.h.

4.15.1.16 #define EXCEPTION_SIMD 19

SIMD Floating-Point Exception.

Definition at line 90 of file irq.h.

4.15.1.17 #define EXCEPTION_STACK_SEGMENT 12

Stack-Segment Fault.

Definition at line 72 of file irq.h.

4.15.1.18 #define HAS_ERRCODE(*x*) ((x) == EXCEPTION_DOUBLE_FAULT || (x) == EXCEPTION_ALIGNMENT || ((x) >= EXCEPTION_INVALID_TSS && (x) <= EXCEPTION_PAGE_FAULT))

Definition at line 92 of file irq.h.

4.15.1.19 #define IDT_LAST_EXCEPTION 31

Definition at line 37 of file irq.h.

Referenced by dispatch_interrupt().

4.15.1.20 #define IDT_PIC8259_BASE (IDT_LAST_EXCEPTION + 1)

Definition at line 39 of file irq.h.

Referenced by dispatch_interrupt(), and hal_init().

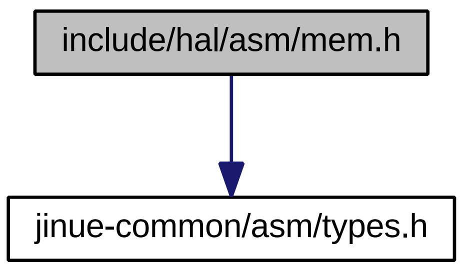
4.15.1.21 #define IDT_VECTOR_COUNT 256

Definition at line 35 of file irq.h.

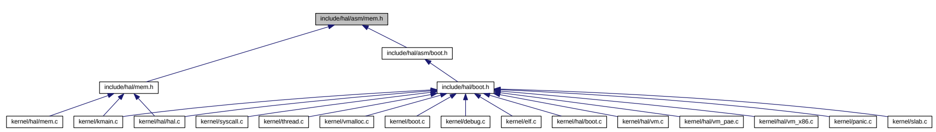
4.16 include/hal/asm/mem.h File Reference

`#include <jinue-common/asm/types.h>`

Include dependency graph for mem.h:



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



Macros

- #define **MEM_ZONE_DMA16_START** (1 ∗ **MB**)
- #define **MEM_ZONE_DMA16_END** (15 ∗ **MB**)
- #define **MEM_ZONE_MEM32_START** (16 ∗ **MB**)
- #define **MEM_ZONE_MEM32_END** 0xc0000000

4.16.1 Macro Definition Documentation

4.16.1.1 #define MEM_ZONE_DMA16_END (15 ∗ MB)

Definition at line 39 of file mem.h.

Referenced by mem_check_memory().

4.16.1.2 #define MEM_ZONE_DMA16_START (1 ∗ MB)

Definition at line 37 of file mem.h.

Referenced by mem_check_memory().

4.16.1.3 #define MEM_ZONE_MEM32_END 0xc0000000

Definition at line 43 of file mem.h.

Referenced by mem_check_memory().

4.16.1.4 #define MEM_ZONE_MEM32_START (16 ∗ MB)

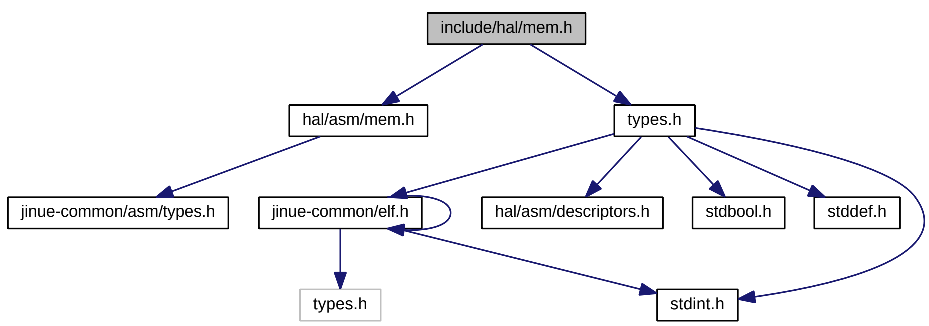
Definition at line 41 of file mem.h.

Referenced by mem_check_memory().

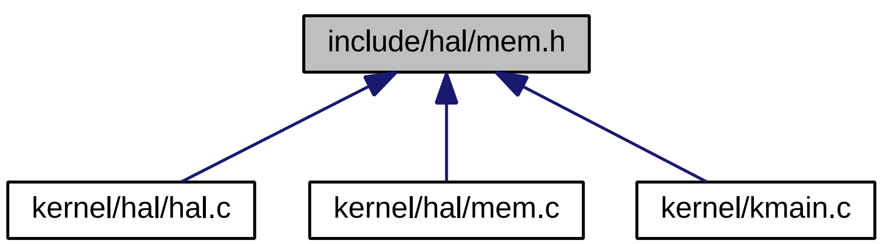
4.17 include/hal/mem.h File Reference

```
#include <hal/asm/mem.h>
#include <types.h>
```

Include dependency graph for mem.h:



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



Functions

- void **mem_check_memory** (**boot_alloc_t** ∗boot_alloc, const **boot_info_t** ∗**boot_info**)

4.17.1 Function Documentation

4.17.1.1 void mem_check_memory (**boot_alloc_t** ∗ *boot_alloc,* const **boot_info_t** ∗ *boot_info*)

ASSERTION: Any unsigned value less than MEM_ZONE_MEM32_END can be stored in 32 bits.

Definition at line 57 of file mem.c.

References e820_t::addr, assert, boot_info_t::boot_end, boot_info_t::e820_entries, boot_info_t::e820_map, E820_RAM, EARLY_VIRT_TO_PHYS, GB, KERNEL_EARLY_LIMIT, boot_alloc_t::kernel_paddr_limit, boot_alloc_t::kernel_paddr_top, boot_alloc_t::kernel_vm_limit, boot_alloc_t::kernel_vm_top, MEM_ZONE_DMA16_END, MEM_ZONE_DMA16_START, MEM_ZONE_MEM32_END, MEM_ZONE_MEM32_START, panic(), boot_info_t::ramdisk_size, boot_info_t::ramdisk_start, and e820_t::type.

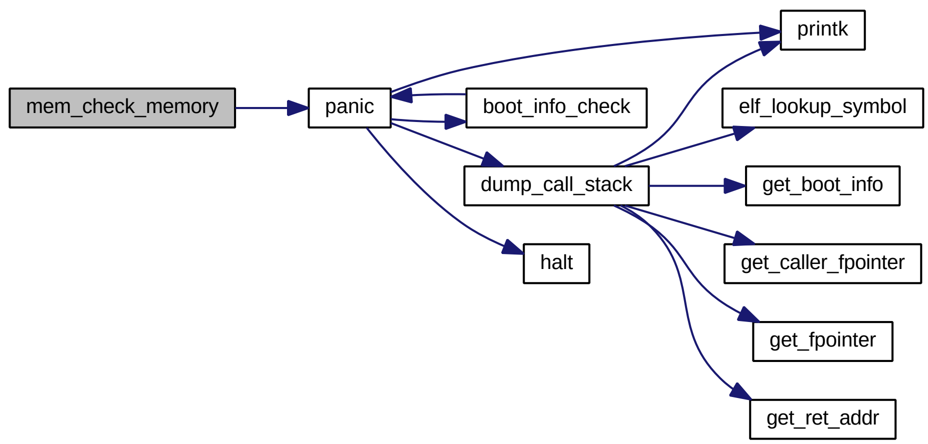
Referenced by kmain().

```
57                                                                                 {
58     int idx;
59
60 #if 0
61     if((uint64_t)boot_info->ramdisk_start + boot_info->ramdisk_size >
       MEM_ZONE_MEM32_END) {
62         panic("Initial RAM disk loaded too high in memory.");
63     }
64
65     uint32_t ramdisk_end = boot_info->ramdisk_start + boot_info->ramdisk_size;
66 #endif
67
68     /* ------------------------------------------------------------------------
69      * We consult the memory map provided by the BIOS to figure out how much
70      * memory is available in both zones usable by the kernel. We also want to
71      * make sure the initial RAM disk image is in available RAM.
72      *
73      * The first step in accomplishing this is to iterate over all entries that
74      * are reported as available RAM and confirm at least one of them covers
75      * each of both zones (at least the start) and the initial RAM disk.
76      * ---------------------------------------------------------------------- */
77     uint32_t zone_dma16_top = 0;
78     uint32_t zone_mem32_top = 0;
79 #if 0
80     bool ramdisk_ok         = false;
81 #endif
82
84     assert(MEM_ZONE_MEM32_END < (uint64_t)4 * GB);
85
86     for(idx = 0; idx < boot_info->e820_entries; ++idx) {
87         const e820_t *entry = &boot_info->e820_map[idx];
88
89         /* Consider only usable RAM entries. */
90         if(entry->type != E820_RAM) {
91             continue;
92         }
93
94         /* Ignore entries that start past MEM_ZONE_MEM32_END since the kernel
95          * cannot use them. Past this check, entry->addr is assumed to be
96          * representable in 32 bits.  */
97         if(entry->addr >= MEM_ZONE_MEM32_END) {
98             continue;
99         }
100
101         uint32_t entry_end = clip_e820_entry_end(entry);
102
103         /* If this entry covers the start the DMA16 zone, adjust the top pointer
104          * accordingly. Overlapping entries are resolved in favor of the largest
105          * entry. */
106         if(entry->addr <= MEM_ZONE_DMA16_START && entry_end >
       MEM_ZONE_DMA16_START) {
107             /* This condition covers the initial case where zone_dma16_top is zero. */
108             if(entry_end > zone_dma16_top) {
109                 zone_dma16_top = entry_end;
110             }
111         }
112
113         /* Do the same for the MEM32 zone. */
114         if(entry->addr <= MEM_ZONE_MEM32_START && entry_end >
       MEM_ZONE_MEM32_START) {
```

```
            if(entry_end > zone_mem32_top) {
                zone_mem32_top = entry_end;
            }
        }

        /* If this entry covers the initial RAM disk, this is good argument in
         * favor of it being in available RAM (one more check below). Unlike the
         * above, the entry must cover the initial RAM disk image completely,
         * not just the start. */
#if 0
        if(entry->addr <= boot_info->ramdisk_start && entry_end >= ramdisk_end) {
            ramdisk_ok = true;
        }
#endif
    }

    /* ----------------------------------------------------------------------
     * Next, iterate over non-available RAM entries of the map to ensure nothing
     * is relevant there.
     * ---------------------------------------------------------------------- */
    for(idx = 0; idx < boot_info->e820_entries; ++idx) {
        const e820_t *entry = &boot_info->e820_map[idx];

        /* Consider only non-usable RAM entries. */
        if(entry->type == E820_RAM) {
            continue;
        }

        if(entry->addr >= MEM_ZONE_MEM32_END) {
            continue;
        }

        uint32_t entry_end = clip_e820_entry_end(entry);

        if(ranges_overlap(MEM_ZONE_DMA16_START, MEM_ZONE_DMA16_END, entry->addr, entry_end)) {
            if(entry->addr > MEM_ZONE_DMA16_START) {
                if(entry->addr < zone_dma16_top) {
                    zone_dma16_top = entry->addr;
                }
            }
            else {
                /* This reserved entry covers the start of the zone. */
                zone_dma16_top = 0;
            }
        }

        if(ranges_overlap(MEM_ZONE_MEM32_START, MEM_ZONE_MEM32_END, entry->addr, entry_end)) {
            if(entry->addr > MEM_ZONE_MEM32_START) {
                if(entry->addr < zone_mem32_top) {
                    zone_mem32_top = entry->addr;
                }
            }
            else {
                zone_mem32_top = 0;
            }
        }

        /* Check for overlap with the initial RAM disk. */
#if 0
        if(ranges_overlap(boot_info->ramdisk_start, ramdisk_end, entry->addr, entry_end)) {
            ramdisk_ok = false;
        }
#endif
    }

    /* ----------------------------------------------------------------------
     * Now that we are done, let's look at the results.
     * ---------------------------------------------------------------------- */
#if 0
    if(! ramdisk_ok) {
        panic("Initial RAM disk was loaded in reserved memory.");
    }
#endif

    /* It is early during the boot process and the page table set up by the
     * setup code is still being used. This (single) page table maps the first
     * two megabytes of RAM linearly starting at KLIMIT in the virtual address
     * space. */
    boot_alloc->kernel_vm_top      = boot_info->boot_end;
    boot_alloc->kernel_vm_limit    = (addr_t)KERNEL_EARLY_LIMIT;
    boot_alloc->kernel_paddr_top   = EARLY_VIRT_TO_PHYS(boot_alloc->
```

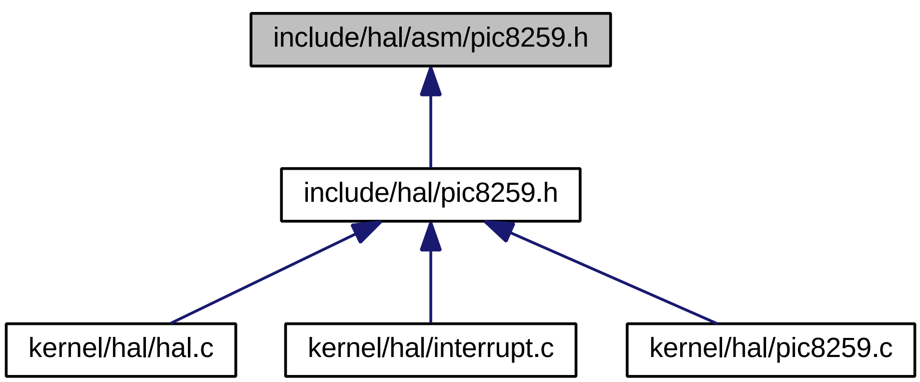
```
      kernel_vm_top);
    boot_alloc->kernel_paddr_limit = zone_dma16_top;

    if(boot_alloc->kernel_paddr_top > boot_alloc->kernel_paddr_limit) {
        panic("Kernel image was loaded in reserved memory.");
    }

    /* TODO Compute sequential allocation limit taking initrd into account */
    /* TODO Report zone limits */
}
```

Here is the call graph for this function:



4.18 include/hal/asm/pic8259.h File Reference

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



Macros

- #define **PIC8259_MASTER_BASE** 0x20

  Base I/O port for the master interrupt controller.
- #define **PIC8259_SLAVE_BASE** 0xa0

  Base I/O port for the slave interrupt controller.
- #define **PIC8259_ICW1_IC4** (1<<0)

ICW1 bit 0: ICW4 needed.

- #define **PIC8259_ICW1_SNGL** (1<<1)

  ICW1 bit 1: single (1) or cascade (0) mode.
- #define **PIC8259_ICW1_LTIM** (1<<3)

  ICW1 bit 3: level-triggered (1) or edge-triggered (0) interrupts.
- #define **PIC8259_ICW1_1** (1<<4)

  ICW1 bit 4: a control word with this bit set indicates this is ICW1.
- #define **PIC8259_ICW4_UPM** (1<<0)

  ICW4 bit 0: 8086/8088 mode (1) or MCS-80/85 mode (0)
- #define **PIC8259_ICW4_AEOI** (1<<1)

  ICW4 bit 1: Auto EOI.
- #define **PIC8259_EOI** 0x20

  OCW2: non-specific EOI command.
- #define **PIC8259_CASCADE_INPUT** 2

  Slave PIC is connected to input 2 of the master.
- #define **PIC8259_IRQ_COUNT** 16

  Number of IRQs handled by both cascaded PIC8259s together.

4.18.1 Macro Definition Documentation

4.18.1.1 #define PIC8259_CASCADE_INPUT 2

Slave PIC is connected to input 2 of the master.

Definition at line 63 of file pic8259.h.

Referenced by pic8259_init(), pic8259_mask_irq(), and pic8259_unmask_irq().

4.18.1.2 #define PIC8259_EOI 0x20

OCW2: non-specific EOI command.

Definition at line 60 of file pic8259.h.

Referenced by pic8259_eoi().

4.18.1.3 #define PIC8259_ICW1_1 (1<<4)

ICW1 bit 4: a control word with this bit set indicates this is ICW1.

Definition at line 51 of file pic8259.h.

Referenced by pic8259_init().

4.18.1.4 #define PIC8259_ICW1_IC4 (1<<0)

ICW1 bit 0: ICW4 needed.

Definition at line 42 of file pic8259.h.

Referenced by pic8259_init().

4.18.1.5 #define PIC8259_ICW1_LTIM (1<<3)

ICW1 bit 3: level-triggered (1) or edge-triggered (0) interrupts.

Definition at line 48 of file pic8259.h.

4.18.1.6 #define PIC8259_ICW1_SNGL (1<<1)

ICW1 bit 1: single (1) or cascade (0) mode.

Definition at line 45 of file pic8259.h.

4.18.1.7 #define PIC8259_ICW4_AEOI (1<<1)

ICW4 bit 1: Auto EOI.

Definition at line 57 of file pic8259.h.

4.18.1.8 #define PIC8259_ICW4_UPM (1<<0)

ICW4 bit 0: 8086/8088 mode (1) or MCS-80/85 mode (0)

Definition at line 54 of file pic8259.h.

Referenced by pic8259_init().

4.18.1.9 #define PIC8259_IRQ_COUNT 16

Number of IRQs handled by both cascaded PIC8259s together.

Definition at line 66 of file pic8259.h.

Referenced by dispatch_interrupt(), pic8259_eoi(), pic8259_mask_irq(), and pic8259_unmask_irq().

4.18.1.10 #define PIC8259_MASTER_BASE 0x20

Base I/O port for the master interrupt controller.

Definition at line 36 of file pic8259.h.

Referenced by pic8259_eoi(), pic8259_init(), pic8259_mask_irq(), and pic8259_unmask_irq().

4.18.1.11 #define PIC8259_SLAVE_BASE 0xa0

Base I/O port for the slave interrupt controller.

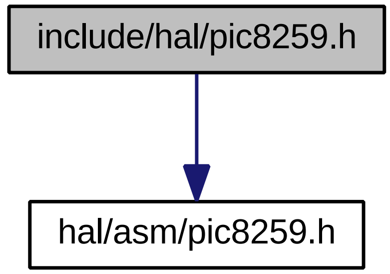
Definition at line 39 of file pic8259.h.

Referenced by pic8259_eoi(), pic8259_init(), pic8259_mask_irq(), and pic8259_unmask_irq().

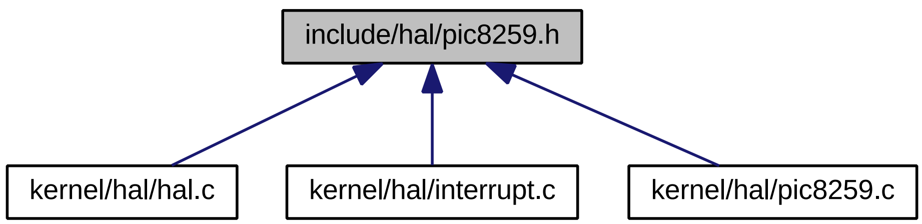
4.19 include/hal/pic8259.h File Reference

```
#include <hal/asm/pic8259.h>
```

Include dependency graph for pic8259.h:



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



Functions

- void **pic8259_init** (int intrvect_base)
- void **pic8259_mask_irq** (int irq)
- void **pic8259_unmask_irq** (int irq)
- void **pic8259_eoi** (int irq)

4.19.1 Function Documentation

4.19.1.1 void pic8259_eoi (int *irq*)

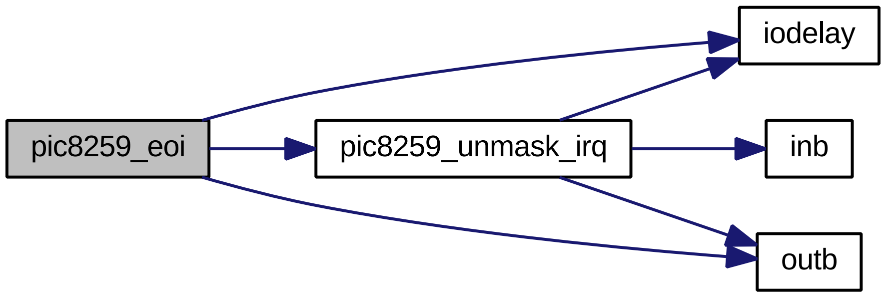
Definition at line 124 of file pic8259.c.

References iodelay(), outb(), PIC8259_EOI, PIC8259_IRQ_COUNT, PIC8259_MASTER_BASE, PIC8259_SLAVE_B-ASE, and pic8259_unmask_irq().

Referenced by dispatch_interrupt().

```
124                             {
125     if(irq < PIC8259_IRQ_COUNT) {
126         if(irq >= 8) {
127             outb(PIC8259_SLAVE_BASE+0, PIC8259_EOI);
128             iodelay();
129         }
130 
131         outb(PIC8259_MASTER_BASE+0, PIC8259_EOI);
132         iodelay();
133 
134         pic8259_unmask_irq(irq);
135     }
136 }
```

Here is the call graph for this function:



4.19.1.2 void pic8259_init (int *intrvect_base*)

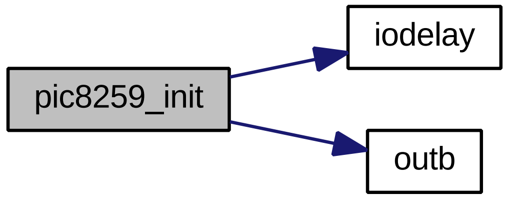
Definition at line 36 of file pic8259.c.

References iodelay(), outb(), PIC8259_CASCADE_INPUT, PIC8259_ICW1_1, PIC8259_ICW1_IC4, PIC8259_ICW4_-UPM, PIC8259_MASTER_BASE, and PIC8259_SLAVE_BASE.

Referenced by hal_init().

```
36                                       {
37      /* Issue ICW1 to start initialization sequence of both interrupt controllers.
38       * Specify there will be an ICW4 in the sequence. Specify the interrupts are
39       * edge-triggered and that the PICs are in a cascaded configuration by
40       * leaving the relevant flags cleared. */
41      outb(PIC8259_MASTER_BASE+0, PIC8259_ICW1_1 | PIC8259_ICW1_IC4);
42      iodelay();
43      outb(PIC8259_SLAVE_BASE+0, PIC8259_ICW1_1 | PIC8259_ICW1_IC4);
44      iodelay();
45
46      /* ICW2: base interrupt vector */
47      outb(PIC8259_MASTER_BASE+1, intrvect_base);
48      iodelay();
49      outb(PIC8259_SLAVE_BASE+1, intrvect_base + 8);
50      iodelay();
51
52      /* ICW3: master-slave connections */
53      outb(PIC8259_MASTER_BASE+1, (1<<PIC8259_CASCADE_INPUT));
54      iodelay();
55      outb(PIC8259_SLAVE_BASE+1, PIC8259_CASCADE_INPUT);
56      iodelay();
57
58      /* ICW4: Use 8088/8086 mode */
59      outb(PIC8259_MASTER_BASE+1, PIC8259_ICW4_UPM);
60      iodelay();
61      outb(PIC8259_SLAVE_BASE+1, PIC8259_ICW4_UPM);
62      iodelay();
63
64      /* Set interrupt mask: all masked */
65      outb(PIC8259_MASTER_BASE+1, 0xff & ~(1<<PIC8259_CASCADE_INPUT));
66      iodelay();
67      outb(PIC8259_SLAVE_BASE+1, 0xff);
68      iodelay();
69 }
```

Here is the call graph for this function:



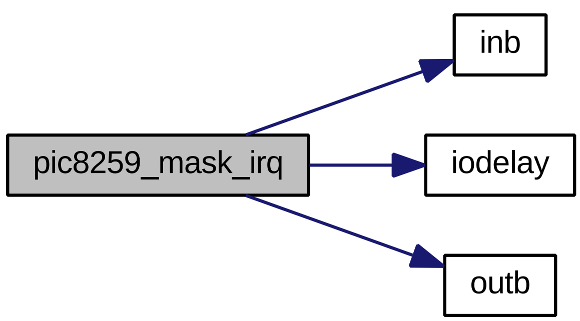
4.19.1.3 void pic8259_mask_irq (int *irq*)

Definition at line 71 of file pic8259.c.

References inb(), iodelay(), outb(), PIC8259_CASCADE_INPUT, PIC8259_IRQ_COUNT, PIC8259_MASTER_BASE, and PIC8259_SLAVE_BASE.

```
71                                {
72     if(irq < PIC8259_IRQ_COUNT) {
73         if(irq < 8 && irq != PIC8259_CASCADE_INPUT) {
74             int mask = inb(PIC8259_MASTER_BASE+1);
75             iodelay();
76 
77             mask |= (1<<irq);
78             outb(PIC8259_MASTER_BASE+1, mask);
79             iodelay();
80         }
81         else {
82             int mask = inb(PIC8259_SLAVE_BASE+1);
83             iodelay();
84 
85             mask |= (1<<(irq - 8));
86             outb(PIC8259_SLAVE_BASE+1, mask);
87             iodelay();
88         }
89     }
90 }
```

Here is the call graph for this function:



4.19.1.4 void pic8259_unmask_irq (int *irq*)

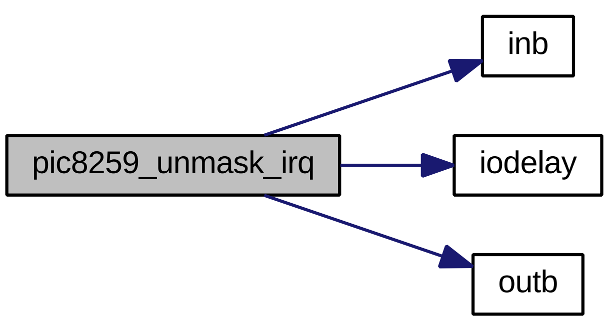
Definition at line 92 of file pic8259.c.

References inb(), iodelay(), outb(), PIC8259_CASCADE_INPUT, PIC8259_IRQ_COUNT, PIC8259_MASTER_BASE, and PIC8259_SLAVE_BASE.

Referenced by pic8259_eoi().

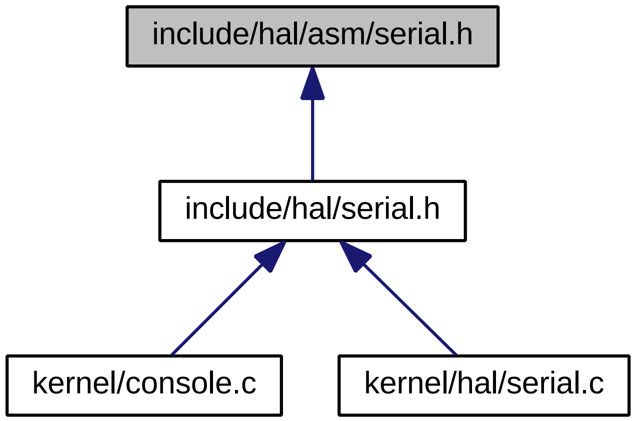
```
                                    {
    if(irq < PIC8259_IRQ_COUNT) {
        int master_irq;

        if(irq < 8) {
            master_irq = irq;
        }
        else {
            /* Unmask interrupt in slave PIC. */
            int mask = inb(PIC8259_SLAVE_BASE+1);
            iodelay();

            mask &= ~(1<<(irq - 8));
            outb(PIC8259_SLAVE_BASE+1, mask);
            iodelay();

            /* We will also want to unmask the cascaded interrupt line in the
             * master PIC. */
            master_irq = PIC8259_CASCADE_INPUT;
        }

        if(irq != PIC8259_CASCADE_INPUT) {
            int mask = inb(PIC8259_MASTER_BASE+1);
            iodelay();

            mask &= ~(1<<master_irq);
            outb(PIC8259_MASTER_BASE+1, mask);
            iodelay();
        }
    }
}
```

Here is the call graph for this function:



4.20 include/hal/asm/serial.h File Reference

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



Macros

- #define **SERIAL_COM1_IOPORT** 0x3f8
- #define **SERIAL_COM2_IOPORT** 0x2f8
- #define **SERIAL_COM3_IOPORT** 0x3e8
- #define **SERIAL_COM4_IOPORT** 0x2e8
- #define **SERIAL_REG_DATA_BUFFER** 0
- #define **SERIAL_REG_DIVISOR_LOW** 0
- #define **SERIAL_REG_INTR_ENABLE** 1
- #define **SERIAL_REG_DIVISOR_HIGH** 1
- #define **SERIAL_REG_INTR_ID** 2
- #define **SERIAL_REG_FIFO_CTRL** 2
- #define **SERIAL_REG_LINE_CTRL** 3
- #define **SERIAL_REG_MODEM_CTRL** 4
- #define **SERIAL_REG_LINE_STATUS** 5
- #define **SERIAL_REG_MODEM_STATUS** 6
- #define **SERIAL_REG_SCRATCH** 7

4.20.1 Macro Definition Documentation

4.20.1.1 #define SERIAL_COM1_IOPORT 0x3f8

Definition at line 4 of file serial.h.

4.20.1.2 #define SERIAL_COM2_IOPORT 0x2f8

Definition at line 6 of file serial.h.

4.20.1.3 #define SERIAL_COM3_IOPORT 0x3e8

Definition at line 8 of file serial.h.

4.20.1.4 #define SERIAL_COM4_IOPORT 0x2e8

Definition at line 10 of file serial.h.

4.20.1.5 #define SERIAL_REG_DATA_BUFFER 0

Definition at line 13 of file serial.h.

Referenced by serial_putc().

4.20.1.6 #define SERIAL_REG_DIVISOR_HIGH 1

Definition at line 19 of file serial.h.

Referenced by serial_init().

4.20.1.7 #define SERIAL_REG_DIVISOR_LOW 0

Definition at line 15 of file serial.h.

Referenced by serial_init().

4.20.1.8 #define SERIAL_REG_FIFO_CTRL 2

Definition at line 23 of file serial.h.

Referenced by serial_init().

4.20.1.9 #define SERIAL_REG_INTR_ENABLE 1

Definition at line 17 of file serial.h.

Referenced by serial_init().

4.20.1.10 #define SERIAL_REG_INTR_ID 2

Definition at line 21 of file serial.h.

4.20.1.11 #define SERIAL_REG_LINE_CTRL 3

Definition at line 25 of file serial.h.

Referenced by serial_init().

4.20.1.12 #define SERIAL_REG_LINE_STATUS 5

Definition at line 29 of file serial.h.

Referenced by serial_putc().

4.20.1.13 #define SERIAL_REG_MODEM_CTRL 4

Definition at line 27 of file serial.h.

Referenced by serial_init().

4.20.1.14 #define SERIAL_REG_MODEM_STATUS 6

Definition at line 31 of file serial.h.

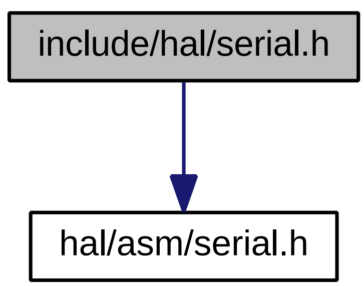
4.20.1.15 #define SERIAL_REG_SCRATCH 7

Definition at line 33 of file serial.h.

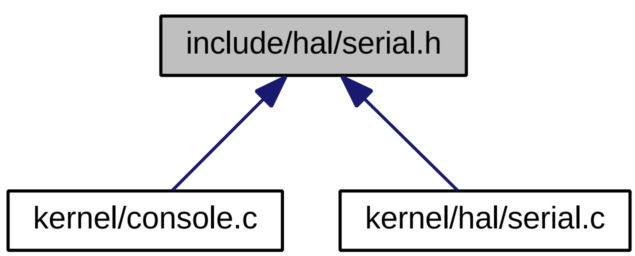
4.21 include/hal/serial.h File Reference

```
#include <hal/asm/serial.h>
```

Include dependency graph for serial.h:



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



Functions

- void **serial_init** (int base_ioport, unsigned int baud_rate)
- void **serial_printn** (int base_ioport, const char ∗message, unsigned int n)
- void **serial_putc** (int base_ioport, char c)

4.21.1 Function Documentation

4.21.1.1 void serial_init (int *base_ioport*, unsigned int *baud_rate*)

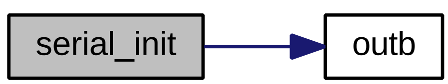
Definition at line 4 of file serial.c.

References outb(), SERIAL_REG_DIVISOR_HIGH, SERIAL_REG_DIVISOR_LOW, SERIAL_REG_FIFO_CTRL, SERIAL_REG_INTR_ENABLE, SERIAL_REG_LINE_CTRL, and SERIAL_REG_MODEM_CTRL.

Referenced by console_init().

```
4                                                           {
5     unsigned int divisor = 115200 / baud_rate;
6
7     /* disable interrupts */
8     outb(base_ioport + SERIAL_REG_INTR_ENABLE,  0);
9
10     /* 8N1, enable DLAB to allow setting baud rate */
11     outb(base_ioport + SERIAL_REG_LINE_CTRL,    0x83);
12
13     /* set baud rate */
14     outb(base_ioport + SERIAL_REG_DIVISOR_LOW,  (divisor & 0xff));
15     outb(base_ioport + SERIAL_REG_DIVISOR_HIGH, ((divisor >> 8) & 0xff));
16
17     /* 8N1, disable DLAB */
18     outb(base_ioport + SERIAL_REG_LINE_CTRL,    0x03);
19
20     /* enable and clear FIFO
21      *
22      * Receive FIFO trigger level is not relevant for us as we are only
23      * transmitting. */
24     outb(base_ioport + SERIAL_REG_FIFO_CTRL,    0x07);
25
26     /* assert DTR and RTS */
27     outb(base_ioport + SERIAL_REG_MODEM_CTRL,   0x03);
28 }
```

Here is the call graph for this function:



4.21.1.2 void serial_printn (int *base_ioport*, const char ∗ *message*, unsigned int *n*)

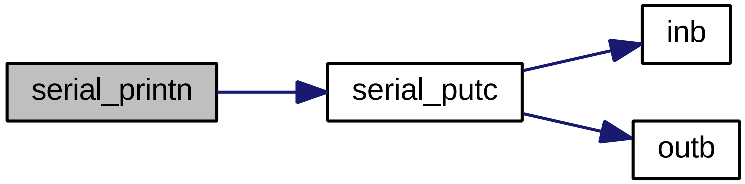
Definition at line 30 of file serial.c.

References serial_putc().

Referenced by console_printn().

```
30                                                                            {
31     int idx;
32
33     for(idx = 0; idx < n; ++idx) {
34         serial_putc(base_ioport, message[idx]);
35     }
36 }
```

Here is the call graph for this function:



4.21.1.3 void serial_putc (int *base_ioport*, char *c*)

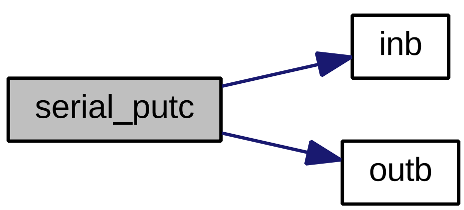
Definition at line 38 of file serial.c.

References inb(), outb(), SERIAL_REG_DATA_BUFFER, and SERIAL_REG_LINE_STATUS.

Referenced by console_putc(), and serial_printn().

```
38                                              {
39      /* wait for the UART to be ready to accept a new character */
40      while( (inb(base_ioport + SERIAL_REG_LINE_STATUS) & 0x20) == 0) {}
41
42      outb(base_ioport + SERIAL_REG_DATA_BUFFER, c);
43 }
```

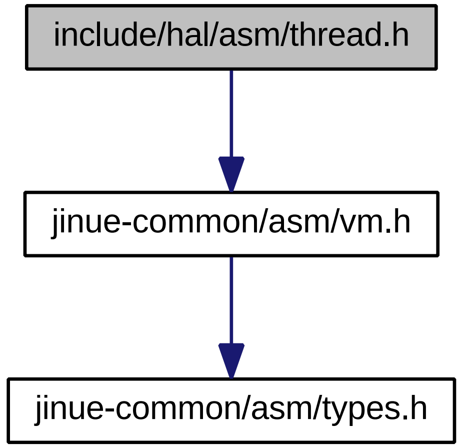
Here is the call graph for this function:



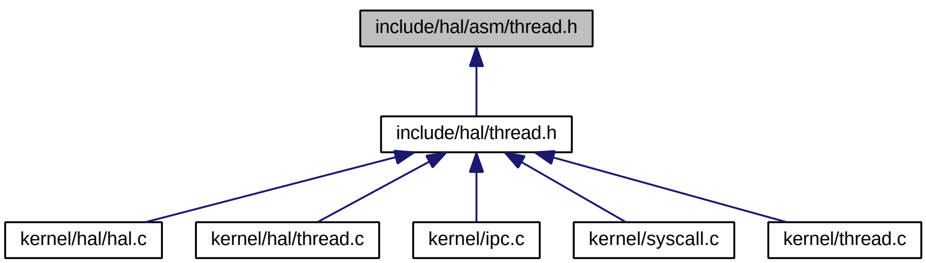
4.22 include/hal/asm/thread.h File Reference

`#include <jinue-common/asm/vm.h>`

Include dependency graph for thread.h:



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



Macros

- #define **THREAD_CONTEXT_SIZE PAGE_SIZE**
- #define **THREAD_CONTEXT_MASK** (∼(**THREAD_CONTEXT_SIZE** - 1))

4.22.1 Macro Definition Documentation

4.22.1.1 #define THREAD_CONTEXT_MASK (∼(THREAD_CONTEXT_SIZE - 1))

Definition at line 40 of file thread.h.

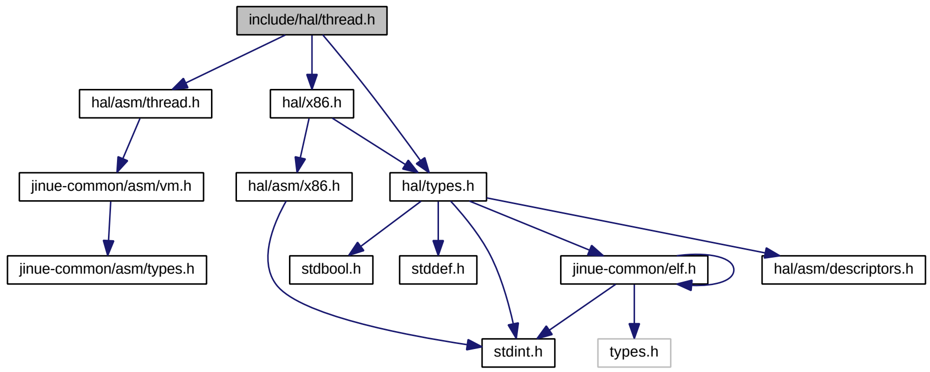
4.22.1.2 #define THREAD_CONTEXT_SIZE PAGE_SIZE

Definition at line 38 of file thread.h.

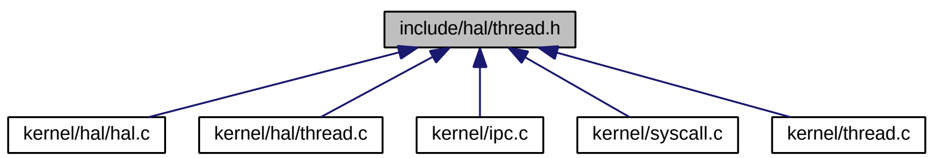
4.23 include/hal/thread.h File Reference

```
#include <hal/asm/thread.h>
#include <hal/x86.h>
#include <types.h>
```

Include dependency graph for thread.h:



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



Functions

- **thread_t** ∗ **thread_page_init** (**addr_t** thread_page, **addr_t** entry, **addr_t** user_stack)
- void **thread_context_switch** (**thread_context_t** ∗from_ctx, **thread_context_t** ∗to_ctx, **bool** destroy_from)

4.23.1 Function Documentation

4.23.1.1 void thread_context_switch (thread_context_t ∗ *from_ctx,* thread_context_t ∗ *to_ctx,* bool *destroy_from*)

ASSERTION: to_ctx argument must not be NULL

ASSERTION: from_ctx argument must not be NULL if destroy_from is true

Definition at line 122 of file thread.c.

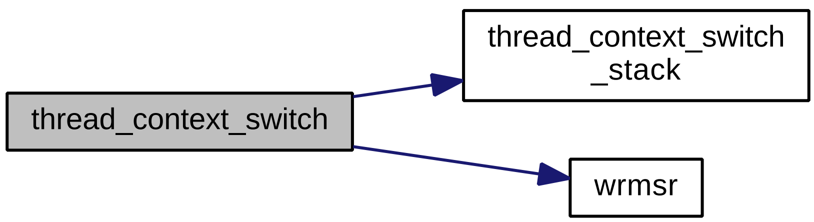
References assert, CPU_FEATURE_SYSENTER, tss_t::esp0, tss_t::esp1, tss_t::esp2, MSR_IA32_SYSENTER_ESP, NULL, thread_context_switch_stack(), and wrmsr().

Referenced by thread_switch().

```
                                               {

        assert(to_ctx != NULL);

        assert(from_ctx != NULL || ! destroy_from);
```

```

    /* nothing to do if this is already the current thread */
    if(from_ctx != to_ctx) {
        /* setup TSS with kernel stack base for this thread context */
        addr_t kernel_stack_base = get_kernel_stack_base(to_ctx);
        tss_t *tss = get_tss();

        tss->esp0 = kernel_stack_base;
        tss->esp1 = kernel_stack_base;
        tss->esp2 = kernel_stack_base;

        /* update kernel stack address for SYSENTER instruction */
        if(cpu_has_feature(CPU_FEATURE_SYSENTER)) {
            wrmsr(MSR_IA32_SYSENTER_ESP, (uint64_t)(uintptr_t)kernel_stack_base);
        }

        /* switch thread context stack */
        thread_context_switch_stack(from_ctx, to_ctx, destroy_from);
    }
}
```

Here is the call graph for this function:



4.23.1.2 thread_t* thread_page_init (addr_t *thread_page*, addr_t *entry*, addr_t *user_stack*)

Definition at line 81 of file thread.c.

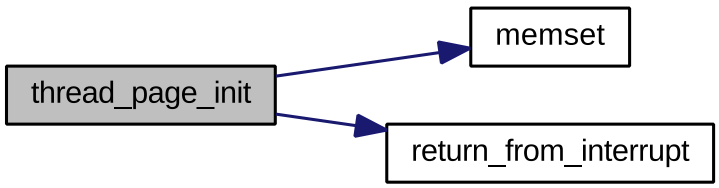
References trapframe_t::cs, trapframe_t::ds, trapframe_t::eflags, trapframe_t::eip, kernel_context_t::eip, trapframe_t::es, trapframe_t::esp, trapframe_t::fs, GDT_USER_CODE, GDT_USER_DATA, trapframe_t::gs, thread_context_t::local_storage_addr, memset(), NULL, return_from_interrupt(), RPL_USER, thread_context_t::saved_stack_pointer, SEG_SELECTOR, trapframe_t::ss, and thread_t::thread_ctx.

Referenced by thread_create(), and thread_create_boot().

```
                                        {

    /* initialize fields */
    thread_t *thread                = (thread_t *)thread_page;
    thread_context_t *thread_ctx    = &thread->thread_ctx;

    thread_ctx->local_storage_addr  = NULL;

    /* setup stack for initial return to user space */
    void *kernel_stack_base = get_kernel_stack_base(thread_ctx);

    trapframe_t *trapframe = (trapframe_t *)kernel_stack_base - 1;

    memset(trapframe, 0, sizeof(trapframe_t));

    trapframe->eip      = (uint32_t)entry;
     trapframe->esp      = (uint32_t)user_stack;
     trapframe->eflags   = 2;
     trapframe->cs       = SEG_SELECTOR(GDT_USER_CODE, RPL_USER);
     trapframe->ss       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);
     trapframe->ds       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);
     trapframe->es       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);
     trapframe->fs       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);
     trapframe->gs       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);
```

```
    kernel_context_t *kernel_context = (kernel_context_t *)trapframe - 1;

    memset(kernel_context, 0, sizeof(kernel_context_t));

    /* This is the address to which thread_context_switch_stack() will return. */
    kernel_context->eip = (uint32_t)return_from_interrupt;

    /* set thread stack pointer */
    thread_ctx->saved_stack_pointer = (addr_t)kernel_context;

    return thread;
}
```

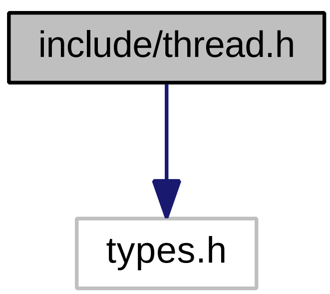
Here is the call graph for this function:



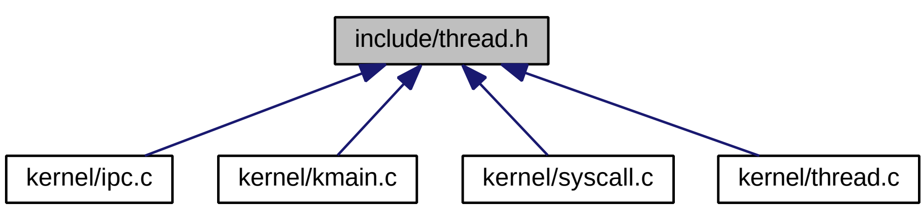
4.24 include/thread.h File Reference

`#include <types.h>`

Include dependency graph for thread.h:



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



Functions

- **thread_t** ∗ **thread_create** (**process_t** ∗process, **addr_t** entry, **addr_t** user_stack)

- **thread_t** ∗ **thread_create_boot** (**process_t** ∗process, **addr_t** entry, **addr_t** user_stack, **boot_alloc_t** ∗boot_-alloc)
- void **thread_destroy** (**thread_t** ∗thread)
- void **thread_ready** (**thread_t** ∗thread)
- void **thread_switch** (**thread_t** ∗from_thread, **thread_t** ∗to_thread, **bool** blocked, **bool** do_destroy)
- void **thread_yield_from** (**thread_t** ∗from_thread, **bool** blocked, **bool** do_destroy)

4.24.1 Function Documentation

4.24.1.1 thread_t∗ thread_create (process_t ∗ *process,* addr_t *entry,* addr_t *user_stack*)

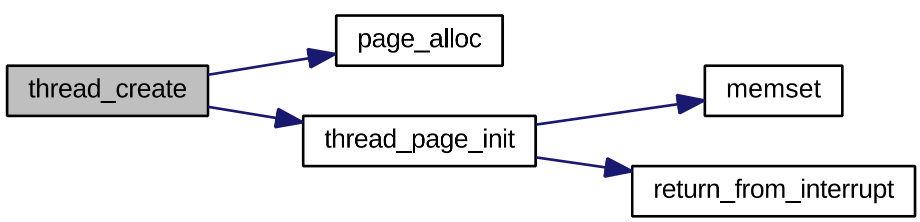
Definition at line 56 of file thread.c.

References NULL, page_alloc(), and thread_page_init().

Referenced by dispatch_syscall().

```
                                    {

    void *thread_page = page_alloc();

    if(thread_page == NULL) {
        return NULL;
    }

    thread_t *thread = thread_page_init(thread_page, entry, user_stack);
    thread_init(thread, process);

    return thread;
}
```

Here is the call graph for this function:



4.24.1.2 thread_t∗ thread_create_boot (process_t ∗ *process,* addr_t *entry,* addr_t *user_stack,* boot_alloc_t ∗ *boot_alloc*)

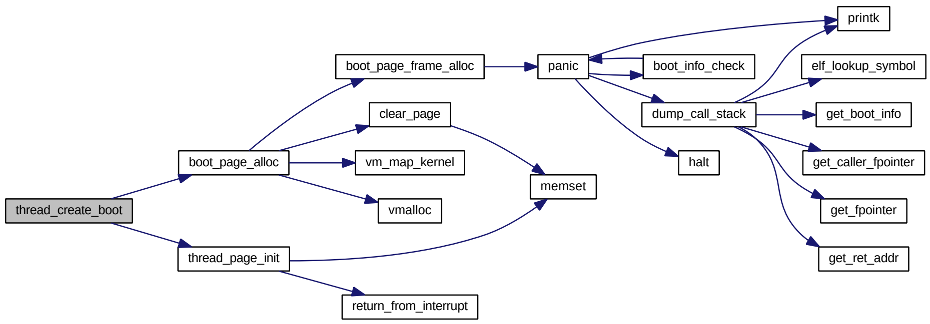
Definition at line 73 of file thread.c.

References boot_page_alloc(), and thread_page_init().

Referenced by kmain().

```
                                    {

    /* The kernel panics if this allocation fails. */
    void *thread_page   = boot_page_alloc(boot_alloc);
    thread_t *thread    = thread_page_init(thread_page, entry, user_stack);
    thread_init(thread, process);

    return thread;
}
```

Here is the call graph for this function:



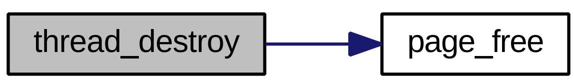
4.24.1.3 void thread_destroy (**thread_t** ∗ ***thread***)

Definition at line 88 of file thread.c.

References page_free(), and PAGE_MASK.

```
88                                        {
89     void * thread_page = (void *)((uintptr_t)thread & ~PAGE_MASK);
90     page_free(thread_page);
91 }
```

Here is the call graph for this function:



4.24.1.4 void thread_ready (**thread_t** ∗ ***thread***)

Definition at line 93 of file thread.c.

References thread_t::thread_list.

Referenced by thread_switch().

```
93                                      {
94     /* add thread to the tail of the ready list to give other threads a chance
95      * to run */
96     jinue_list_enqueue(&ready_list, &thread->thread_list);
97 }
```

4.24.1.5 void thread_switch (**thread_t** ∗ ***from_thread,*** **thread_t** ∗ ***to_thread,*** **bool** ***blocked,*** **bool** ***do_destroy***)

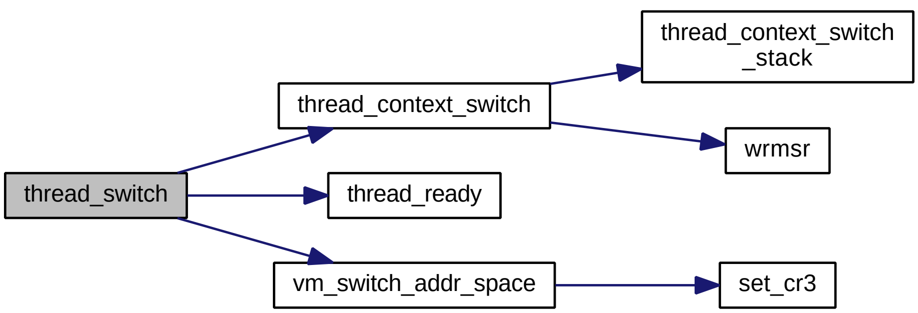
Definition at line 99 of file thread.c.

References process_t::addr_space, NULL, thread_t::process, thread_context_switch(), thread_t::thread_ctx, thread_-ready(), and vm_switch_addr_space().

Referenced by ipc_receive(), ipc_reply(), ipc_send(), and thread_yield_from().

```
                             {

    if(to_thread != from_thread) {
        thread_context_t    *from_context;
        process_t           *from_process;

        if(from_thread == NULL) {
            from_context = NULL;
            from_process = NULL;
        }
        else {
            from_context = &from_thread->thread_ctx;
            from_process = from_thread->process;

            /* Put the the thread we are switching away from (the current thread)
             * back into the ready list, unless it just blocked or it is being
             * destroyed. */
            if(! (do_destroy || blocked)) {
                thread_ready(from_thread);
            }
        }

        if(from_process != to_thread->process) {
            vm_switch_addr_space(
                    &to_thread->process->addr_space,
                    get_cpu_local_data()
            );
        }

        thread_context_switch(
            from_context,
            &to_thread->thread_ctx,
            do_destroy);
    }
}
```

Here is the call graph for this function:



4.24.1.6 void thread_yield_from (thread_t * *from_thread*, bool *blocked*, bool *do_destroy*)

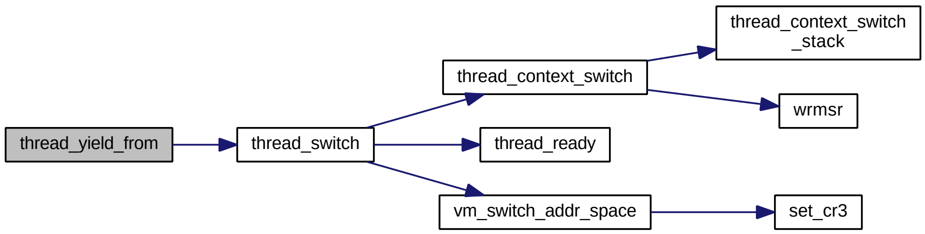
Definition at line 163 of file thread.c.

References thread_switch().

Referenced by dispatch_syscall(), ipc_receive(), ipc_send(), and kmain().

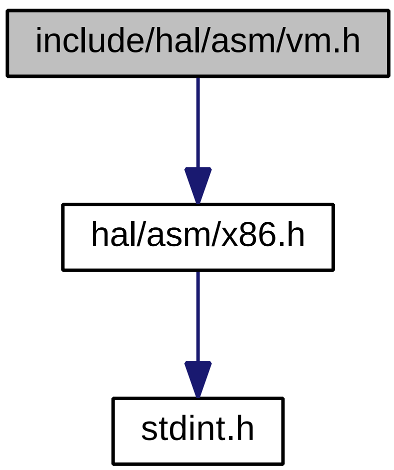
```
163                                                                              {
164     bool from_can_run = ! (blocked || do_destroy);
165 
166     thread_switch(
167             from_thread,
168             reschedule(from_thread, from_can_run),
169             blocked,
170             do_destroy);
171 }
```

Here is the call graph for this function:

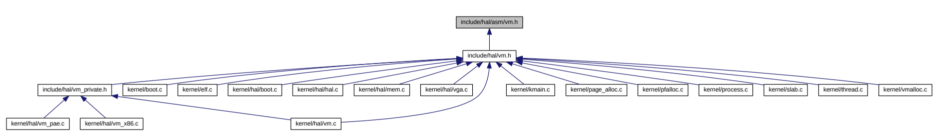


4.25 include/hal/asm/vm.h File Reference

`#include <hal/asm/x86.h>`
Include dependency graph for vm.h:



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



Macros

- #define **VM_FLAG_PRESENT X86_PTE_PRESENT**

  page is present in memory
- #define **VM_FLAG_READ_ONLY** 0

  page is read only
- #define **VM_FLAG_READ_WRITE X86_PTE_READ_WRITE**

  page is read/write accessible
- #define **VM_FLAG_KERNEL X86_PTE_GLOBAL**

kernel mode page

- #define **VM_FLAG_USER X86_PTE_USER**

  user mode page
- #define **VM_FLAG_ACCESSED X86_PTE_ACCESSED**

  page was accessed (read)
- #define **VM_FLAG_DIRTY X86_PTE_DIRTY**

  page was written to

4.25.1 Macro Definition Documentation

4.25.1.1 #define VM_FLAG_ACCESSED X86_PTE_ACCESSED

page was accessed (read)

Definition at line 53 of file vm.h.

4.25.1.2 #define VM_FLAG_DIRTY X86_PTE_DIRTY

page was written to

Definition at line 56 of file vm.h.

4.25.1.3 #define VM_FLAG_KERNEL X86_PTE_GLOBAL

kernel mode page

Definition at line 47 of file vm.h.

Referenced by vm_boot_init(), and vm_map_kernel().

4.25.1.4 #define VM_FLAG_PRESENT X86_PTE_PRESENT

page is present in memory

Definition at line 38 of file vm.h.

Referenced by vm_pae_create_initial_addr_space(), vm_pae_destroy_addr_space(), and vm_pae_lookup_page_-directory().

4.25.1.5 #define VM_FLAG_READ_ONLY 0

page is read only

Definition at line 41 of file vm.h.

Referenced by elf_load().

4.25.1.6 #define VM_FLAG_READ_WRITE X86_PTE_READ_WRITE

page is read/write accessible

Definition at line 44 of file vm.h.

Referenced by add_page_frame(), boot_page_alloc(), boot_page_alloc_image(), elf_load(), elf_setup_stack(), vm-_boot_init(), vm_clone_page_directory(), vm_destroy_page_directory(), vm_init_initial_page_directory(), vm_pae_-lookup_page_directory(), and vm_x86_lookup_page_directory().

4.25.1.7 #define VM_FLAG_USER X86_PTE_USER

user mode page

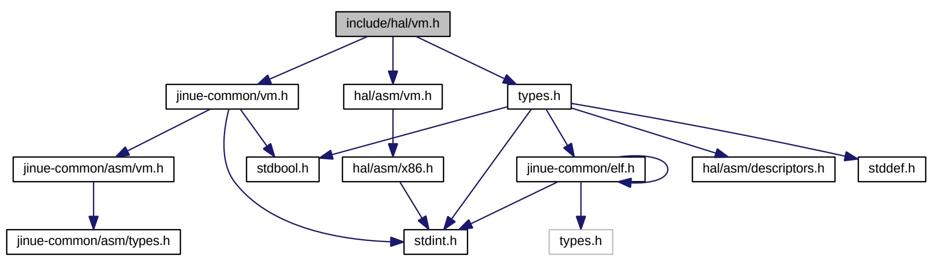
Definition at line 50 of file vm.h.

Referenced by vm_map_user().

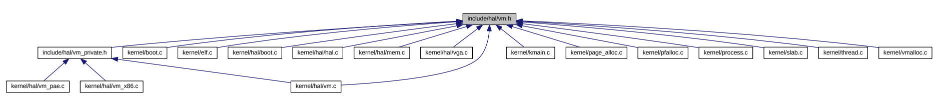
4.26 include/hal/vm.h File Reference

```
#include <jinue-common/vm.h>
#include <hal/asm/vm.h>
#include <types.h>
```

Include dependency graph for vm.h:



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



Macros

- #define **EARLY_PHYS_TO_VIRT**(x) (((**uintptr_t**)(x)) + **KLIMIT**)

  *This header file contains the public interface of the low-level page table management code located in **hal/vm.c** (p. 331) and **hal/vm_pae.c** (p. 342).*
- #define **EARLY_VIRT_TO_PHYS**(x) (((**uintptr_t**)(x)) - **KLIMIT**)

  convert a virtual address to a physical address before the switch to the first address space
- #define **EARLY_PTR_TO_PHYS_ADDR**(x) ((**kern_paddr_t**)**EARLY_VIRT_TO_PHYS**(x))

  convert a pointer to a page frame address (early mappings)
- #define **ADDR_4GB UINT64_C**(0x100000000)

Functions

- void **vm_boot_init** (const **boot_info_t** ∗**boot_info**, **bool** use_pae, **cpu_data_t** ∗cpu_data, **boot_alloc_t** ∗boot-_alloc)

- void **vm_boot_postinit** (const **boot_info_t** ∗**boot_info**, **boot_alloc_t** ∗boot_alloc, **bool** use_pae)
- void **vm_map_kernel** (**addr_t** vaddr, **kern_paddr_t** paddr, int flags)
- void **vm_map_user** (**addr_space_t** ∗addr_space, **addr_t** vaddr, **user_paddr_t** paddr, int flags)
- void **vm_unmap_kernel** (**addr_t** addr)
- void **vm_unmap_user** (**addr_space_t** ∗addr_space, **addr_t** addr)
- **kern_paddr_t vm_lookup_kernel_paddr** (**addr_t** addr)
- void **vm_change_flags** (**addr_space_t** ∗addr_space, **addr_t** addr, int flags)
- void **vm_map_early** (**addr_t** vaddr, **kern_paddr_t** paddr, int flags)
- **addr_space_t** ∗ **vm_create_addr_space** (**addr_space_t** ∗addr_space)
- **addr_space_t** ∗ **vm_create_initial_addr_space** (**bool** use_pae, **boot_alloc_t** ∗boot_alloc)
- void **vm_destroy_addr_space** (**addr_space_t** ∗addr_space)
- void **vm_switch_addr_space** (**addr_space_t** ∗addr_space, **cpu_data_t** ∗cpu_data)

Variables

- **addr_space_t initial_addr_space**

4.26.1 Macro Definition Documentation

4.26.1.1 #define ADDR_4GB UINT64_C(0x100000000)

Definition at line 51 of file vm.h.

4.26.1.2 #define EARLY_PHYS_TO_VIRT(*x*) (((uintptr_t)(x)) + KLIMIT)

This header file contains the public interface of the low-level page table management code located in **hal/vm.c** (p. 331) and **hal/vm_pae.c** (p. 342).

convert a physical address to a virtual address before the switch to the first address space

Definition at line 43 of file vm.h.

Referenced by vm_pae_create_initial_addr_space().

4.26.1.3 #define EARLY_PTR_TO_PHYS_ADDR(*x*) ((kern_paddr_t)EARLY_VIRT_TO_PHYS(x))

convert a pointer to a page frame address (early mappings)

Definition at line 49 of file vm.h.

Referenced by boot_page_alloc_early(), elf_load(), vm_boot_init(), vm_init_initial_page_directory(), vm_pae_create_-initial_addr_space(), and vm_x86_create_initial_addr_space().

4.26.1.4 #define EARLY_VIRT_TO_PHYS(*x*) (((uintptr_t)(x)) - KLIMIT)

convert a virtual address to a physical address before the switch to the first address space

Definition at line 46 of file vm.h.

Referenced by mem_check_memory(), vm_map_early(), vm_pae_create_initial_addr_space(), and vm_x86_create_-initial_addr_space().

4.26.2 Function Documentation

4.26.2.1 void vm_boot_init (const **boot_info_t** ∗ *boot_info*, **bool** *use_pae*, **cpu_data_t** ∗ *cpu_data*, **boot_alloc_t** ∗ *boot_alloc*)

below this point, it is no longer safe to call pfalloc_early()

Definition at line 57 of file vm.c.

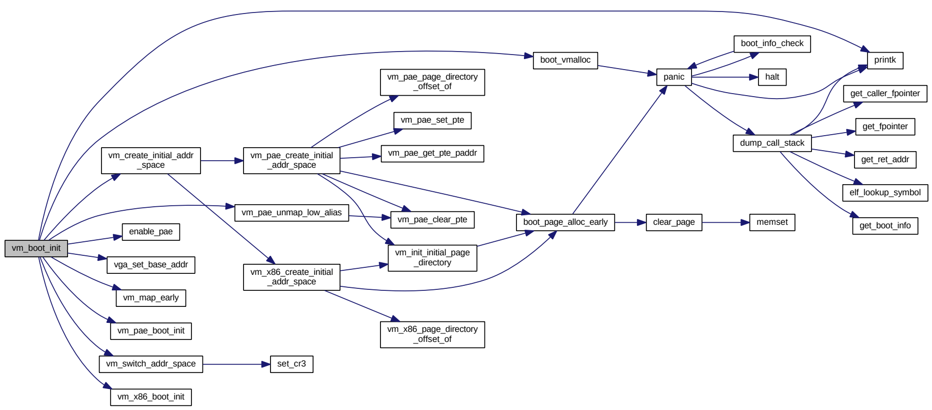
References boot_vmalloc(), addr_space_t::cr3, EARLY_PTR_TO_PHYS_ADDR, enable_pae(), boot_info_t::image_start, boot_alloc_t::its_early, PAGE_SIZE, printk(), vga_set_base_addr(), VGA_TEXT_VID_BASE, VGA_TEXT_VID_TOP, vm_create_initial_addr_space(), VM_FLAG_KERNEL, VM_FLAG_READ_WRITE, vm_map_early(), vm_pae_boot_init(), vm_pae_unmap_low_alias(), vm_switch_addr_space(), and vm_x86_boot_init().

Referenced by hal_init().

```
                                        {

    addr_t addr;

    if(use_pae) {
        printk("Enabling Physical Address Extension (PAE).\n");
        vm_pae_boot_init();
    }
    else {
        vm_x86_boot_init();
    }

    pgtable_format_pae = use_pae;

    /* create initial address space */
    addr_space_t *addr_space = vm_create_initial_addr_space(use_pae, boot_alloc);

    boot_alloc->its_early = false;

    /* perform 1:1 mapping of kernel image and data */
    for(addr = (addr_t)boot_info->image_start; addr < boot_alloc->kernel_vm_top; addr +=
   PAGE_SIZE) {
        vm_map_early(addr, EARLY_PTR_TO_PHYS_ADDR(addr), VM_FLAG_KERNEL |
   VM_FLAG_READ_WRITE);
    }

    /* map VGA text buffer in the new address space
     *
     * This is a good place to do this because:
     *
     * 1) It is our last chance to allocate a continuous region of virtual memory.
     *    Once the page allocator is initialized (see call to vm_alloc_init_allocator()
     *    below) and we start using vm_alloc() to allocate memory, pages can only
     *    be allocated one at a time.
     *
     * 2) Doing this last makes things simpler because this is the only place where
     *    we have to allocate a continuous region of virtual memory but no physical
     *    memory to back it. To allocate it, we just have to increase kernel_vm_top,
     *    which represents the end of the virtual memory region that is used by the
     *    kernel. */
    addr_t vga_text_base;
    kern_paddr_t paddr = VGA_TEXT_VID_BASE;

    while(paddr < VGA_TEXT_VID_TOP) {
        /* Pages allocated by successive calls to vmalloc_boot() are guaranteed
         * to be contiguous. */
        addr = boot_vmalloc(boot_alloc);

        if(paddr == VGA_TEXT_VID_BASE) {
            /* First iteration */
            vga_text_base = addr;
        }

        vm_map_early(addr, paddr, VM_FLAG_KERNEL | VM_FLAG_READ_WRITE);
        paddr           += PAGE_SIZE;
    }

    /* remap VGA text buffer
     *
     * Note: after the call to vga_set_base_addr() below until we switch to the
```

```
     * new address space, VGA output is not possible. Calling printk() will cause
     * a kernel panic due to a page fault (and the panic handler calls printk()). */
    printk("Remapping text video memory at 0x%x\n", vga_text_base);

    vga_set_base_addr(vga_text_base);

    if(use_pae) {
        /* If we are enabling PAE, this is where the switch to the new page
         * tables actually happens instead of at the call to vm_switch_addr_space()
         * as would be expected. */
        enable_pae(addr_space->cr3);

        /* Now that PAE has been enabled, there is no need to ever disable paging
         * again, so the low alias for the first 2MB of RAM can be unmapped. This
         * is only relevant for PAE because, for the non-PAE case, this low alias
         * is just never set up in the initial address space in the first place,
         * which means there is no longer a low alias once vm_switch_addr_space()
         * is called below.
         *
         * This call to vm_pae_unmap_low_alias() does not do any TLB invalidation
         * but this is fine because the call to vm_switch_addr_space() below
         * reloads CR3.*/
        vm_pae_unmap_low_alias(addr_space);
    }

    /* switch to new address space */
    vm_switch_addr_space(addr_space, cpu_data);
}
```

Here is the call graph for this function:



4.26.2.2 void vm_boot_postinit (const boot_info_t * *boot_info*, boot_alloc_t * *boot_alloc*, bool *use_pae*)

Definition at line 149 of file vm.c.

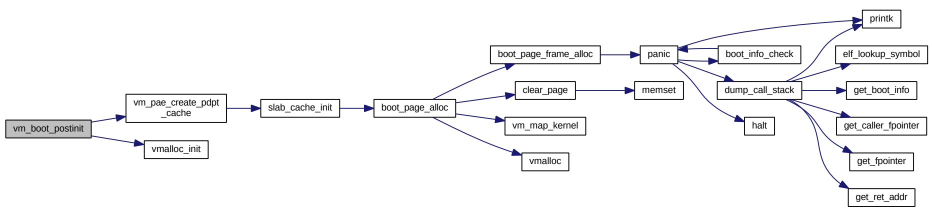
References KERNEL_IMAGE_END, KERNEL_PREALLOC_LIMIT, MB, vm_pae_create_pdpt_cache(), and vmalloc_init().

Referenced by hal_init().

```
149                                                                                   {
150     /* initialize global page allocator (region starting at KLIMIT)
151      *
152      * TODO Some work needs to be done in the page allocator to support allocating
153      * up to the top of memory (i.e. 0x100000000, which cannot be represented on
154      * 32 bits). In the mean time, we leave a 4MB (one block) gap. */
155     vmalloc_init(
156             (addr_t)KERNEL_IMAGE_END,
157             (addr_t)0 - 4 * MB,
158             (addr_t)KERNEL_PREALLOC_LIMIT,
159             boot_alloc);
```

```

    /* create slab cache to allocate PDPTs
     *
     * This must be done after the global page allocator has been initialized
     * because the slab allocator needs to allocate a slab to allocate the new
     * slab cache on the slab cache cache.
     *
     * This must be done before the first time vm_create_addr_space() is called. */
    if(use_pae) {
        vm_pae_create_pdpt_cache(boot_alloc);
    }
}
```

Here is the call graph for this function:



4.26.2.3 void vm_change_flags (addr_space_t ∗ *addr_space,* addr_t *addr,* int *flags*)

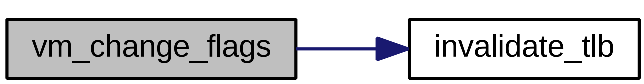
ASSERTION: there is a page table entry marked present for this address

Definition at line 463 of file vm.c.

References assert, invalidate_tlb(), and NULL.

```
463                                                                                  {
464     pte_t *pte = vm_lookup_page_table_entry(addr_space, addr, false);
465
467     assert(pte != NULL && (get_pte_flags(pte) & VM_FLAG_PRESENT));
468
469     /* perform the flags change */
470     set_pte_flags(pte, flags | VM_FLAG_PRESENT);
471
472     vm_free_page_table_entry(addr, pte);
473
474     /* invalidate TLB entry for the affected page */
475     invalidate_tlb(addr);
476 }
```

Here is the call graph for this function:



4.26.2.4 addr_space_t∗ vm_create_addr_space (addr_space_t ∗ *addr_space*)

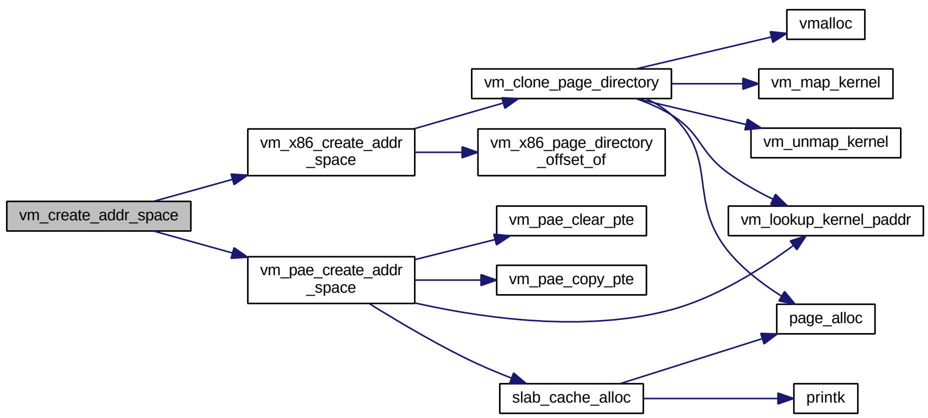
Definition at line 520 of file vm.c.

References vm_pae_create_addr_space(), and vm_x86_create_addr_space().

Referenced by process_create().

```
                                                          {
    if(pgtable_format_pae) {
        return vm_pae_create_addr_space(addr_space);
    }
    else {
        return vm_x86_create_addr_space(addr_space);
    }
}
```

Here is the call graph for this function:



4.26.2.5 addr_space_t∗ vm_create_initial_addr_space (bool *use_pae*, boot_alloc_t ∗ *boot_alloc*)

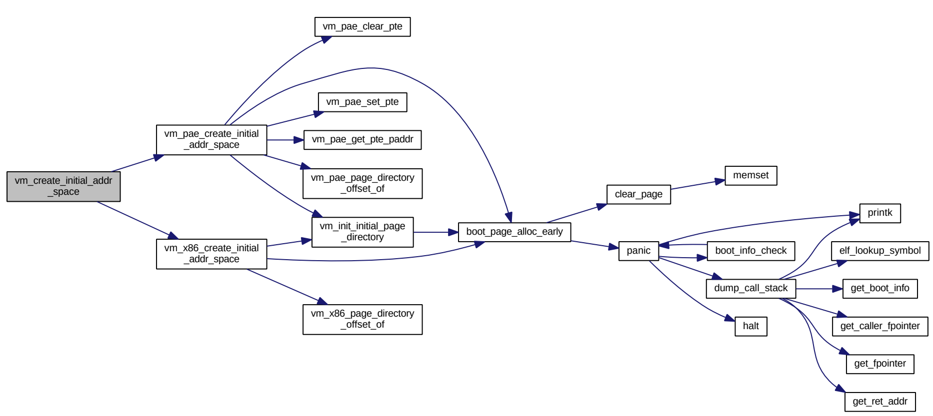
Definition at line 572 of file vm.c.

References vm_pae_create_initial_addr_space(), and vm_x86_create_initial_addr_space().

Referenced by vm_boot_init().

```
                                    {

    if(use_pae) {
        return vm_pae_create_initial_addr_space(boot_alloc);
    }
    else {
        return vm_x86_create_initial_addr_space(boot_alloc);
    }
}
```

Here is the call graph for this function:



4.26.2.6 void vm_destroy_addr_space (addr_space_t * *addr_space*)

ASSERTION: address space must not be NULL

ASSERTION: the initial address space should not be destroyed

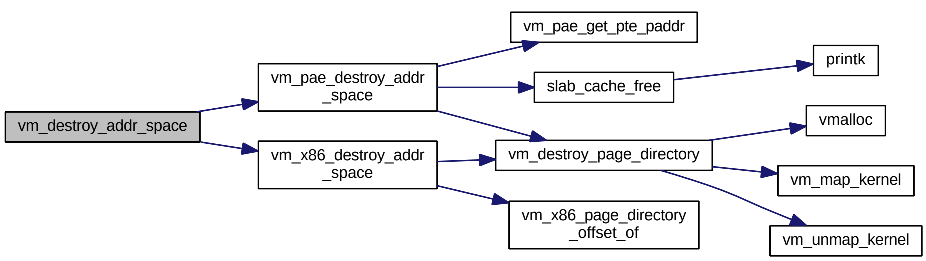
ASSERTION: the current address space should not be destroyed

Definition at line 603 of file vm.c.

References assert, NULL, vm_pae_destroy_addr_space(), and vm_x86_destroy_addr_space().

```
603                                                     {
605     assert(addr_space != NULL);
606
608     assert(addr_space != &initial_addr_space);
609
611     assert( addr_space != get_current_addr_space() );
612
613     if(pgtable_format_pae) {
614         vm_pae_destroy_addr_space(addr_space);
615     }
616     else {
617         vm_x86_destroy_addr_space(addr_space);
618     }
619 }
```

Here is the call graph for this function:



4.26.2.7 kern_paddr_t vm_lookup_kernel_paddr (addr_t *addr*)

ASSERTION: there is a page table entry marked present for this address

Definition at line 450 of file vm.c.

References assert, and NULL.

Referenced by hal_init(), remove_page_frame(), vm_clone_page_directory(), and vm_pae_create_addr_space().

```
450                                                 {
451     pte_t *pte = vm_lookup_page_table_entry(NULL, addr, false);
452
454     assert(pte != NULL && (get_pte_flags(pte) & VM_FLAG_PRESENT));
455
456     kern_paddr_t paddr = (kern_paddr_t)get_pte_paddr(pte);
457
458     vm_free_page_table_entry(addr, pte);
459
460     return paddr;
461 }
```

4.26.2.8 void vm_map_early (addr_t *vaddr*, kern_paddr_t *paddr*, int *flags*)

ASSERTION: we are within the mapping set up by the setup code

ASSERTION: we assume vaddr is aligned on a page boundary

Definition at line 478 of file vm.c.

References assert, EARLY_VIRT_TO_PHYS, page_number_of, and page_offset_of.

Referenced by vm_boot_init().

```
478                                                         {
480     assert( is_early_pointer(vaddr) );
481 
483     assert( page_offset_of(vaddr) == 0 );
484 
485     pte_t *pte = get_pte_with_offset(global_page_tables, page_number_of(
      EARLY_VIRT_TO_PHYS((uintptr_t)vaddr) ));
486     set_pte(pte, paddr, flags | VM_FLAG_PRESENT);
487 }
```

4.26.2.9 void vm_map_kernel (addr_t *vaddr*, kern_paddr_t *paddr*, int *flags*)

Definition at line 434 of file vm.c.

References NULL, and VM_FLAG_KERNEL.

Referenced by add_page_frame(), boot_page_alloc(), boot_page_alloc_image(), elf_setup_stack(), vm_clone_page_-directory(), vm_destroy_page_directory(), vm_pae_lookup_page_directory(), and vm_x86_lookup_page_directory().

```
434                                                          {
435     vm_map(NULL, vaddr, paddr, flags | VM_FLAG_KERNEL);
436 }
```

4.26.2.10 void vm_map_user (addr_space_t * *addr_space*, addr_t *vaddr*, user_paddr_t *paddr*, int *flags*)

Definition at line 438 of file vm.c.

References VM_FLAG_USER.

Referenced by elf_load(), and elf_setup_stack().

```
438                                                                                    {
439     vm_map(addr_space, vaddr, paddr, flags | VM_FLAG_USER);
440 }
```

4.26.2.11 void vm_switch_addr_space (addr_space_t * *addr_space*, cpu_data_t * *cpu_data*)

Definition at line 621 of file vm.c.

References addr_space_t::cr3, cpu_data_t::current_addr_space, and set_cr3().

Referenced by thread_switch(), and vm_boot_init().

```
621                                                                     {
622     set_cr3(addr_space->cr3);
623 
624     cpu_data->current_addr_space = addr_space;
625 }
```

Here is the call graph for this function:

4.26.2.12 void vm_unmap_kernel (addr_t *addr*)

Definition at line 442 of file vm.c.

References NULL.

Referenced by elf_setup_stack(), remove_page_frame(), vm_clone_page_directory(), and vm_destroy_page_-directory().

```
442                                 {
443     vm_unmap(NULL, addr);
444 }
```

4.26.2.13 void vm_unmap_user (addr_space_t * *addr_space*, addr_t *addr*)

Definition at line 446 of file vm.c.

```
446                                                         {
447     vm_unmap(addr_space, addr);
448 }
```

4.26.3 Variable Documentation

4.26.3.1 addr_space_t initial_addr_space

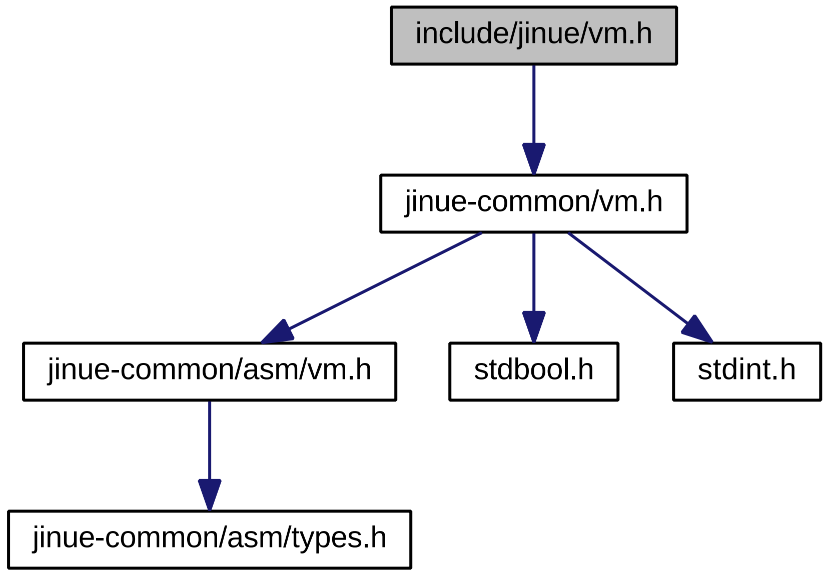
Definition at line 51 of file vm.c.

Referenced by process_create_initial(), vm_pae_create_addr_space(), vm_pae_create_initial_addr_space(), vm_x86-_create_addr_space(), and vm_x86_create_initial_addr_space().

4.27 include/jinue/vm.h File Reference

```
#include <jinue-common/vm.h>
```

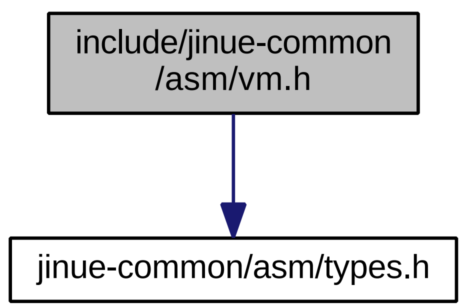
Include dependency graph for vm.h:



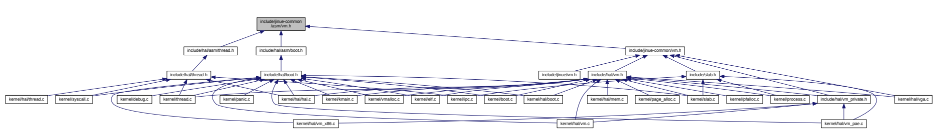
4.28 include/jinue-common/asm/vm.h File Reference

`#include <jinue-common/asm/types.h>`

Include dependency graph for vm.h:



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



Macros

- #define **PAGE_BITS** 12

  number of bits in virtual address for offset inside page

- #define **PAGE_SIZE** (1<<**PAGE_BITS**) /∗ 4096 ∗/

  size of page
- #define **PAGE_MASK** (**PAGE_SIZE** - 1)

  bit mask for offset in page
- #define **KLIMIT** 0xc0000000

  The virtual address range starting at KLIMIT is reserved by the kernel.
- #define **KERNEL_EARLY_LIMIT** (**KLIMIT** + 2 ∗ **MB**)

  limit of initial mapping performed by the 32-bit setup code
- #define **KERNEL_IMAGE_END** (**KLIMIT** + 16 ∗ **MB**)

  limit of the kernel image region
- #define **KERNEL_PREALLOC_LIMIT** (**KLIMIT** + 32 ∗ **MB**)

  limit up to which page tables are preallocated during kernel initialization
- #define **STACK_BASE KLIMIT**

  stack base address (stack top)
- #define **STACK_SIZE** (8 ∗ **PAGE_SIZE**)

  initial stack size
- #define **STACK_START** (**STACK_BASE** - **STACK_SIZE**)

  initial stack lower address

4.28.1 Macro Definition Documentation

4.28.1.1 #define KERNEL_EARLY_LIMIT (KLIMIT + 2 ∗ MB)

limit of initial mapping performed by the 32-bit setup code

Definition at line 53 of file vm.h.

Referenced by mem_check_memory().

4.28.1.2 #define KERNEL_IMAGE_END (KLIMIT + 16 ∗ MB)

limit of the kernel image region

Definition at line 56 of file vm.h.

Referenced by vm_boot_postinit().

4.28.1.3 #define KERNEL_PREALLOC_LIMIT (KLIMIT + 32 ∗ MB)

limit up to which page tables are preallocated during kernel initialization

Definition at line 59 of file vm.h.

Referenced by vm_boot_postinit(), vm_pae_create_initial_addr_space(), and vm_x86_create_initial_addr_space().

4.28.1.4 #define KLIMIT 0xc0000000

The virtual address range starting at KLIMIT is reserved by the kernel.

The region above KLIMIT has the same mapping in all address spaces. KLIMIT must be aligned on a page directory boundary in PAE mode.

Definition at line 50 of file vm.h.

Referenced by vm_pae_create_addr_space(), vm_pae_create_initial_addr_space(), vm_pae_destroy_addr_space(), vm_x86_create_addr_space(), vm_x86_create_initial_addr_space(), and vm_x86_destroy_addr_space().

4.28.1.5 #define PAGE_BITS 12

number of bits in virtual address for offset inside page

Definition at line 39 of file vm.h.

4.28.1.6 #define PAGE_MASK (PAGE_SIZE - 1)

bit mask for offset in page

Definition at line 45 of file vm.h.

Referenced by boot_page_alloc_early(), thread_destroy(), vm_pae_get_pte_flags(), vm_pae_get_pte_paddr(), vm_-pae_set_pte_flags(), vm_x86_get_pte_flags(), vm_x86_get_pte_paddr(), and vm_x86_set_pte_flags().

4.28.1.7 #define PAGE_SIZE (1<<PAGE_BITS) /* 4096 */

size of page

Definition at line 42 of file vm.h.

Referenced by boot_page_alloc_early(), boot_page_frame_alloc(), boot_vmalloc(), clear_page(), elf_load(), elf_setup-_stack(), vm_boot_init(), and vm_pae_lookup_page_directory().

4.28.1.8 #define STACK_BASE KLIMIT

stack base address (stack top)

Definition at line 62 of file vm.h.

Referenced by elf_setup_stack().

4.28.1.9 #define STACK_SIZE (8 * PAGE_SIZE)

initial stack size

Definition at line 65 of file vm.h.

4.28.1.10 #define STACK_START (STACK_BASE - STACK_SIZE)

initial stack lower address

Definition at line 68 of file vm.h.

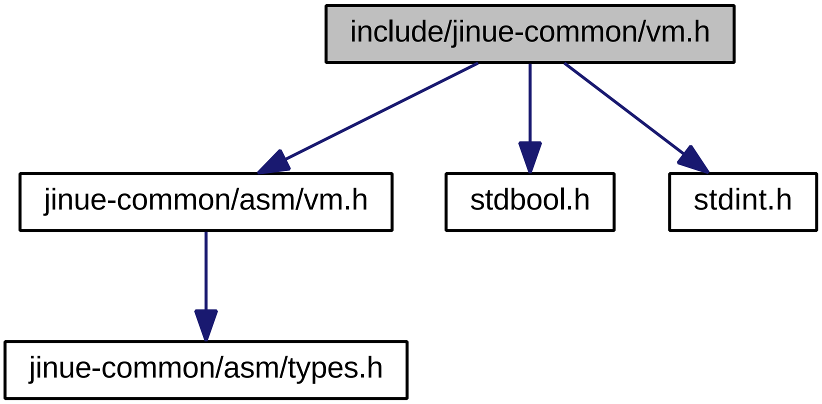
Referenced by elf_setup_stack().

4.29 include/jinue-common/vm.h File Reference

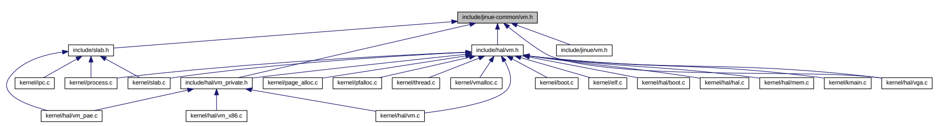
```
#include <jinue-common/asm/vm.h>
```

```
#include <stdbool.h>
#include <stdint.h>
```

Include dependency graph for vm.h:



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



Macros

- #define **page_offset_of**(x) ((**uintptr_t**)(x) & **PAGE_MASK**)

  byte offset in page of virtual (linear) address
- #define **page_address_of**(x) ((**uintptr_t**)(x) & ∼**PAGE_MASK**)

  address of the page that contains a virtual (linear) address
- #define **page_number_of**(x) ((**uintptr_t**)(x) >> **PAGE_BITS**)

  sequential page number of virtual (linear) address

4.29.1 Macro Definition Documentation

4.29.1.1 #define page_address_of(*x*) ((uintptr_t)(x) & ∼PAGE_MASK)

address of the page that contains a virtual (linear) address

Definition at line 45 of file vm.h.

Referenced by vm_pae_create_addr_space().

4.29.1.2 #define page_number_of(*x*) ((uintptr_t)(x) >> PAGE_BITS)

sequential page number of virtual (linear) address

Definition at line 48 of file vm.h.

Referenced by vm_map_early().

4.29.1.3 #define page_offset_of(*x*) ((uintptr_t)(x) & PAGE_MASK)

byte offset in page of virtual (linear) address

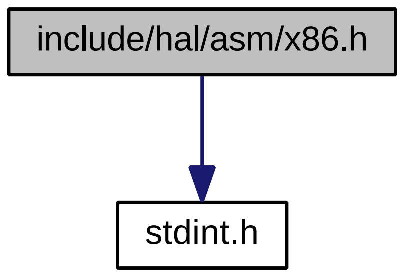
Definition at line 42 of file vm.h.

Referenced by boot_info_check(), vm_map_early(), and vm_pae_create_addr_space().

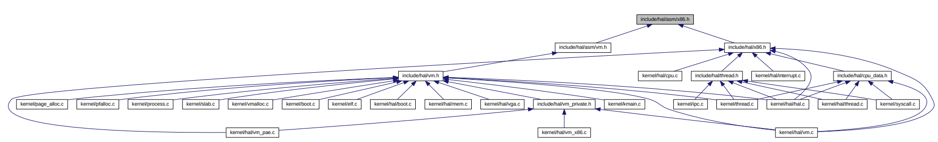
4.30 include/hal/asm/x86.h File Reference

`#include <stdint.h>`

Include dependency graph for x86.h:



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



Macros

- #define **X86_CR0_WP** (1<<16)

  CR0 register: Write Protect.
- #define **X86_CR0_PG** (1<<31)

  CR0 register: Paging.
- #define **X86_CR4_PSE** (1<<4)

  CR4 register: Page Size Extension (PSE)
- #define **X86_CR4_PAE** (1<<5)

  CR4 register: Physical Address Extension (PAE)
- #define **X86_CR4_PGE** (1<<7)

  CR4 register: global pages.
- #define **X86_PTE_PRESENT** (1<< 0)

  page is present in memory
- #define **X86_PTE_READ_WRITE** (1<< 1)

  page is read/write accessible
- #define **X86_PTE_USER** (1<< 2)

  user mode page

- #define **X86_PTE_WRITE_THROUGH** (1<< 3)

  write-through cache policy for page
- #define **X86_PTE_CACHE_DISABLE** (1<< 4)

  uncached page
- #define **X86_PTE_ACCESSED** (1<< 5)

  page was accessed (read)
- #define **X86_PTE_DIRTY** (1<< 6)

  page was written to
- #define **X86_PDE_PAGE_SIZE** (1<< 7)

  page directory entry describes a 4M page
- #define **X86_PTE_GLOBAL** (1<< 8)

  page is global (mapped in every address space)
- #define **X86_PTE_NX** (**UINT64_C**(1)<< 63)

  do not execute bit

4.30.1 Macro Definition Documentation

4.30.1.1 #define X86_CR0_PG (1<<31)

CR0 register: Paging.

Definition at line 43 of file x86.h.

4.30.1.2 #define X86_CR0_WP (1<<16)

CR0 register: Write Protect.

Definition at line 40 of file x86.h.

4.30.1.3 #define X86_CR4_PAE (1<<5)

CR4 register: Physical Address Extension (PAE)

Definition at line 50 of file x86.h.

4.30.1.4 #define X86_CR4_PGE (1<<7)

CR4 register: global pages.

Definition at line 53 of file x86.h.

4.30.1.5 #define X86_CR4_PSE (1<<4)

CR4 register: Page Size Extension (PSE)

Definition at line 47 of file x86.h.

4.30.1.6 #define X86_PDE_PAGE_SIZE (1<< 7)

page directory entry describes a 4M page

Definition at line 78 of file x86.h.

4.30.1.7 #define X86_PTE_ACCESSED (1<< 5)

page was accessed (read)

Definition at line 72 of file x86.h.

4.30.1.8 #define X86_PTE_CACHE_DISABLE (1<< 4)

uncached page

Definition at line 69 of file x86.h.

4.30.1.9 #define X86_PTE_DIRTY (1<< 6)

page was written to

Definition at line 75 of file x86.h.

4.30.1.10 #define X86_PTE_GLOBAL (1<< 8)

page is global (mapped in every address space)

Definition at line 81 of file x86.h.

4.30.1.11 #define X86_PTE_NX (UINT64_C(1)<< 63)

do not execute bit

Definition at line 84 of file x86.h.

4.30.1.12 #define X86_PTE_PRESENT (1<< 0)

page is present in memory

Definition at line 57 of file x86.h.

4.30.1.13 #define X86_PTE_READ_WRITE (1<< 1)

page is read/write accessible

Definition at line 60 of file x86.h.

4.30.1.14 #define X86_PTE_USER (1<< 2)

user mode page

Definition at line 63 of file x86.h.

4.30.1.15 #define X86_PTE_WRITE_THROUGH (1<< 3)

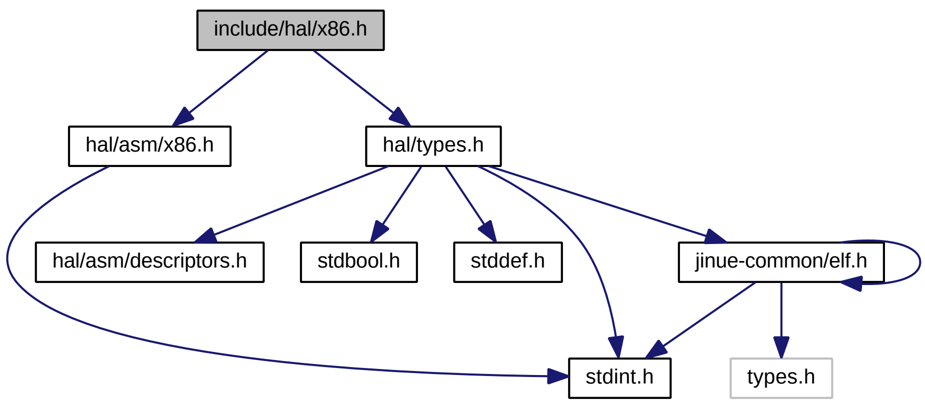
write-through cache policy for page

Definition at line 66 of file x86.h.

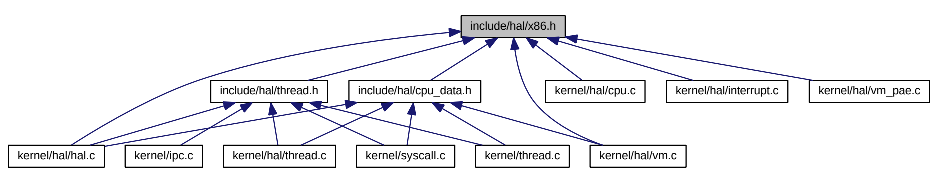
4.31 include/hal/x86.h File Reference

```
#include <hal/asm/x86.h>
#include <hal/types.h>
```

Include dependency graph for x86.h:



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



Data Structures

- struct **x86_cpuid_regs_t**

Typedefs

- typedef **uint32_t msr_addr_t**

Functions

- void **cli** (void)
- void **sti** (void)
- void **invalidate_tlb** (**addr_t** vaddr)
- void **lgdt** (**pseudo_descriptor_t** ∗gdt_info)

- void **lidt** (**pseudo_descriptor_t** ∗idt_info)
- void **ltr** (**seg_selector_t** sel)
- **uint32_t cpuid** (**x86_cpuid_regs_t** ∗regs)
- **uint32_t get_esp** (void)
- **uint32_t get_cr0** (void)
- **uint32_t get_cr2** (void)
- **uint32_t get_cr3** (void)
- **uint32_t get_cr4** (void)
- void **set_cr0** (**uint32_t** val)
- void **set_cr3** (**uint32_t** val)
- void **set_cr4** (**uint32_t** val)
- **uint32_t get_eflags** (void)
- void **set_eflags** (**uint32_t** val)
- void **set_cs** (**uint32_t** val)
- void **set_ds** (**uint32_t** val)
- void **set_es** (**uint32_t** val)
- void **set_fs** (**uint32_t** val)
- void **set_gs** (**uint32_t** val)
- void **set_ss** (**uint32_t** val)
- **uint64_t rdmsr** (**msr_addr_t** addr)
- void **wrmsr** (**msr_addr_t** addr, **uint64_t** val)
- **uint32_t get_gs_ptr** (**uint32_t** ∗ptr)
- **uint64_t rdtsc** (void)
- void **enable_pae** (**uint32_t** cr3_value)

4.31.1 Typedef Documentation

4.31.1.1 typedef uint32_t msr_addr_t

Definition at line 45 of file x86.h.

4.31.2 Function Documentation

4.31.2.1 void cli (void)

4.31.2.2 uint32_t cpuid (x86_cpuid_regs_t ∗ *regs*)

Referenced by cpu_detect_features().

4.31.2.3 void enable_pae (uint32_t *cr3_value*)

Referenced by vm_boot_init().

4.31.2.4 uint32_t get_cr0 (void)

4.31.2.5 uint32_t get_cr2 (void)

Referenced by dispatch_interrupt().

4.31.2.6 uint32_t get_cr3 (void)

4.31.2.7 uint32_t get_cr4 (void)

4.31.2.8 uint32_t get_eflags (void)

Referenced by cpu_detect_features().

4.31.2.9 uint32_t get_esp (void)

4.31.2.10 uint32_t get_gs_ptr (uint32_t * *ptr*)

4.31.2.11 void invalidate_tlb (addr_t *vaddr*)

Referenced by vm_change_flags().

4.31.2.12 void lgdt (pseudo_descriptor_t * *gdt_info*)

4.31.2.13 void lidt (pseudo_descriptor_t * *idt_info*)

4.31.2.14 void ltr (seg_selector_t *sel*)

4.31.2.15 uint64_t rdmsr (msr_addr_t *addr*)

4.31.2.16 uint64_t rdtsc (void)

4.31.2.17 void set_cr0 (uint32_t *val*)

4.31.2.18 void set_cr3 (uint32_t *val*)

Referenced by vm_switch_addr_space().

4.31.2.19 void set_cr4 (uint32_t *val*)

4.31.2.20 void set_cs (uint32_t *val*)

4.31.2.21 void set_ds (uint32_t *val*)

4.31.2.22 void set_eflags (uint32_t *val*)

Referenced by cpu_detect_features().

4.31.2.23 void set_es (uint32_t *val*)

4.31.2.24 void set_fs (uint32_t *val*)

4.31.2.25 void set_gs (uint32_t *val*)

4.31.2.26 void set_ss (uint32_t *val*)

4.31.2.27 void sti (void)

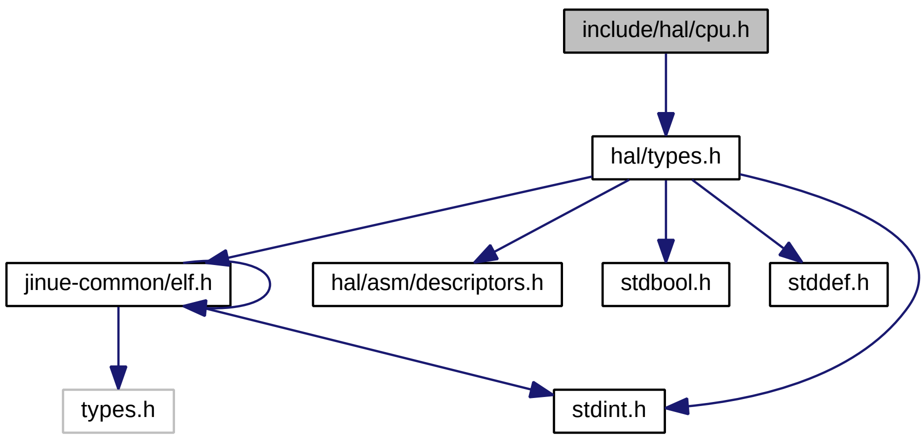
4.31.2.28 void wrmsr (msr_addr_t *addr*, uint64_t *val*)

Referenced by thread_context_switch().

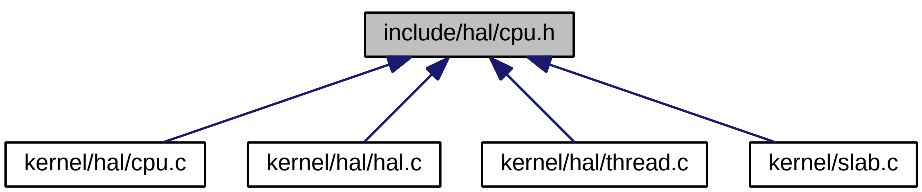
4.32 include/hal/cpu.h File Reference

`#include <hal/types.h>`

Include dependency graph for cpu.h:



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



Data Structures

- struct **cpu_info_t**

Macros

- #define **MSR_IA32_SYSENTER_CS** 0x174
- #define **MSR_IA32_SYSENTER_ESP** 0x175
- #define **MSR_IA32_SYSENTER_EIP** 0x176
- #define **MSR_EFER** 0xC0000080
- #define **MSR_STAR** 0xC0000081
- #define **MSR_FLAG_STAR_SCE** (1<<0)
- #define **CPU_FEATURE_CPUID** (1<<0)

- #define **CPU_FEATURE_SYSENTER** (1<<1)
- #define **CPU_FEATURE_SYSCALL** (1<<2)
- #define **CPU_FEATURE_LOCAL_APIC** (1<<3)
- #define **CPU_FEATURE_PAE** (1<<4)
- #define **CPU_EFLAGS_ID** (1<<21)
- #define **CPUID_FEATURE_FPU** (1<<0)
- #define **CPUID_FEATURE_PAE** (1<<6)
- #define **CPUID_FEATURE_APIC** (1<<9)
- #define **CPUID_FEATURE_SEP** (1<<11)
- #define **CPUID_FEATURE_CLFLUSH** (1<<19)
- #define **CPUID_FEATURE_HTT** (1<<28)
- #define **CPUID_EXT_FEATURE_SYSCALL** (1<<11)
- #define **CPU_VENDOR_GENERIC** 0
- #define **CPU_VENDOR_AMD** 1
- #define **CPU_VENDOR_INTEL** 2
- #define **CPU_VENDOR_AMD_DW0** 0x68747541 /* Auth */
- #define **CPU_VENDOR_AMD_DW1** 0x69746e65 /* enti */
- #define **CPU_VENDOR_AMD_DW2** 0x444d4163 /* cAMD */
- #define **CPU_VENDOR_INTEL_DW0** 0x756e6547 /* Genu */
- #define **CPU_VENDOR_INTEL_DW1** 0x49656e69 /* ineI */
- #define **CPU_VENDOR_INTEL_DW2** 0x6c65746e /* ntel */

Functions

- void **cpu_init_data** (**cpu_data_t** *data)
- void **cpu_detect_features** (void)

Variables

- **cpu_info_t cpu_info**

4.32.1 Macro Definition Documentation

4.32.1.1 #define CPU_EFLAGS_ID (1<<21)

Definition at line 63 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.2 #define CPU_FEATURE_CPUID (1<<0)

Definition at line 52 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.3 #define CPU_FEATURE_LOCAL_APIC (1<<3)

Definition at line 58 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.4 #define CPU_FEATURE_PAE (1<<4)

Definition at line 60 of file cpu.h.

Referenced by cpu_detect_features(), and hal_init().

4.32.1.5 #define CPU_FEATURE_SYSCALL (1<<2)

Definition at line 56 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.6 #define CPU_FEATURE_SYSENTER (1<<1)

Definition at line 54 of file cpu.h.

Referenced by cpu_detect_features(), and thread_context_switch().

4.32.1.7 #define CPU_VENDOR_AMD 1

Definition at line 84 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.8 #define CPU_VENDOR_AMD_DW0 0x68747541 /* Auth */

Definition at line 89 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.9 #define CPU_VENDOR_AMD_DW1 0x69746e65 /* enti */

Definition at line 90 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.10 #define CPU_VENDOR_AMD_DW2 0x444d4163 /* cAMD */

Definition at line 91 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.11 #define CPU_VENDOR_GENERIC 0

Definition at line 82 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.12 #define CPU_VENDOR_INTEL 2

Definition at line 86 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.13 #define CPU_VENDOR_INTEL_DW0 0x756e6547 /* Genu */

Definition at line 93 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.14 #define CPU_VENDOR_INTEL_DW1 0x49656e69 /* inel */

Definition at line 94 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.15 #define CPU_VENDOR_INTEL_DW2 0x6c65746e /* ntel */

Definition at line 95 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.16 #define CPUID_EXT_FEATURE_SYSCALL (1<<11)

Definition at line 79 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.17 #define CPUID_FEATURE_APIC (1<<9)

Definition at line 70 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.18 #define CPUID_FEATURE_CLFLUSH (1<<19)

Definition at line 74 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.19 #define CPUID_FEATURE_FPU (1<<0)

Definition at line 66 of file cpu.h.

4.32.1.20 #define CPUID_FEATURE_HTT (1<<28)

Definition at line 76 of file cpu.h.

4.32.1.21 #define CPUID_FEATURE_PAE (1<<6)

Definition at line 68 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.22 #define CPUID_FEATURE_SEP (1<<11)

Definition at line 72 of file cpu.h.

Referenced by cpu_detect_features().

4.32.1.23 #define MSR_EFER 0xC0000080

Definition at line 44 of file cpu.h.

4.32.1.24 #define MSR_FLAG_STAR_SCE (1<<0)

Definition at line 49 of file cpu.h.

4.32.1.25 #define MSR_IA32_SYSENTER_CS 0x174

Definition at line 38 of file cpu.h.

4.32.1.26 #define MSR_IA32_SYSENTER_EIP 0x176

Definition at line 42 of file cpu.h.

4.32.1.27 #define MSR_IA32_SYSENTER_ESP 0x175

Definition at line 40 of file cpu.h.

Referenced by thread_context_switch().

4.32.1.28 #define MSR_STAR 0xC0000081

Definition at line 46 of file cpu.h.

4.32.2 Function Documentation

4.32.2.1 void cpu_detect_features (void)

Definition at line 87 of file cpu.c.

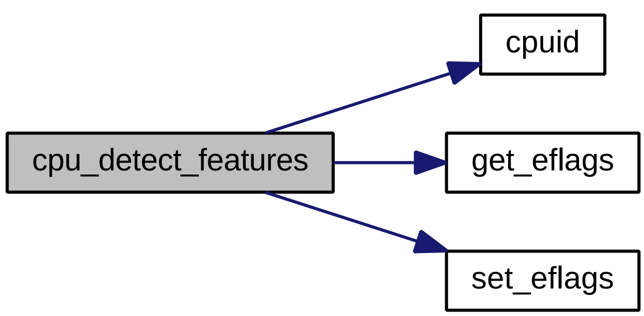
References CPU_EFLAGS_ID, CPU_FEATURE_CPUID, CPU_FEATURE_LOCAL_APIC, CPU_FEATURE_PAE, CP-U_FEATURE_SYSCALL, CPU_FEATURE_SYSENTER, CPU_VENDOR_AMD, CPU_VENDOR_AMD_DW0, CPU_V-ENDOR_AMD_DW1, CPU_VENDOR_AMD_DW2, CPU_VENDOR_GENERIC, CPU_VENDOR_INTEL, CPU_VEND-OR_INTEL_DW0, CPU_VENDOR_INTEL_DW1, CPU_VENDOR_INTEL_DW2, cpuid(), CPUID_EXT_FEATURE_S-YSCALL, CPUID_FEATURE_APIC, CPUID_FEATURE_CLFLUSH, CPUID_FEATURE_PAE, CPUID_FEATURE_SE-P, cpu_info_t::dcache_alignment, x86_cpuid_regs_t::eax, x86_cpuid_regs_t::ebx, x86_cpuid_regs_t::ecx, x86_cpuid_-regs_t::edx, cpu_info_t::family, cpu_info_t::features, get_eflags(), cpu_info_t::model, set_eflags(), cpu_info_t::stepping, and cpu_info_t::vendor.

Referenced by hal_init().

```
                               {
    uint32_t temp_eflags;

    /* default values */
    cpu_info.dcache_alignment   = 32;
    cpu_info.features           = 0;
    cpu_info.vendor             = CPU_VENDOR_GENERIC;
    cpu_info.family             = 0;
    cpu_info.model              = 0;
    cpu_info.stepping           = 0;

    /* The CPUID instruction is available if we can change the value of eflags
     * bit 21 (ID) */
     temp_eflags  = get_eflags();
     temp_eflags ^= CPU_EFLAGS_ID;
     set_eflags(temp_eflags);

     if(temp_eflags == get_eflags()) {
         cpu_info.features |= CPU_FEATURE_CPUID;
     }

     if(cpu_has_feature(CPU_FEATURE_CPUID)) {
         uint32_t            signature;
         uint32_t            flags, ext_flags;
         uint32_t            vendor_dw0, vendor_dw1, vendor_dw2;
         uint32_t            cpuid_max;
         uint32_t            cpuid_ext_max;
         x86_cpuid_regs_t    regs;

         /* default values */
         flags               = 0;
         ext_flags           = 0;

         /* function 0: vendor ID string, max value of eax when calling CPUID */
         regs.eax = 0;

         /* call CPUID instruction */
         cpuid_max  = cpuid(&regs);
         vendor_dw0 = regs.ebx;
         vendor_dw1 = regs.edx;
         vendor_dw2 = regs.ecx;

         /* identify vendor */
         if(    vendor_dw0 == CPU_VENDOR_AMD_DW0
             && vendor_dw1 == CPU_VENDOR_AMD_DW1
             && vendor_dw2 == CPU_VENDOR_AMD_DW2) {

             cpu_info.vendor = CPU_VENDOR_AMD;
         }
         else if (vendor_dw0 == CPU_VENDOR_INTEL_DW0
             &&   vendor_dw1 == CPU_VENDOR_INTEL_DW1
             &&   vendor_dw2 == CPU_VENDOR_INTEL_DW2) {

             cpu_info.vendor = CPU_VENDOR_INTEL;
         }

         /* get processor signature (family/model/stepping) and feature flags */
         if(cpuid_max >= 1) {
             /* function 1: processor signature and feature flags */
             regs.eax = 1;

             /* call CPUID instruction */
             signature = cpuid(&regs);

             /* set processor signature */
             cpu_info.stepping  = signature       & 0xf;
             cpu_info.model     = (signature>>4)  & 0xf;
             cpu_info.family    = (signature>>8)  & 0xf;

             /* feature flags */
             flags = regs.edx;

             /* cache alignment */
             if(flags & CPUID_FEATURE_CLFLUSH) {
                 cpu_info.dcache_alignment = ((regs.ebx >> 8) & 0xff) * 8;
             }
         }

         /* extended function 0: max value of eax when calling CPUID (extended function) */
         regs.eax = 0x80000000;
         cpuid_ext_max = cpuid(&regs);
```

```

        /* get extended feature flags */
        if(cpuid_ext_max >= 0x80000001) {
            /* extended function 1: extended feature flags */
            regs.eax = 0x80000001;
            (void)cpuid(&regs);

            /* extended feature flags */
            ext_flags = regs.edx;
        }

        /* support for SYSENTER/SYSEXIT instructions */
        if(flags & CPUID_FEATURE_SEP) {
            if(cpu_info.vendor == CPU_VENDOR_AMD) {
                cpu_info.features |= CPU_FEATURE_SYSENTER;
            }
            else if(cpu_info.vendor == CPU_VENDOR_INTEL) {
                if(cpu_info.family == 6 && cpu_info.model < 3 && cpu_info.stepping < 3) {
                    /* not supported */
                }
                else {
                    cpu_info.features |= CPU_FEATURE_SYSENTER;
                }
            }
        }

        /* support for SYSCALL/SYSRET instructions */
        if(cpu_info.vendor == CPU_VENDOR_AMD) {
            if(ext_flags & CPUID_EXT_FEATURE_SYSCALL) {
                cpu_info.features |= CPU_FEATURE_SYSCALL;
            }
        }

        /* support for local APIC */
        if(cpu_info.vendor == CPU_VENDOR_AMD || cpu_info.vendor == CPU_VENDOR_INTEL) {
            if(flags & CPUID_FEATURE_APIC) {
                cpu_info.features |= CPU_FEATURE_LOCAL_APIC;
            }
        }

        /* support for physical address extension (PAE) */
        if(cpu_info.vendor == CPU_VENDOR_AMD || cpu_info.vendor == CPU_VENDOR_INTEL) {
            if(flags & CPUID_FEATURE_PAE) {
                cpu_info.features |= CPU_FEATURE_PAE;
            }
        }
    }
}
```

Here is the call graph for this function:



4.32.2.2 void cpu_init_data (cpu_data_t * *data*)

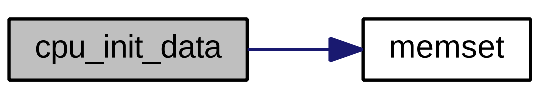
Definition at line 42 of file cpu.c.

References cpu_data_t::current_addr_space, tss_t::esp0, tss_t::esp1, tss_t::esp2, cpu_data_t::gdt, GDT_KERNEL_-CODE, GDT_KERNEL_DATA, GDT_NULL, GDT_PER_CPU_DATA, GDT_TSS, GDT_USER_CODE, GDT_USER_-DATA, GDT_USER_TLS_DATA, memset(), NULL, RPL_KERNEL, SEG_DESCRIPTOR, SEG_FLAG_32BIT, SEG_-FLAG_IN_BYTES, SEG_FLAG_KERNEL, SEG_FLAG_NORMAL, SEG_FLAG_NOSYSTEM, SEG_FLAG_PRESENT, SEG_FLAG_TSS, SEG_FLAG_USER, SEG_SELECTOR, SEG_TYPE_CODE, SEG_TYPE_DATA, SEG_TYPE_TSS, cpu_data_t::self, tss_t::ss0, tss_t::ss1, tss_t::ss2, cpu_data_t::tss, and TSS_LIMIT.

Referenced by hal_init().

```
                                    {
    tss_t *tss;

    tss = &data->tss;

    /* initialize with zeroes  */
    memset(data, '\0', sizeof(cpu_data_t));

    data->self                     = data;
    data->current_addr_space       = NULL;

    /* initialize GDT */
    data->gdt[GDT_NULL] = SEG_DESCRIPTOR(0, 0, 0);

    data->gdt[GDT_KERNEL_CODE] =
        SEG_DESCRIPTOR( 0,      0xfffff,                SEG_TYPE_CODE  |
   SEG_FLAG_KERNEL | SEG_FLAG_NORMAL);

    data->gdt[GDT_KERNEL_DATA] =
        SEG_DESCRIPTOR( 0,      0xfffff,                SEG_TYPE_DATA  |
   SEG_FLAG_KERNEL | SEG_FLAG_NORMAL);

    data->gdt[GDT_USER_CODE] =
        SEG_DESCRIPTOR( 0,      0xfffff,                SEG_TYPE_CODE  |
   SEG_FLAG_USER   | SEG_FLAG_NORMAL);

    data->gdt[GDT_USER_DATA] =
        SEG_DESCRIPTOR( 0,      0xfffff,                SEG_TYPE_DATA  |
   SEG_FLAG_USER   | SEG_FLAG_NORMAL);

    data->gdt[GDT_TSS] =
        SEG_DESCRIPTOR( tss,    TSS_LIMIT-1,            SEG_TYPE_TSS   |
   SEG_FLAG_KERNEL | SEG_FLAG_TSS);

    data->gdt[GDT_PER_CPU_DATA] =
        SEG_DESCRIPTOR( data,   sizeof(cpu_data_t)-1,   SEG_TYPE_DATA  |
   SEG_FLAG_KERNEL | SEG_FLAG_32BIT | SEG_FLAG_IN_BYTES | SEG_FLAG_NOSYSTEM |
   SEG_FLAG_PRESENT);

    data->gdt[GDT_USER_TLS_DATA] = SEG_DESCRIPTOR(0, 0, 0);

    /* setup kernel stack in TSS */
    tss->ss0  = SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL);
    tss->ss1  = SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL);
    tss->ss2  = SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL);

    /* kernel stack address is updated by thread_context_switch() */
    tss->esp0 = NULL;
    tss->esp1 = NULL;
    tss->esp2 = NULL;
}
```

Here is the call graph for this function:



4.32.3 Variable Documentation

4.32.3.1 cpu_info_t cpu_info

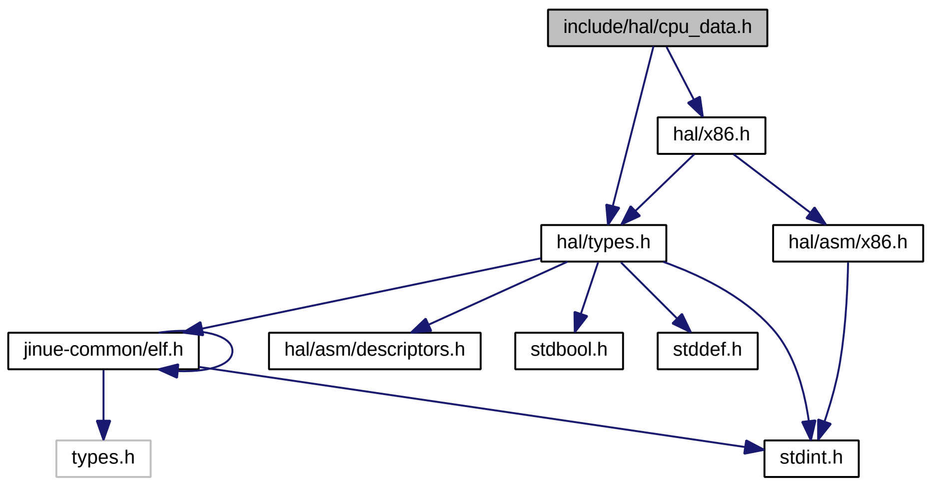
Definition at line 39 of file cpu.c.

Referenced by slab_cache_init().

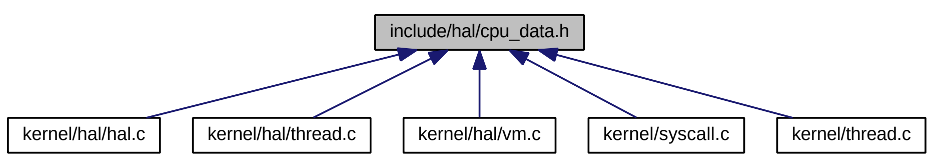
4.33 include/hal/cpu_data.h File Reference

```
#include <hal/types.h>
#include <hal/x86.h>
```

Include dependency graph for cpu_data.h:



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



Macros

- #define **CPU_DATA_ALIGNMENT** 256

4.33.1 Macro Definition Documentation

4.33.1.1 #define CPU_DATA_ALIGNMENT 256

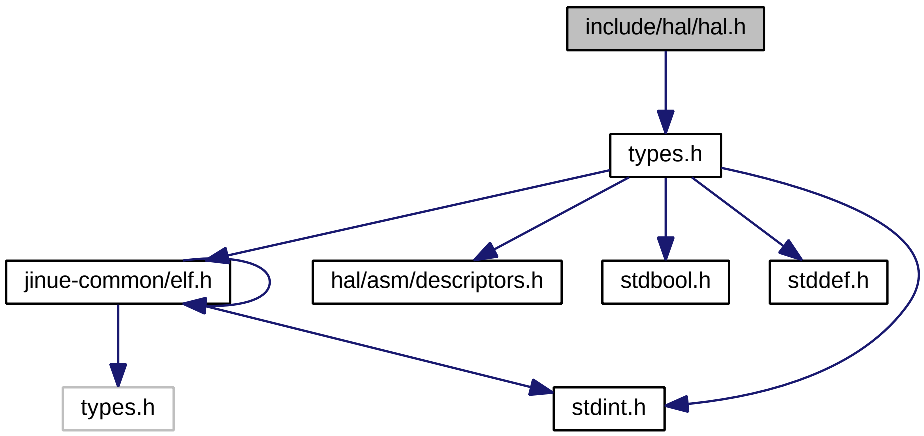
Definition at line 39 of file cpu_data.h.

Referenced by hal_init().

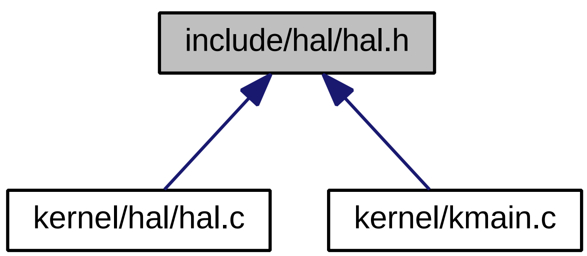
4.34 include/hal/hal.h File Reference

`#include <types.h>`

Include dependency graph for hal.h:



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



Functions

- void **hal_init** (**boot_alloc_t** ∗boot_alloc, const **boot_info_t** ∗**boot_info**)

4.34.1 Function Documentation

4.34.1.1 void hal_init (**boot_alloc_t** ∗ *boot_alloc,* const **boot_info_t** ∗ *boot_info*)

Definition at line 156 of file hal.c.

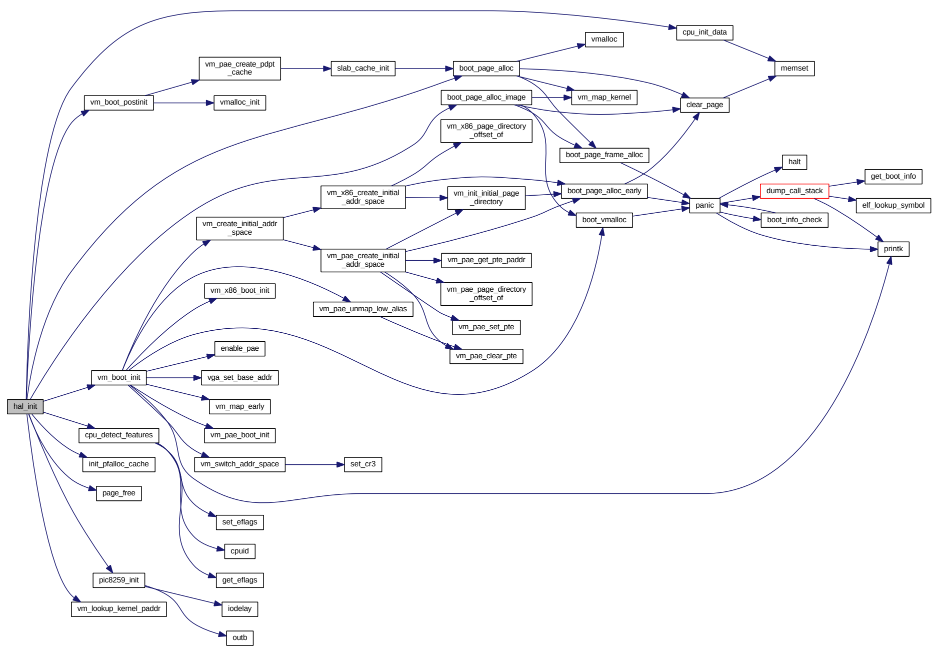
References assert, boot_heap_alloc, boot_page_alloc(), boot_page_alloc_image(), CPU_DATA_ALIGNMENT, cpu_detect_features(), CPU_FEATURE_PAE, cpu_init_data(), global_pfalloc_cache, IDT_PIC8259_BASE, init_pfalloc_cache(), KERNEL_PAGE_STACK_INIT, page_free(), pffree, pic8259_init(), vm_boot_init(), vm_boot_postinit(), and vm_lookup_kernel_paddr().

Referenced by kmain().

```
156                                                                   {
157     int idx;
158
```

```
    /* get cpu info */
    cpu_detect_features();

    /* allocate per-CPU data
     *
     * We need to ensure that the Task State Segment (TSS) contained in this
     * memory block does not cross a page boundary. */
    assert(sizeof(cpu_data_t) < CPU_DATA_ALIGNMENT);

    cpu_data_t *cpu_data = boot_heap_alloc(boot_alloc, cpu_data_t,
  CPU_DATA_ALIGNMENT);

    /* initialize per-CPU data */
    cpu_init_data(cpu_data);

    /* Initialize interrupt descriptor table (IDT)
     *
     * This function modifies the IDT in-place (see trap.asm). This must be
     * done before vm_boot_init() because the page protection bits set up by
     * vm_boot_init() prevent this. */
    hal_init_idt();

    /* Initialize programmable interrupt_controller. */
    pic8259_init(IDT_PIC8259_BASE);

    /* initialize virtual memory management, enable paging
     *
     * below this point, it is no longer safe to call pfalloc_early() */
    bool use_pae = cpu_has_feature(CPU_FEATURE_PAE);
    vm_boot_init(boot_info, use_pae, cpu_data, boot_alloc);

    /* Initialize GDT and TSS */
    hal_init_descriptors(cpu_data, boot_alloc);

    /* initialize the page frame allocator */
    kern_paddr_t *page_stack_buffer = (kern_paddr_t *)boot_page_alloc_image(boot_alloc);
    init_pfalloc_cache(&global_pfalloc_cache, page_stack_buffer);

    /* TODO Remove this once vm.c rework is done. */
    for(idx = 0; idx < KERNEL_PAGE_STACK_INIT; ++idx) {
        pffree(
                vm_lookup_kernel_paddr(
                        boot_page_alloc_image(boot_alloc)));
    }

    /* Initialize virtual memory allocator and VM management caches. */
    vm_boot_postinit(boot_info, boot_alloc, use_pae);

    /* TODO Remove this once add page frame system call is implemented.
     *
     * The test user space program needs one page to create a new thread. */
    page_free(boot_page_alloc(boot_alloc));

    /* choose system call method */
    hal_select_syscall_method();
}
```

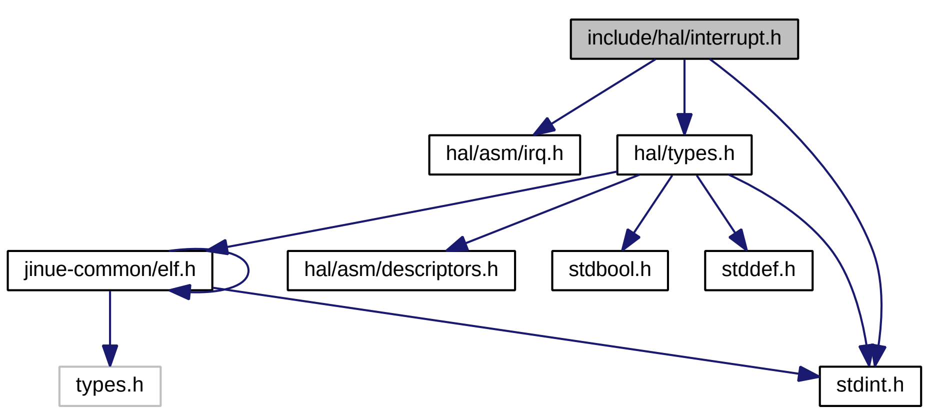
Here is the call graph for this function:



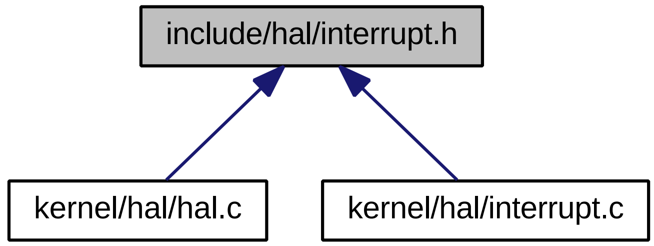
4.35 include/hal/interrupt.h File Reference

```
#include <hal/asm/irq.h>
#include <hal/types.h>
#include <stdint.h>
```

Include dependency graph for interrupt.h:



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



Functions

- void **dispatch_interrupt** (**trapframe_t** ∗trapframe)

Variables

- **seg_descriptor_t idt** []

4.35.1 Function Documentation

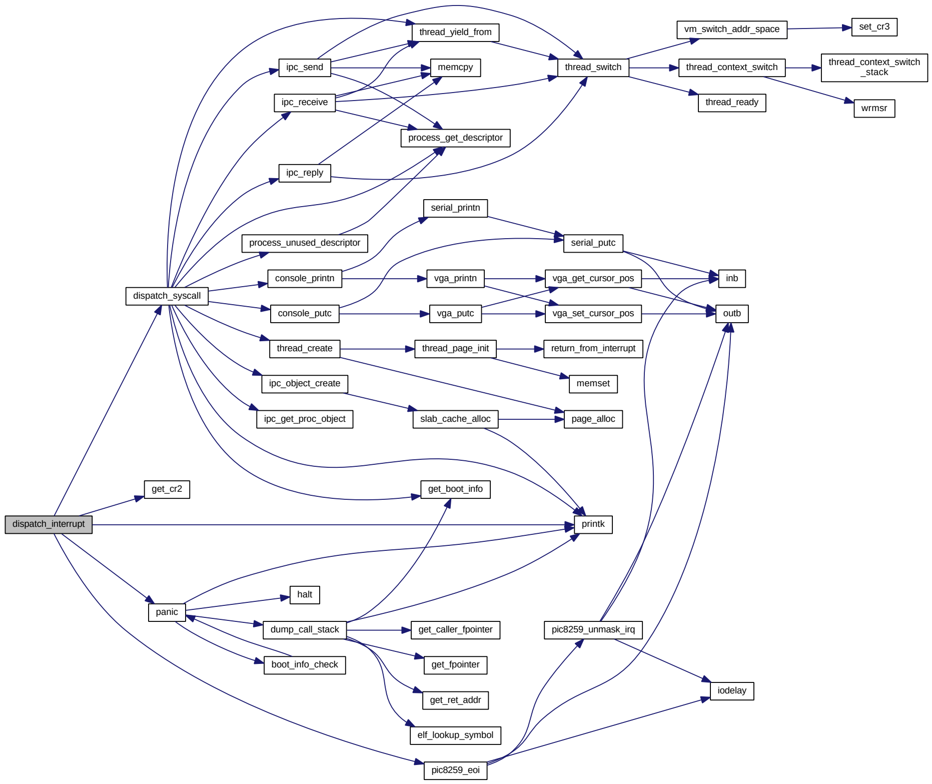
4.35.1.1 void dispatch_interrupt (trapframe_t ∗ *trapframe*)

Definition at line 42 of file interrupt.c.

References dispatch_syscall(), trapframe_t::eip, trapframe_t::errcode, get_cr2(), IDT_LAST_EXCEPTION, IDT_PIC8259_BASE, trapframe_t::ivt, panic(), pic8259_eoi(), PIC8259_IRQ_COUNT, printk(), and SYSCALL_IRQ.

```
42                                                    {
43     unsigned int    ivt         = trapframe->ivt;
44     uintptr_t       eip         = trapframe->eip;
45     uint32_t        errcode     = trapframe->errcode;
46 
47     /* exceptions */
48     if(ivt <= IDT_LAST_EXCEPTION) {
49         printk("EXCEPT: %u cr2=0x%x errcode=0x%x eip=0x%x\n", ivt, get_cr2(), errcode, eip);
50 
51         /* never returns */
52         panic("caught exception");
53     }
54 
55     if(ivt == SYSCALL_IRQ) {
56         /* slow system call method */
57         dispatch_syscall(trapframe);
58     }
59     else if(ivt >= IDT_PIC8259_BASE && ivt < IDT_PIC8259_BASE +
      PIC8259_IRQ_COUNT) {
60         int irq = ivt - IDT_PIC8259_BASE;
61         printk("IRQ: %u (vector %u)\n", irq, ivt);
62         pic8259_eoi(irq);
63     }
64     else {
65         printk("INTR: vector %u\n", ivt);
66     }
67 }
```

Here is the call graph for this function:



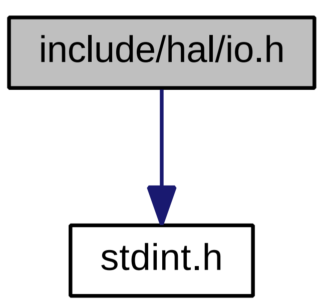
4.35.2 Variable Documentation

4.35.2.1 seg_descriptor_t idt[ ]

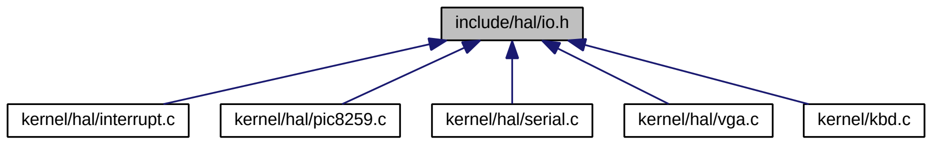
4.36 include/hal/io.h File Reference

`#include <stdint.h>`

Include dependency graph for io.h:



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



Functions

- **uint8_t inb** (**uint16_t** port)
- **uint16_t inw** (**uint16_t** port)
- **uint32_t inl** (**uint16_t** port)
- void **outb** (**uint16_t** port, **uint8_t** value)
- void **outw** (**uint16_t** port, **uint16_t** value)
- void **outl** (**uint16_t** port, **uint32_t** value)
- void **iodelay** (void)

4.36.1 Function Documentation

4.36.1.1 uint8_t inb (uint16_t *port*)

Referenced by any_key(), pic8259_mask_irq(), pic8259_unmask_irq(), serial_putc(), vga_get_cursor_pos(), and vga_-init().

4.36.1.2 uint32_t inl (uint16_t *port*)

4.36.1.3 uint16_t inw (uint16_t *port*)

4.36.1.4 void iodelay (void)

Referenced by pic8259_eoi(), pic8259_init(), pic8259_mask_irq(), and pic8259_unmask_irq().

4.36.1.5 void outb (uint16_t *port,* uint8_t *value*)

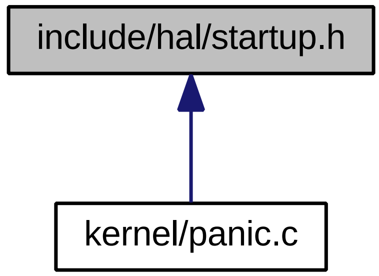
Referenced by pic8259_eoi(), pic8259_init(), pic8259_mask_irq(), pic8259_unmask_irq(), serial_init(), serial_putc(), vga_get_cursor_pos(), vga_init(), and vga_set_cursor_pos().

4.36.1.6 void outl (uint16_t *port,* uint32_t *value*)

4.36.1.7 void outw (uint16_t *port,* uint16_t *value*)

4.37 include/hal/startup.h File Reference

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



Functions

- void **halt** (void)

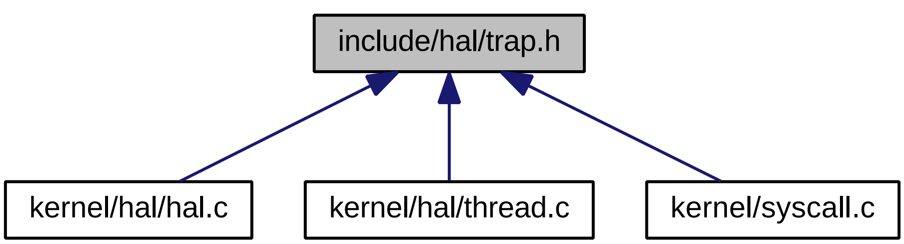
4.37.1 Function Documentation

4.37.1.1 void halt (void)

Referenced by panic().

4.38 include/hal/trap.h File Reference

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



Functions

- void **fast_intel_entry** (void)

  entry point for Intel fast system call mechanism (SYSENTER/SYSEXIT)
- void **fast_amd_entry** (void)

  entry point for AMD fast system call mechanism (SYSCALL/SYSRET)
- void **return_from_interrupt** (void)

Variables

- int **syscall_method**
  Specifies the entry point to use for system calls.

4.38.1 Function Documentation

4.38.1.1 void fast_amd_entry (void)

entry point for AMD fast system call mechanism (SYSCALL/SYSRET)

4.38.1.2 void fast_intel_entry (void)

entry point for Intel fast system call mechanism (SYSENTER/SYSEXIT)

4.38.1.3 void return_from_interrupt (void)

Referenced by thread_page_init().

4.38.2 Variable Documentation

4.38.2.1 int syscall_method

Specifies the entry point to use for system calls.

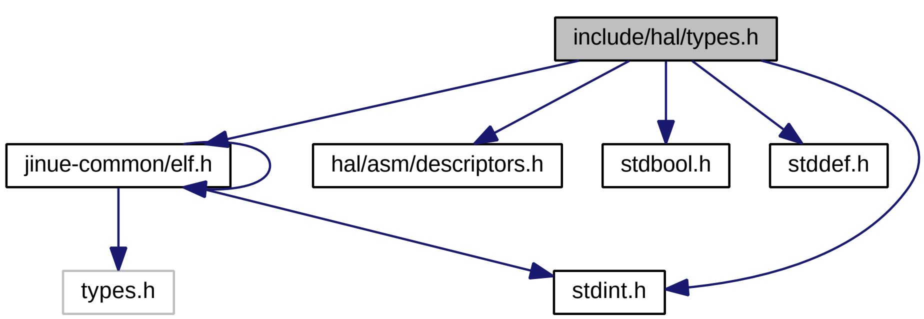
Definition at line 58 of file hal.c.

Referenced by dispatch_syscall().

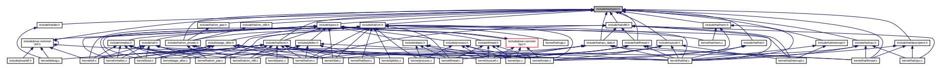
4.39 include/hal/types.h File Reference

```
#include <jinue-common/elf.h>
#include <hal/asm/descriptors.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
```

Include dependency graph for types.h:



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



Data Structures

- struct **thread_context_t**
- struct **addr_space_t**
- struct **e820_t**
- struct **boot_info_t**
- struct **pseudo_descriptor_t**
- struct **tss_t**
- struct **cpu_data_t**
- struct **trapframe_t**
- struct **kernel_context_t**

Macros

- #define **PFNULL** ((**kern_paddr_t**)-1)

  an invalid page frame address used as null value
- #define **msg_arg0** eax
- #define **msg_arg1** ebx
- #define **msg_arg2** esi
- #define **msg_arg3** edi

Typedefs

- typedef unsigned char ∗ **addr_t**

  Virtual memory address (pointer) with pointer arithmetic allowed.
- typedef **uint32_t kern_paddr_t**

  Physical memory address for use by the kernel.
- typedef **uint64_t user_paddr_t**

  Physical memory address for use by user space.
- typedef struct **pte_t pte_t**

  type of a page table entry
- typedef struct **pdpt_t pdpt_t**
- typedef **uint64_t seg_descriptor_t**
- typedef **uint32_t seg_selector_t**
- typedef struct **cpu_data_t cpu_data_t**

4.39.1 Macro Definition Documentation

4.39.1.1 #define msg_arg0 eax

Definition at line 195 of file types.h.

4.39.1.2 #define msg_arg1 ebx

Definition at line 197 of file types.h.

4.39.1.3 #define msg_arg2 esi

Definition at line 199 of file types.h.

4.39.1.4 #define msg_arg3 edi

Definition at line 201 of file types.h.

4.39.1.5 #define PFNULL ((kern_paddr_t)-1)

an invalid page frame address used as null value

Definition at line 51 of file types.h.

Referenced by init_pfalloc_cache(), and remove_page_frame().

4.39.2 Typedef Documentation

4.39.2.1 typedef unsigned char* addr_t

Virtual memory address (pointer) with pointer arithmetic allowed.

Definition at line 42 of file types.h.

4.39.2.2 typedef struct cpu_data_t cpu_data_t

Definition at line 191 of file types.h.

4.39.2.3 typedef uint32_t kern_paddr_t

Physical memory address for use by the kernel.

Definition at line 45 of file types.h.

4.39.2.4 typedef struct pdpt_t pdpt_t

Definition at line 70 of file types.h.

4.39.2.5 typedef struct pte_t pte_t

type of a page table entry

Definition at line 66 of file types.h.

4.39.2.6 typedef uint64_t seg_descriptor_t

Definition at line 115 of file types.h.

4.39.2.7 typedef uint32_t seg_selector_t

Definition at line 117 of file types.h.

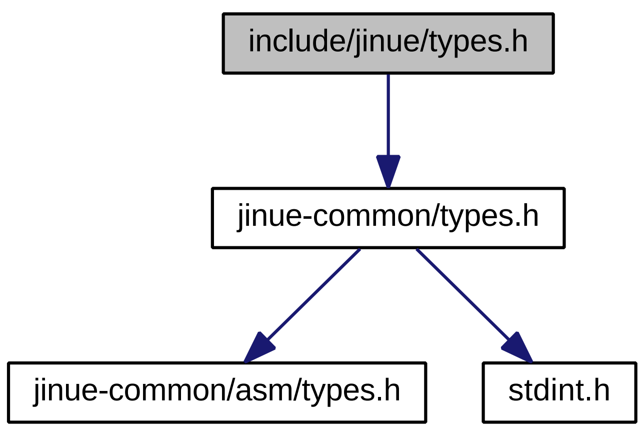
4.39.2.8 typedef uint64_t user_paddr_t

Physical memory address for use by user space.

Definition at line 48 of file types.h.

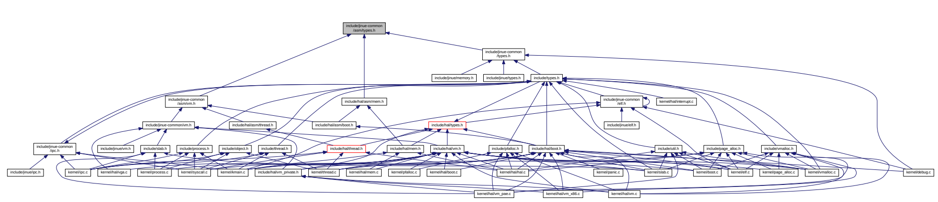
4.40 include/jinue/types.h File Reference

`#include <jinue-common/types.h>`
Include dependency graph for types.h:



4.41 include/jinue-common/asm/types.h File Reference

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



Macros

- #define **KB** (1024)

- #define **MB** (1024 ∗ **KB**)
- #define **GB** (1024 ∗ **MB**)

4.41.1 Macro Definition Documentation

4.41.1.1 #define GB (1024 ∗ MB)

Definition at line 39 of file types.h.

Referenced by mem_check_memory().

4.41.1.2 #define KB (1024)

Definition at line 35 of file types.h.

4.41.1.3 #define MB (1024 ∗ KB)

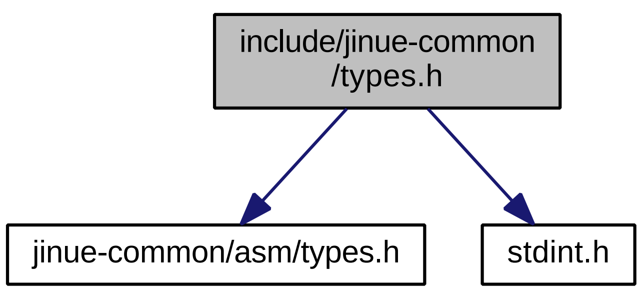
Definition at line 37 of file types.h.

Referenced by vm_boot_postinit().

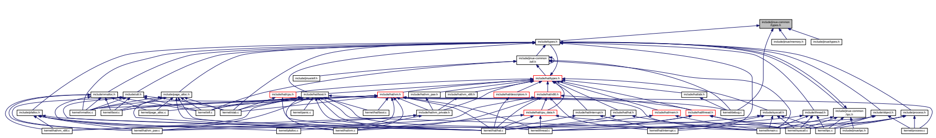
4.42 include/jinue-common/types.h File Reference

```
#include <jinue-common/asm/types.h>
#include <stdint.h>
```

Include dependency graph for types.h:



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



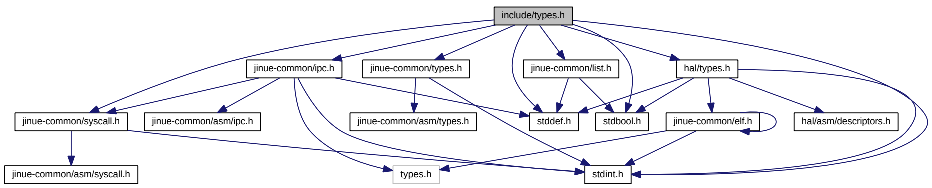
Data Structures

- struct **jinue_mem_entry_t**
- struct **jinue_mem_map_t**

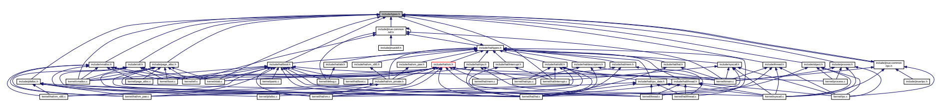
4.43 include/types.h File Reference

```
#include <jinue-common/ipc.h>
#include <jinue-common/list.h>
#include <jinue-common/syscall.h>
#include <jinue-common/types.h>
#include <hal/types.h>
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
```

Include dependency graph for types.h:



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



Data Structures

- struct **boot_heap_pushed_state**
- struct **boot_alloc_t**
- struct **object_header_t**
- struct **object_ref_t**
- struct **process_t**
- struct **message_info_t**
- struct **thread_t**
- struct **ipc_t**

Macros

- #define **PROCESS_MAX_DESCRIPTORS** 12

Typedefs

- typedef struct **thread_t thread_t**

4.43.1 Macro Definition Documentation

4.43.1.1 #define PROCESS_MAX_DESCRIPTORS 12

Definition at line 70 of file types.h.

Referenced by process_get_descriptor(), and process_unused_descriptor().

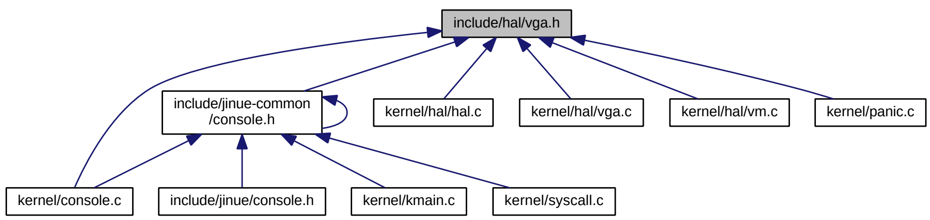
4.43.2 Typedef Documentation

4.43.2.1 typedef struct thread_t thread_t

Definition at line 98 of file types.h.

4.44 include/hal/vga.h File Reference

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



Macros

- #define **VGA_TEXT_VID_BASE** 0xb8000
- #define **VGA_TEXT_VID_TOP** 0xc0000
- #define **VGA_TEXT_VID_SIZE** (**VGA_TEXT_VID_TOP** - **VGA_TEXT_VID_BASE**)
- #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_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_set_base_addr** (void ∗base_addr)
- void **vga_clear** (void)
- void **vga_print** (const char ∗message, int colour)
- void **vga_printn** (const char ∗message, unsigned int n, int colour)
- void **vga_putc** (char c, int colour)
- void **vga_scroll** (void)
- **vga_pos_t vga_get_cursor_pos** (void)
- void **vga_set_cursor_pos** (**vga_pos_t** pos)

4.44.1 Macro Definition Documentation

4.44.1.1 #define VGA_COL(*x*) ((x) % (VGA_WIDTH))

Definition at line 68 of file vga.h.

4.44.1.2 #define VGA_COLOR_BLACK 0x00

Definition at line 45 of file vga.h.

4.44.1.3 #define VGA_COLOR_BLUE 0x01

Definition at line 46 of file vga.h.

4.44.1.4 #define VGA_COLOR_BRIGHTBLUE 0x09

Definition at line 54 of file vga.h.

4.44.1.5 #define VGA_COLOR_BRIGHTCYAN 0x0b

Definition at line 56 of file vga.h.

4.44.1.6 #define VGA_COLOR_BRIGHTGREEN 0x0a

Definition at line 55 of file vga.h.

4.44.1.7 #define VGA_COLOR_BRIGHTMAGENTA 0x0d

Definition at line 58 of file vga.h.

4.44.1.8 #define VGA_COLOR_BRIGHTRED 0x0c

Definition at line 57 of file vga.h.

4.44.1.9 #define VGA_COLOR_BRIGHTWHITE 0x0f

Definition at line 60 of file vga.h.

4.44.1.10 #define VGA_COLOR_BROWN 0x06

Definition at line 51 of file vga.h.

4.44.1.11 #define VGA_COLOR_CYAN 0x03

Definition at line 48 of file vga.h.

4.44.1.12 #define VGA_COLOR_ERASE VGA_COLOR_RED

Definition at line 61 of file vga.h.

Referenced by vga_clear(), and vga_scroll().

4.44.1.13 #define VGA_COLOR_GRAY 0x08

Definition at line 53 of file vga.h.

4.44.1.14 #define VGA_COLOR_GREEN 0x02

Definition at line 47 of file vga.h.

4.44.1.15 #define VGA_COLOR_MAGENTA 0x05

Definition at line 50 of file vga.h.

4.44.1.16 #define VGA_COLOR_RED 0x04

Definition at line 49 of file vga.h.

Referenced by panic().

4.44.1.17 #define VGA_COLOR_WHITE 0x07

Definition at line 52 of file vga.h.

4.44.1.18 #define VGA_COLOR_YELLOW 0x0e

Definition at line 59 of file vga.h.

Referenced by kmain().

4.44.1.19 #define VGA_CRTC_ADDR 0x3d4

Definition at line 40 of file vga.h.

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

4.44.1.20 #define VGA_CRTC_DATA 0x3d5

Definition at line 41 of file vga.h.

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

4.44.1.21 #define VGA_FB_FLAG_ACTIVE 1

Definition at line 43 of file vga.h.

4.44.1.22 #define VGA_LINE(*x*) ((x) / (VGA_WIDTH))

Definition at line 67 of file vga.h.

4.44.1.23 #define VGA_LINES 25

Definition at line 63 of file vga.h.

Referenced by vga_clear(), and vga_scroll().

4.44.1.24 #define VGA_MISC_OUT_RD 0x3cc

Definition at line 39 of file vga.h.

Referenced by vga_init().

4.44.1.25 #define VGA_MISC_OUT_WR 0x3c2

Definition at line 38 of file vga.h.

Referenced by vga_init().

4.44.1.26 #define VGA_TAB_WIDTH 8

Definition at line 65 of file vga.h.

4.44.1.27 #define VGA_TEXT_VID_BASE 0xb8000

Definition at line 35 of file vga.h.

Referenced by vm_boot_init().

4.44.1.28 #define VGA_TEXT_VID_SIZE (VGA_TEXT_VID_TOP - VGA_TEXT_VID_BASE)

Definition at line 37 of file vga.h.

4.44.1.29 #define VGA_TEXT_VID_TOP 0xc0000

Definition at line 36 of file vga.h.

Referenced by vm_boot_init().

4.44.1.30 #define VGA_WIDTH 80

Definition at line 64 of file vga.h.

Referenced by vga_clear(), and vga_scroll().

4.44.2 Typedef Documentation

4.44.2.1 typedef unsigned int vga_pos_t

Definition at line 71 of file vga.h.

4.44.3 Function Documentation

4.44.3.1 void vga_clear (void)

Definition at line 65 of file vga.c.

References VGA_COLOR_ERASE, VGA_LINES, and VGA_WIDTH.

Referenced by vga_init().

```
65                      {
66     unsigned int idx = 0;
67 
68     while( idx < (VGA_LINES * VGA_WIDTH * 2) ) {
69         video_base_addr[idx++] = 0x20;
70         video_base_addr[idx++] = VGA_COLOR_ERASE;
71     }
72 }
```

4.44.3.2 vga_pos_t vga_get_cursor_pos (void)

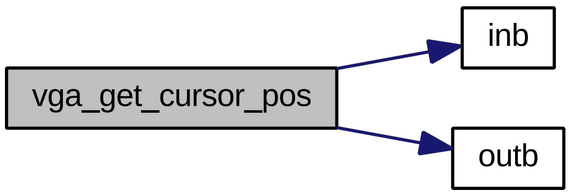
Definition at line 89 of file vga.c.

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

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

```
                                {
    unsigned char h, l;

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

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

Here is the call graph for this function:



4.44.3.3 void vga_init (void)

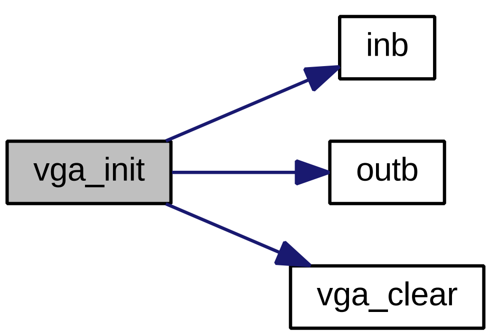
Definition at line 43 of file vga.c.

References inb(), outb(), vga_clear(), VGA_CRTC_ADDR, VGA_CRTC_DATA, VGA_MISC_OUT_RD, and VGA_MIS-C_OUT_WR.

Referenced by console_init().

```
43                     {
44     unsigned char data;
45
46     /* Set address select bit in a known state: CRTC regs at 0x3dx */
47     data = inb(VGA_MISC_OUT_RD);
48     data |= 1;
49     outb(VGA_MISC_OUT_WR, data);
50
51     /* Move cursor to line 0 col 0 */
52     outb(VGA_CRTC_ADDR, 0x0e);
53     outb(VGA_CRTC_DATA, 0x0);
54     outb(VGA_CRTC_ADDR, 0x0f);
55     outb(VGA_CRTC_DATA, 0x0);
56
57     /* Clear the screen */
58     vga_clear();
59 }
```

Here is the call graph for this function:



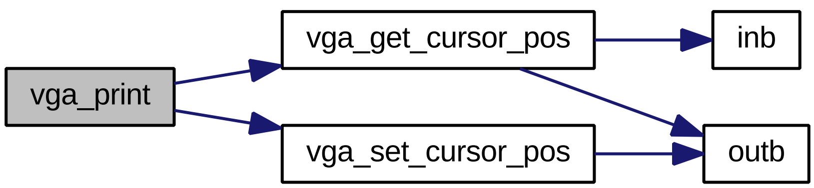
4.44.3.4 void vga_print (const char ∗ *message,* int *colour*)

Definition at line 111 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

```
111                                           {
112     unsigned short int pos = vga_get_cursor_pos();
113     char c;
114 
115     while( (c = *(message++)) ) {
116         pos = vga_raw_putc(c, pos, colour);
117     }
118 
119     vga_set_cursor_pos(pos);
120 }
```

Here is the call graph for this function:



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

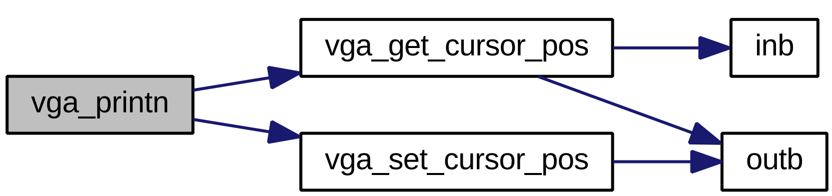
Definition at line 122 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by console_printn().

```
122                                                             {
123     vga_pos_t pos = vga_get_cursor_pos();
124     char c;
125 
126     while(n) {
127         c = *(message++);
128         pos = vga_raw_putc(c, pos, colour);
129         --n;
130     }
131 
132     vga_set_cursor_pos(pos);
133 }
```

Here is the call graph for this function:



4.44.3.6 void vga_putc (char *c*, int *colour*)

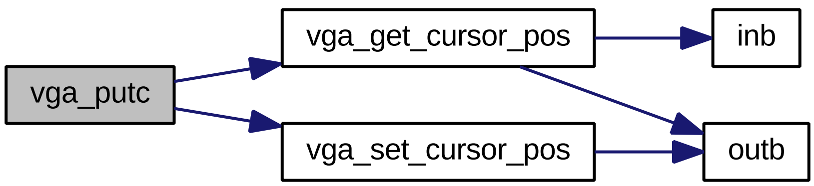
Definition at line 135 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by console_putc().

```
135                                 {
136     vga_pos_t pos = vga_get_cursor_pos();
137 
138     pos = vga_raw_putc(c, pos, colour);
139 
140     vga_set_cursor_pos(pos);
141 }
```

Here is the call graph for this function:



4.44.3.7 void vga_scroll (void)

Definition at line 74 of file vga.c.

References VGA_COLOR_ERASE, VGA_LINES, and VGA_WIDTH.

```
74                      {
75     unsigned char *di = video_base_addr;
76     unsigned char *si = video_base_addr + 2 * VGA_WIDTH;
77     unsigned int idx;
78 
79     for(idx = 0; idx < 2 * VGA_WIDTH * (VGA_LINES - 1); ++idx) {
80         *(di++) = *(si++);
81     }
82 
83     for(idx = 0; idx < VGA_WIDTH; ++idx) {
84         *(di++) = 0x20;
85         *(di++) = VGA_COLOR_ERASE;
86     }
87 }
```

4.44.3.8 void vga_set_base_addr (void * *base_addr*)

Definition at line 61 of file vga.c.

Referenced by vm_boot_init().

```
61                                      {
62     video_base_addr = base_addr;
63 }
```

4.44.3.9 void vga_set_cursor_pos (vga_pos_t *pos*)

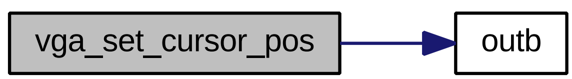
Definition at line 100 of file vga.c.

References outb(), VGA_CRTC_ADDR, and VGA_CRTC_DATA.

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

```
100                                         {
101     unsigned char h = pos >> 8;
102     unsigned char l = pos;
103
104     outb(VGA_CRTC_ADDR, 0x0e);
105     outb(VGA_CRTC_DATA, h);
106     outb(VGA_CRTC_ADDR, 0x0f);
107     outb(VGA_CRTC_DATA, l);
108 }
```

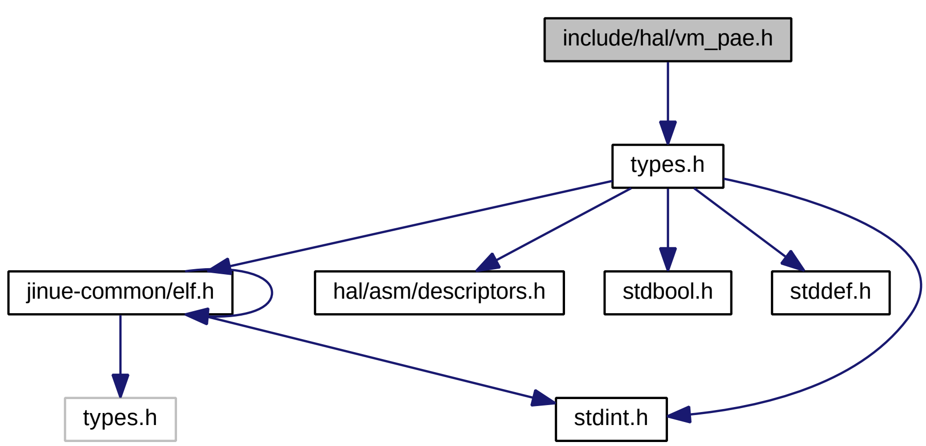
Here is the call graph for this function:



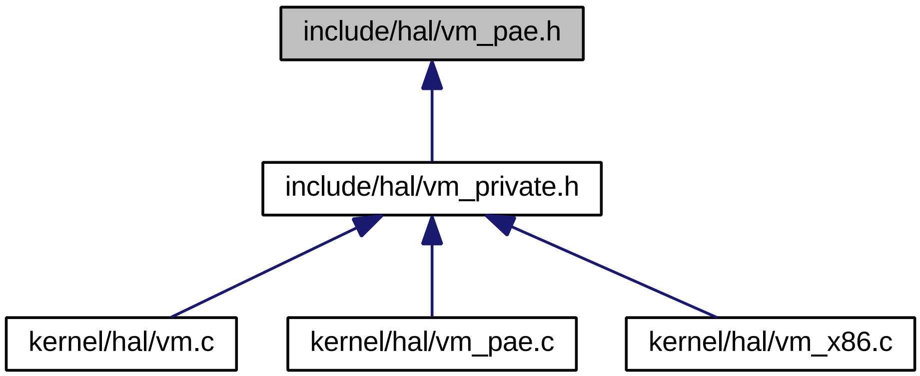
4.45 include/hal/vm_pae.h File Reference

```
#include <types.h>
```

Include dependency graph for vm_pae.h:



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



Functions

- void **vm_pae_boot_init** (void)

  *This header file contains declarations for the PAE functions defined in **hal/vm_pae.c** (p. 342).*
- **pte_t** ∗ **vm_pae_lookup_page_directory** (**addr_space_t** ∗addr_space, void ∗addr, **bool** create_as_needed)

  Lookup and map the page directory for a specified address and address space.
- unsigned int **vm_pae_page_table_offset_of** (**addr_t** addr)
- unsigned int **vm_pae_page_directory_offset_of** (**addr_t** addr)
- **pte_t** ∗ **vm_pae_get_pte_with_offset** (**pte_t** ∗pte, unsigned int offset)
- void **vm_pae_set_pte** (**pte_t** ∗pte, **uint64_t** paddr, int flags)

  TODO handle flag bit position > 31 for NX bit support.
- void **vm_pae_set_pte_flags** (**pte_t** ∗pte, int flags)

  TODO handle flag bit position > 31 for NX bit support.
- int **vm_pae_get_pte_flags** (const **pte_t** ∗pte)
- **uint64_t vm_pae_get_pte_paddr** (const **pte_t** ∗pte)

  TODO mask NX bit as well, maximum 52 bits supported.
- void **vm_pae_clear_pte** (**pte_t** ∗pte)
- void **vm_pae_copy_pte** (**pte_t** ∗dest, const **pte_t** ∗src)
- **addr_space_t** ∗ **vm_pae_create_addr_space** (**addr_space_t** ∗addr_space)
- **addr_space_t** ∗ **vm_pae_create_initial_addr_space** (**boot_alloc_t** ∗boot_alloc)
- void **vm_pae_destroy_addr_space** (**addr_space_t** ∗addr_space)
- void **vm_pae_create_pdpt_cache** (**boot_alloc_t** ∗boot_alloc)
- void **vm_pae_unmap_low_alias** (**addr_space_t** ∗addr_space)

4.45.1 Function Documentation

4.45.1.1 void vm_pae_boot_init (void)

This header file contains declarations for the PAE functions defined in **hal/vm_pae.c** (p. 342).

It is intended to be included by **hal/vm.c** (p. 331) and **hal/vm_pae.c** (p. 342). There should be no reason to include it anywhere else.

Definition at line 72 of file vm_pae.c.

References PAGE_TABLE_ENTRIES, and page_table_entries.

Referenced by vm_boot_init().

```
72                          {
73     page_table_entries = (size_t)PAGE_TABLE_ENTRIES;
74 }
```

4.45.1.2 void vm_pae_clear_pte (pte_t ∗ *pte*)

Definition at line 150 of file vm_pae.c.

References pte_t::entry.

Referenced by vm_pae_create_addr_space(), vm_pae_create_initial_addr_space(), and vm_pae_unmap_low_alias().

```
150                               {
151     pte->entry = 0;
152 }
```

4.45.1.3 void vm_pae_copy_pte (pte_t ∗ *dest,* const pte_t ∗ *src*)

Definition at line 154 of file vm_pae.c.

References pte_t::entry.

Referenced by vm_pae_create_addr_space().

```
154                                                {
155     dest->entry = src->entry;
156 }
```

4.45.1.4 addr_space_t∗ vm_pae_create_addr_space (addr_space_t ∗ *addr_space*)

Definition at line 170 of file vm_pae.c.

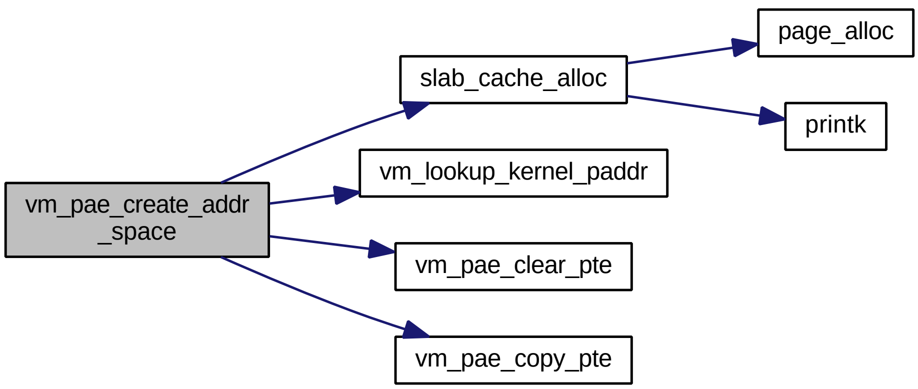
References addr_space_t::cr3, initial_addr_space, KLIMIT, NULL, page_address_of, page_offset_of, pdpt_t::pd, addr-_space_t::pdpt, PDPT_ENTRIES, slab_cache_alloc(), addr_space_t::top_level, vm_lookup_kernel_paddr(), vm_pae_-clear_pte(), and vm_pae_copy_pte().

Referenced by vm_create_addr_space().

```
170                                                            {
171     unsigned int idx;
172     pte_t *pdpte;
173 
174     /* Create a PDPT for the new address space */
175     pdpt_t *pdpt = slab_cache_alloc(&pdpt_cache);
176 
177     if(pdpt == NULL) {
178         return NULL;
179     }
180 
181     /* Use the initial address space as a template for the kernel address range
182      * (address KLIMIT and above). The page tables for that range are shared by
183      * all address spaces. */
184     pdpt_t *template_pdpt = initial_addr_space.top_level.pdpt;
185 
186     for(idx = 0; idx < PDPT_ENTRIES; ++idx) {
187         pdpte = &pdpt->pd[idx];
188 
189         if(idx < pdpt_offset_of((addr_t)KLIMIT)) {
190             /* This PDPT entry describes an address range entirely under KLIMIT
191              * so it is all user space: do not create a page directory at this
```

```
             * time. */
            vm_pae_clear_pte(pdpte);
        }
        else {
            /* This page directory describes an address range entirely above
             * KLIMIT: share the template's page directory. */
            vm_pae_copy_pte(pdpte, &template_pdpt->pd[idx]);
        }
    }

    /* Lookup the physical address of the page where the PDPT resides. */
    kern_paddr_t pdpt_page_paddr = vm_lookup_kernel_paddr((addr_t)
  page_address_of(pdpt));

    /* physical address of PDPT */
    kern_paddr_t pdpt_paddr = pdpt_page_paddr | page_offset_of(pdpt);

    addr_space->top_level.pdpt  = pdpt;
    addr_space->cr3             = pdpt_paddr;

    return addr_space;
}
```

Here is the call graph for this function:



4.45.1.5 addr_space_t∗ vm_pae_create_initial_addr_space (boot_alloc_t ∗ *boot_alloc*)

Definition at line 258 of file vm_pae.c.

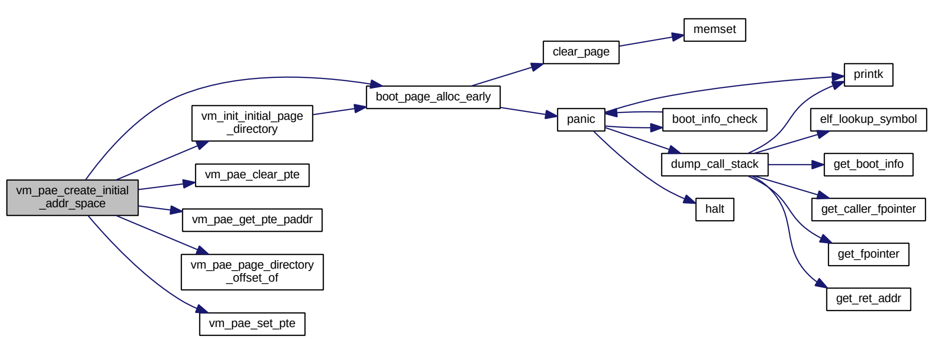
References boot_heap_alloc, boot_page_alloc_early(), addr_space_t::cr3, EARLY_PHYS_TO_VIRT, EARLY_PTR_-TO_PHYS_ADDR, EARLY_VIRT_TO_PHYS, initial_addr_space, initial_pdpt, KERNEL_PREALLOC_LIMIT, KLIMIT, page_table_entries, pdpt_t::pd, addr_space_t::pdpt, PDPT_ENTRIES, addr_space_t::top_level, VM_FLAG_PRESENT, vm_init_initial_page_directory(), vm_pae_clear_pte(), vm_pae_get_pte_paddr(), vm_pae_page_directory_offset_of(), and vm_pae_set_pte().

Referenced by vm_create_initial_addr_space().

```
258                                                                           {
259
260     unsigned int idx;
261
262     /* Allocate initial PDPT. PDPT must be 32-byte aligned. */
263     initial_pdpt = boot_heap_alloc(boot_alloc, pdpt_t, 32);
264
265     /* We want the pre-allocated kernel page tables to be contiguous. For this
266      * reason, we allocate the page directories first, and then the page tables.
267      *
268      * This function allocates pages in this order:
269      *      +---------------+-------...--------+-------...------+
270      *      |    Low alias   |  pre-allocated   |  pre-allocated |
```

```
     *      | page directory |      kernel      |     kernel     |
     *      | and page table | page directories |  page tables   |
     *      +----------------+--------...--------+-------...------+
     * */

    for(idx = 0; idx < PDPT_ENTRIES; ++idx) {
        vm_pae_clear_pte(&initial_pdpt->pd[idx]);
    }

    vm_pae_init_low_alias(initial_pdpt, boot_alloc);

    const unsigned int last_idx = pdpt_offset_of((addr_t)KERNEL_PREALLOC_LIMIT - 1);

    for(idx = pdpt_offset_of((addr_t)KLIMIT); idx <= last_idx; ++idx) {
        pte_t *const pdpte      = &initial_pdpt->pd[idx];
        pte_t *page_directory   = (pte_t *)boot_page_alloc_early(boot_alloc);

        vm_pae_set_pte(
                pdpte,
                EARLY_PTR_TO_PHYS_ADDR(page_directory),
                VM_FLAG_PRESENT);
    }

    for(idx = pdpt_offset_of((addr_t)KLIMIT); idx <= last_idx; ++idx) {
        unsigned int end_index;

        pte_t *const pdpte = &initial_pdpt->pd[idx];
        pte_t *const page_directory = (pte_t *)EARLY_PHYS_TO_VIRT(
      vm_pae_get_pte_paddr(pdpte));

        if(idx < pdpt_offset_of((addr_t)KERNEL_PREALLOC_LIMIT)) {
            end_index = page_table_entries;
        }
        else {
            end_index = vm_pae_page_directory_offset_of((addr_t)KERNEL_PREALLOC_LIMIT);
        }

        vm_init_initial_page_directory(
                page_directory,
                boot_alloc,
                0,
                end_index,
                idx == pdpt_offset_of((addr_t)KLIMIT));
    }

    initial_addr_space.top_level.pdpt   = initial_pdpt;
    initial_addr_space.cr3              = EARLY_VIRT_TO_PHYS(initial_pdpt);

    return &initial_addr_space;
}
```

Here is the call graph for this function:



4.45.1.6 void vm_pae_create_pdpt_cache (boot_alloc_t * *boot_alloc*)

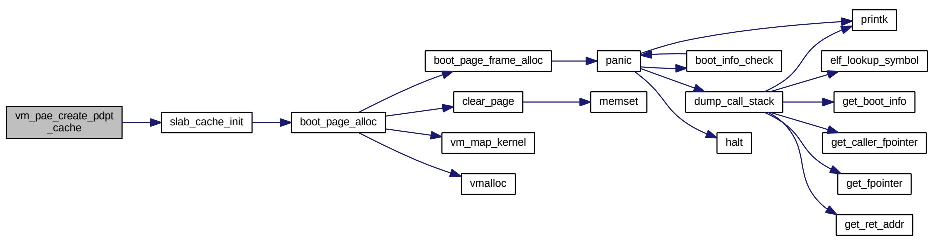
Definition at line 158 of file vm_pae.c.

References NULL, slab_cache_init(), and SLAB_DEFAULTS.

Referenced by vm_boot_postinit().

```
158                                                         {
159     slab_cache_init(
160             &pdpt_cache,
161             "vm_pae_pdpt_cache",
162             sizeof(pdpt_t),
163             sizeof(pdpt_t),
164             NULL,
165             NULL,
166             SLAB_DEFAULTS,
167             boot_alloc);
168 }
```

Here is the call graph for this function:



4.45.1.7 void vm_pae_destroy_addr_space (addr_space_t ∗ *addr_space*)

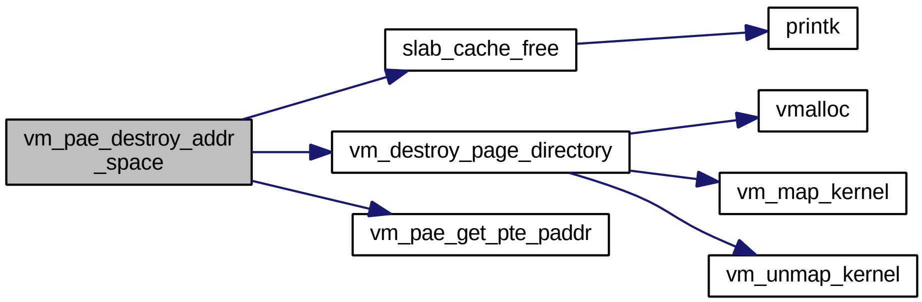
Definition at line 321 of file vm_pae.c.

References pte_t::entry, KLIMIT, pdpt_t::pd, addr_space_t::pdpt, PDPT_ENTRIES, slab_cache_free(), addr_space_t::top_level, vm_destroy_page_directory(), VM_FLAG_PRESENT, and vm_pae_get_pte_paddr().

Referenced by vm_destroy_addr_space().

```
321                                                          {
322     unsigned int idx;
323     pte_t pdpte;
324 
325     pdpt_t *pdpt = addr_space->top_level.pdpt;
326 
327     for(idx = 0; idx < PDPT_ENTRIES; ++idx) {
328         pdpte.entry = pdpt->pd[idx].entry;
329 
330         if(idx < pdpt_offset_of((addr_t)KLIMIT)) {
331             /* This page directory describes an address range entirely under
332              * KLIMIT so it is all user space: free all page tables in this
333              * page directory as well as the page directory itself. */
334             if(pdpte.entry & VM_FLAG_PRESENT) {
335                 vm_destroy_page_directory(
336                         vm_pae_get_pte_paddr(&pdpte),
337                         0,
338                         page_table_entries);
339             }
340         }
341         else {
342             /* This page directory describes an address range entirely above
343              * KLIMIT: do nothing.
344              *
345              * The page directory must not be freed because it is shared by all
346              * address spaces. */
347         }
348     }
349 
350     slab_cache_free(pdpt);
351 }
```

Here is the call graph for this function:



4.45.1.8 int vm_pae_get_pte_flags (const pte_t * *pte*)

Definition at line 141 of file vm_pae.c.

References pte_t::entry, and PAGE_MASK.

Referenced by vm_pae_lookup_page_directory().

```
141                                        {
142     return pte->entry & PAGE_MASK;
143 }
```

4.45.1.9 uint64_t vm_pae_get_pte_paddr (const pte_t * *pte*)

TODO mask NX bit as well, maximum 52 bits supported.

Definition at line 146 of file vm_pae.c.

References pte_t::entry, and PAGE_MASK.

Referenced by vm_pae_create_initial_addr_space(), vm_pae_destroy_addr_space(), and vm_pae_lookup_page_-directory().

```
146                                             {
147     return (pte->entry & ~(uint64_t)PAGE_MASK);
148 }
```

4.45.1.10 pte_t* vm_pae_get_pte_with_offset (pte_t * *pte,* unsigned int *offset*)

Definition at line 127 of file vm_pae.c.

```
127                                                                    {
128     return &pte[offset];
129 }
```

4.45.1.11 pte_t* vm_pae_lookup_page_directory (addr_space_t * *addr_space,* void * *addr,* bool *create_as_needed*)

Lookup and map the page directory for a specified address and address space.

Important note: it is the caller's responsibility to unmap and free the returned page directory when it is done with it.

Parameters

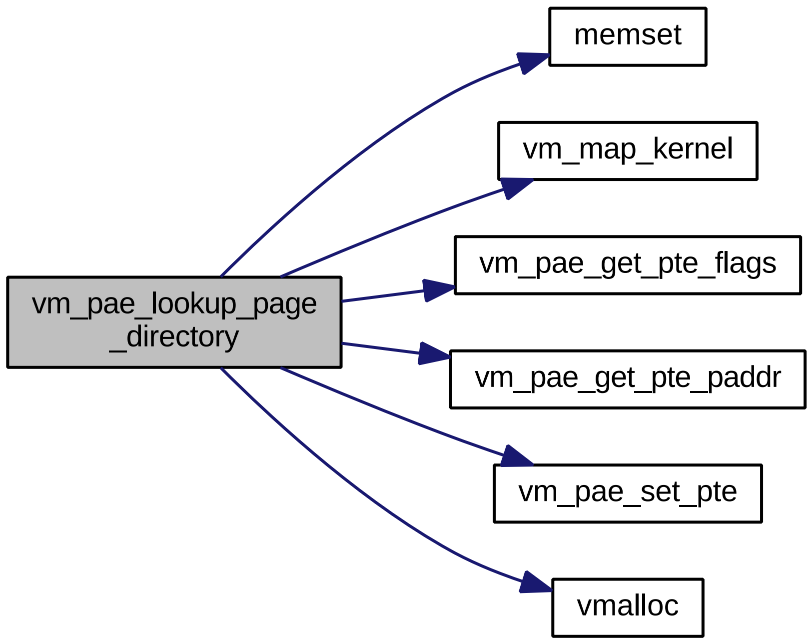
addr_space	address space in which the address is looked up.
addr	address to look up
create_as_need	Whether a page table is allocated if it does not exist

Definition at line 86 of file vm_pae.c.

References memset(), NULL, PAGE_SIZE, pdpt_t::pd, addr_space_t::pdpt, pfalloc, addr_space_t::top_level, VM_FLAG_PRESENT, VM_FLAG_READ_WRITE, vm_map_kernel(), vm_pae_get_pte_flags(), vm_pae_get_pte_paddr(), vm_pae_set_pte(), and vmalloc().

```
86                                                                                                                    {
87     pdpt_t *pdpt    = addr_space->top_level.pdpt;
88     pte_t  *pdpte   = &pdpt->pd[pdpt_offset_of(addr)];
89 
90     if(vm_pae_get_pte_flags(pdpte) & VM_FLAG_PRESENT) {
91         /* map page directory */
92         pte_t *page_directory   = (pte_t *)vmalloc();
93         vm_map_kernel((addr_t)page_directory, vm_pae_get_pte_paddr(pdpte),
      VM_FLAG_READ_WRITE);
94 
95         return page_directory;
96     }
97     else {
98         if(create_as_needed) {
99             /* allocate a new page directory and map it */
100            pte_t *page_directory       = (pte_t *)vmalloc();
101            kern_paddr_t pgdir_paddr    = pfalloc();
102 
103            vm_map_kernel((addr_t)page_directory, pgdir_paddr,
      VM_FLAG_READ_WRITE);
104 
105            /* zero content of page directory */
106            memset(page_directory, 0, PAGE_SIZE);
107 
108            /* link page directory in PDPT */
109            vm_pae_set_pte(pdpte, pgdir_paddr, VM_FLAG_PRESENT);
110 
111            return page_directory;
112        }
113        else {
114            return NULL;
115        }
116    }
117 }
```

Here is the call graph for this function:



4.45.1.12 unsigned int vm_pae_page_directory_offset_of (addr_t *addr*)

Definition at line 123 of file vm_pae.c.

References PAGE_DIRECTORY_OFFSET_OF.

Referenced by vm_pae_create_initial_addr_space().

```
123                                                             {
124     return PAGE_DIRECTORY_OFFSET_OF(addr);
125 }
```

4.45.1.13 unsigned int vm_pae_page_table_offset_of (addr_t *addr*)

Definition at line 119 of file vm_pae.c.

References PAGE_TABLE_OFFSET_OF.

```
119                                                         {
120     return PAGE_TABLE_OFFSET_OF(addr);
121 }
```

4.45.1.14 void vm_pae_set_pte (pte_t * *pte,* uint64_t *paddr,* int *flags*)

TODO handle flag bit position $>$ 31 for NX bit support.

Definition at line 132 of file vm_pae.c.

References pte_t::entry.

Referenced by vm_pae_create_initial_addr_space(), and vm_pae_lookup_page_directory().

```
132                                                         {
133     pte->entry = paddr | flags;
134 }
```

4.45.1.15 void vm_pae_set_pte_flags (pte_t ∗ *pte,* int *flags*)

TODO handle flag bit position > 31 for NX bit support.

Definition at line 137 of file vm_pae.c.

References pte_t::entry, and PAGE_MASK.

```
137                                               {
138     pte->entry = (pte->entry & ~(uint64_t)PAGE_MASK) | flags;
139 }
```

4.45.1.16 void vm_pae_unmap_low_alias (addr_space_t ∗ *addr_space*)

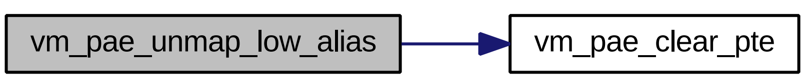
Definition at line 353 of file vm_pae.c.

References pdpt_t::pd, addr_space_t::pdpt, addr_space_t::top_level, and vm_pae_clear_pte().

Referenced by vm_boot_init().

```
353                                                      {
354     /* Enabling PAE requires disabling paging temporarily, which in turn requires
355      * an alias of the kernel image region at address 0 to match its physical
356      * address. This function gets rid of this alias once PAE is enabled.
357      *
358      * There is no need for TLB invalidation because the caller reloads CR3 just
359      * after calling this function. */
360     vm_pae_clear_pte(&addr_space->top_level.pdpt->pd[0]);
361 }
```

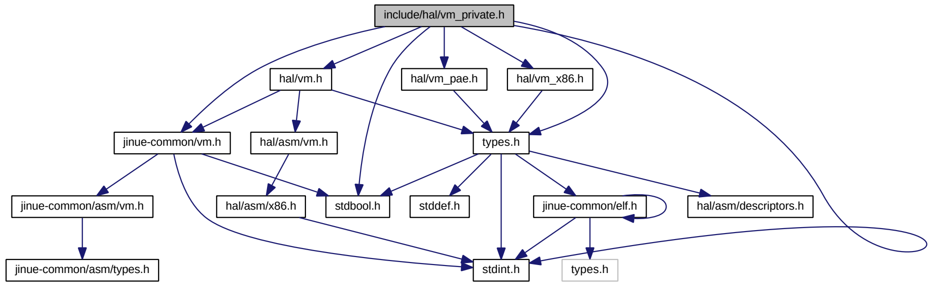
Here is the call graph for this function:



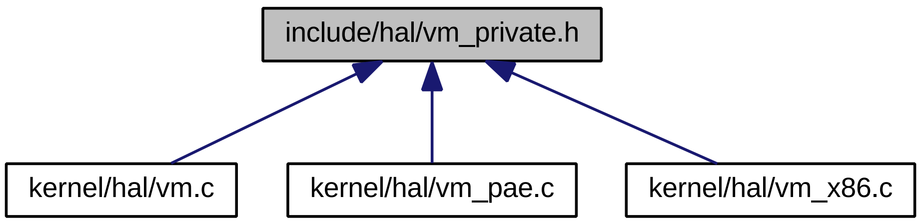
4.46 include/hal/vm_private.h File Reference

#include <jinue-common/vm.h>
#include <hal/vm.h>
#include <hal/vm_pae.h>
#include <hal/vm_x86.h>
#include <stdbool.h>
#include <stdint.h>
#include <types.h>

Include dependency graph for vm_private.h:



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



Macros

- #define **PAGE_TABLE_ENTRIES** (**PAGE_SIZE** / sizeof(**pte_t**))

    *This header file contains private definitions shared by **hal/vm.c** (p. 331), **hal/vm_pae.c** (p. 342) and **hal/vm_x86.c** (p. 351).*
- #define **PAGE_TABLE_MASK** (**PAGE_TABLE_ENTRIES** - 1)

    bit mask for page table or page directory offset
- #define **PAGE_TABLE_OFFSET_OF**(x) (((**uint32_t**)(x) / **PAGE_SIZE**) & **PAGE_TABLE_MASK**)

    page table entry offset of virtual (linear) address
- #define **PAGE_DIRECTORY_OFFSET_OF**(x) (((**uint32_t**)(x) / (**PAGE_SIZE** ∗ **PAGE_TABLE_ENTRIES**)) & **PAGE_TABLE_MASK**)

    page directory entry offset of virtual (linear address)

Functions

- **kern_paddr_t vm_clone_page_directory** (**kern_paddr_t** template_paddr, unsigned int start_index)
- void **vm_init_initial_page_directory** (**pte_t** ∗page_directory, **boot_alloc_t** ∗boot_alloc, unsigned int start_index, unsigned int end_index, **bool** first_directory)
- void **vm_destroy_page_directory** (**kern_paddr_t** pgdir_paddr, unsigned int from_index, unsigned int to_index)

Variables

- **pte_t** ∗ **global_page_tables**
- **size_t page_table_entries**

4.46.1 Macro Definition Documentation

4.46.1.1 #define PAGE_DIRECTORY_OFFSET_OF(*x*) (((uint32_t)(x) / (PAGE_SIZE ∗ PAGE_TABLE_ENTRIES)) & PAGE_TABLE_MASK)

page directory entry offset of virtual (linear address)

Definition at line 56 of file vm_private.h.

Referenced by vm_pae_page_directory_offset_of(), and vm_x86_page_directory_offset_of().

4.46.1.2 #define PAGE_TABLE_ENTRIES (PAGE_SIZE / sizeof(pte_t))

This header file contains private definitions shared by **hal/vm.c** (p. 331), **hal/vm_pae.c** (p. 342) and **hal/vm_x86.c** (p. 351).

There should be no reason to include it anywhere else. number of entries in page table or page directory

Definition at line 47 of file vm_private.h.

Referenced by vm_pae_boot_init(), and vm_x86_boot_init().

4.46.1.3 #define PAGE_TABLE_MASK (PAGE_TABLE_ENTRIES - 1)

bit mask for page table or page directory offset

Definition at line 50 of file vm_private.h.

4.46.1.4 #define PAGE_TABLE_OFFSET_OF(*x*) (((uint32_t)(x) / PAGE_SIZE) & PAGE_TABLE_MASK)

page table entry offset of virtual (linear) address

Definition at line 53 of file vm_private.h.

Referenced by vm_pae_page_table_offset_of(), and vm_x86_page_table_offset_of().

4.46.2 Function Documentation

4.46.2.1 kern_paddr_t vm_clone_page_directory (kern_paddr_t *template_paddr,* unsigned int *start_index*)

Definition at line 489 of file vm.c.

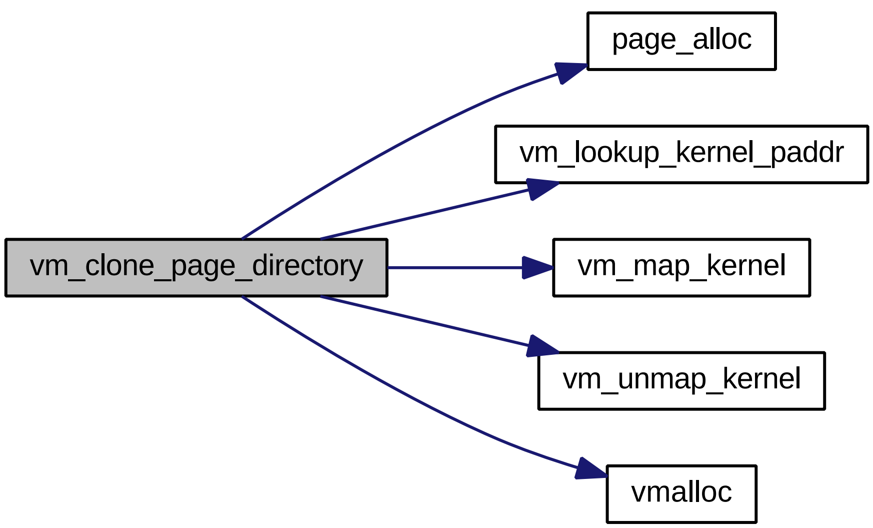
References page_alloc(), page_table_entries, VM_FLAG_READ_WRITE, vm_lookup_kernel_paddr(), vm_map_-kernel(), vm_map_kernel(), and vmalloc().

Referenced by vm_x86_create_addr_space().

```
489                                                                                          {
490     unsigned int     idx;
491
492     /* Allocate new page directory.
493      *
494      * TODO handle allocation failure */
495     pte_t *page_directory = (pte_t *)page_alloc();
496
497     /* map page directory template */
498     pte_t *template = (pte_t *)vmalloc();
499     vm_map_kernel((addr_t)template, template_paddr, VM_FLAG_READ_WRITE);
500
501     /* clear all entries below index start_index */
```

```
    for(idx = 0; idx < start_index; ++idx) {
        clear_pte( get_pte_with_offset(page_directory, idx) );
    }

    /* copy entries from template for indexes start_index and above */
    for(idx = start_index; idx < page_table_entries; ++idx) {
        copy_pte(
            get_pte_with_offset(page_directory, idx),
            get_pte_with_offset(template, idx)
        );
    }

    vm_unmap_kernel((addr_t)page_directory);
    vm_unmap_kernel((addr_t)template);

    return vm_lookup_kernel_paddr((addr_t)page_directory);
}
```

Here is the call graph for this function:



4.46.2.2 void vm_destroy_page_directory (kern_paddr_t *pgdir_paddr,* unsigned int *from_index,* unsigned int *to_index*)

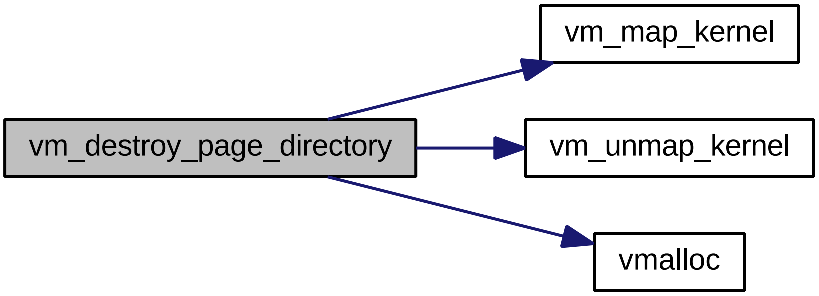
Definition at line 584 of file vm.c.

References pffree, VM_FLAG_READ_WRITE, vm_map_kernel(), vm_unmap_kernel(), and vmalloc().

Referenced by vm_pae_destroy_addr_space(), and vm_x86_destroy_addr_space().

```
584                                                                                                       {
585     unsigned int idx;
586
587     pte_t *page_directory = (pte_t *)vmalloc();
588     vm_map_kernel((addr_t)page_directory, pgdir_paddr, VM_FLAG_READ_WRITE);
589
590     /* be careful not to free the kernel page tables */
591     for(idx = from_index; idx < to_index; ++idx) {
592         pte_t *pte = get_pte_with_offset(page_directory, idx);
593
594         if(get_pte_flags(pte) & VM_FLAG_PRESENT) {
595             pffree( get_pte_paddr(pte) );
596         }
597     }
598
599     vm_unmap_kernel((addr_t)page_directory);
600     pffree(pgdir_paddr);
601 }
```

Here is the call graph for this function:



4.46.2.3 void vm_init_initial_page_directory (pte_t ∗ *page_directory,* boot_alloc_t ∗ *boot_alloc,* unsigned int *start_index,* unsigned int *end_index,* bool *first_directory*)

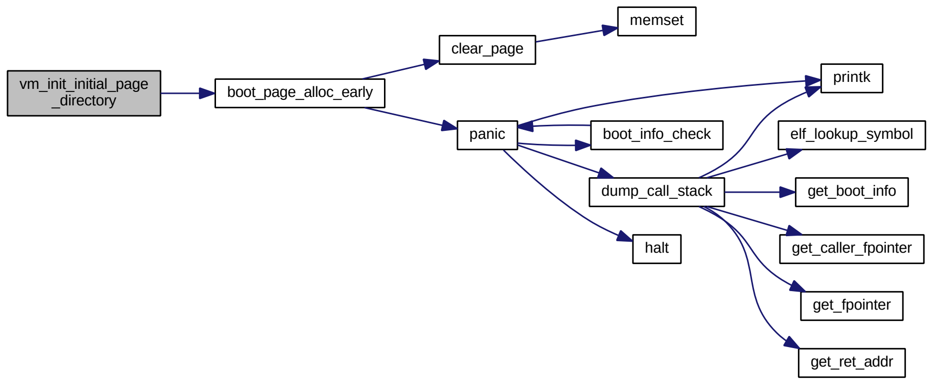
Definition at line 529 of file vm.c.

References boot_page_alloc_early(), EARLY_PTR_TO_PHYS_ADDR, page_table_entries, and VM_FLAG_READ_WRITE.

Referenced by vm_pae_create_initial_addr_space(), and vm_x86_create_initial_addr_space().

```
                                           {

    unsigned int idx, idy;

    /* Allocate page tables and initialize page directory entries. */
    for(idx = 0; idx < page_table_entries; ++idx) {
        if(idx < start_index || idx >= end_index) {
            /* Clear page directory entries for user space and non-preallocated
             * kernel page tables. */
            clear_pte( get_pte_with_offset(page_directory, idx) );
        }
        else {
            /* Allocate page tables for kernel data/code region.
             *
             * Note that the use of pfalloc_early() here guarantees that the
             * page tables are allocated contiguously, and that they keep the
             * same address once paging is enabled. */
            pte_t *page_table = (pte_t *)boot_page_alloc_early(boot_alloc);

            if(first_directory && idx == start_index) {
                /* remember the address of the first page table for use by
                 * vm_map() later */
                global_page_tables = page_table;
            }

            set_pte(
                get_pte_with_offset(page_directory, idx),
                EARLY_PTR_TO_PHYS_ADDR(page_table),
                VM_FLAG_PRESENT | VM_FLAG_READ_WRITE);

            /* clear page table */
            for(idy = 0; idy < page_table_entries; ++idy) {
                clear_pte( get_pte_with_offset(page_table, idy) );
            }
        }
    }
}
```

Here is the call graph for this function:



4.46.3 Variable Documentation

4.46.3.1 pte_t∗ global_page_tables

Definition at line 49 of file vm.c.

4.46.3.2 size_t page_table_entries

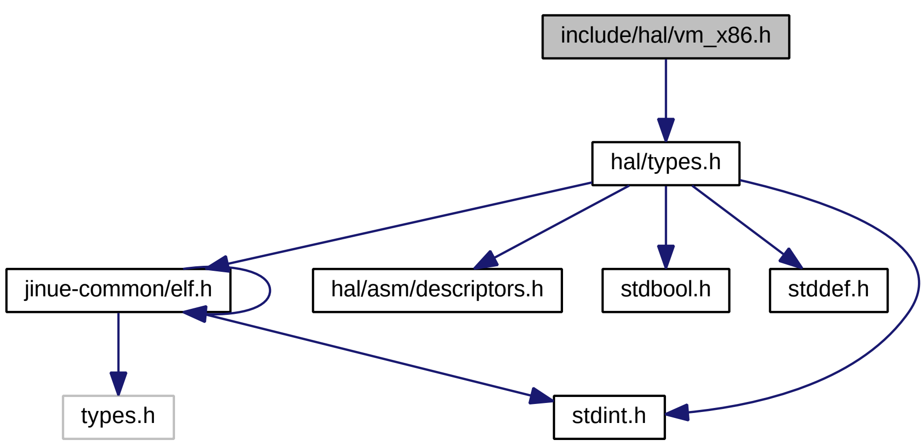
Definition at line 53 of file vm.c.

Referenced by vm_clone_page_directory(), vm_init_initial_page_directory(), vm_pae_boot_init(), vm_pae_create_-initial_addr_space(), and vm_x86_boot_init().

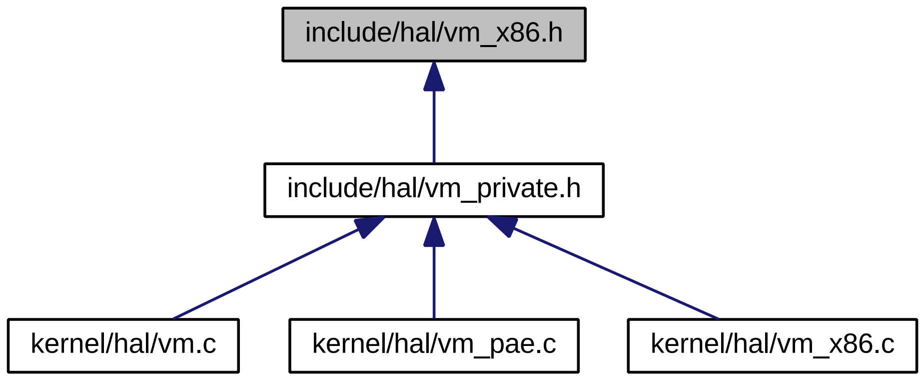
4.47 include/hal/vm_x86.h File Reference

`#include <hal/types.h>`

Include dependency graph for vm_x86.h:



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



Functions

- void **vm_x86_boot_init** (void)

  *This header file contains declarations for the non-PAE functions defined in **hal/vm_x86.c** (p. 351).*
- **addr_space_t** ∗ **vm_x86_create_addr_space** (**addr_space_t** ∗addr_space)
- **addr_space_t** ∗ **vm_x86_create_initial_addr_space** (**boot_alloc_t** ∗boot_alloc)
- void **vm_x86_destroy_addr_space** (**addr_space_t** ∗addr_space)
- unsigned int **vm_x86_page_table_offset_of** (**addr_t** addr)
- unsigned int **vm_x86_page_directory_offset_of** (**addr_t** addr)
- **pte_t** ∗ **vm_x86_lookup_page_directory** (**addr_space_t** ∗addr_space)

  Lookup and map the page directory for a specified address and address space.
- **pte_t** ∗ **vm_x86_get_pte_with_offset** (**pte_t** ∗pte, unsigned int offset)
- void **vm_x86_set_pte** (**pte_t** ∗pte, **uint32_t** paddr, int flags)
- void **vm_x86_set_pte_flags** (**pte_t** ∗pte, int flags)
- int **vm_x86_get_pte_flags** (const **pte_t** ∗pte)
- **uint32_t vm_x86_get_pte_paddr** (const **pte_t** ∗pte)
- void **vm_x86_clear_pte** (**pte_t** ∗pte)
- void **vm_x86_copy_pte** (**pte_t** ∗dest, const **pte_t** ∗src)

4.47.1 Function Documentation

4.47.1.1 void vm_x86_boot_init (void)

This header file contains declarations for the non-PAE functions defined in **hal/vm_x86.c** (p. 351).

It is intended to be included by **hal/vm.c** (p. 331) and **hal/vm_x86.c** (p. 351). There should be no reason to include it anywhere else.

Definition at line 41 of file vm_x86.c.

References PAGE_TABLE_ENTRIES, and page_table_entries.

Referenced by vm_boot_init().

```
41                              {
42     page_table_entries = (size_t)PAGE_TABLE_ENTRIES;
43 }
```

4.47.1.2 void vm_x86_clear_pte (pte_t * *pte*)

Definition at line 133 of file vm_x86.c.

References pte_t::entry.

```
133                                 {
134     pte->entry = 0;
135 }
```

4.47.1.3 void vm_x86_copy_pte (pte_t * *dest,* const pte_t * *src*)

Definition at line 137 of file vm_x86.c.

References pte_t::entry.

```
137                                                 {
138     dest->entry = src->entry;
139 }
```

4.47.1.4 addr_space_t* vm_x86_create_addr_space (addr_space_t * *addr_space*)

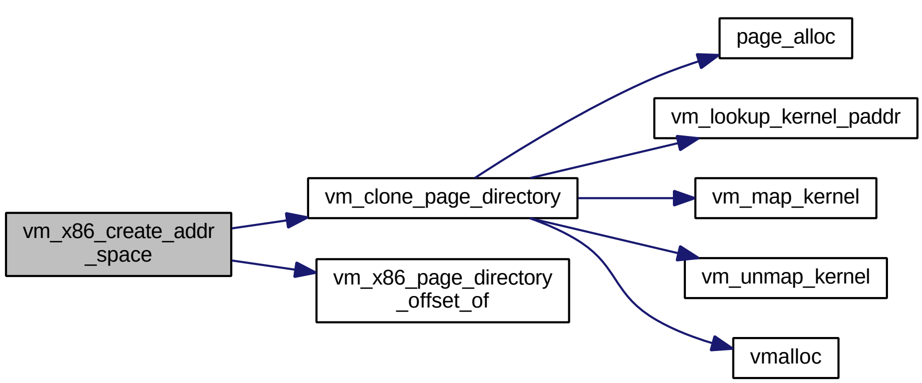
Definition at line 45 of file vm_x86.c.

References addr_space_t::cr3, initial_addr_space, KLIMIT, addr_space_t::pd, addr_space_t::top_level, vm_clone_page_directory(), and vm_x86_page_directory_offset_of().

Referenced by vm_create_addr_space().

```
45                                                            {
46      /* Create a new page directory where entries for the address range starting
47       * at KLIMIT are copied from the initial address space. The mappings starting
48       * at KLIMIT belong to the kernel and are identical in all address spaces. */
49      kern_paddr_t paddr = vm_clone_page_directory(
50              initial_addr_space.top_level.pd,
51              vm_x86_page_directory_offset_of((addr_t)KLIMIT));
52
53      addr_space->top_level.pd = paddr;
54      addr_space->cr3          = paddr;
55
56      return addr_space;
57 }
```

Here is the call graph for this function:



4.47.1.5 addr_space_t∗ vm_x86_create_initial_addr_space (boot_alloc_t ∗ *boot_alloc*)

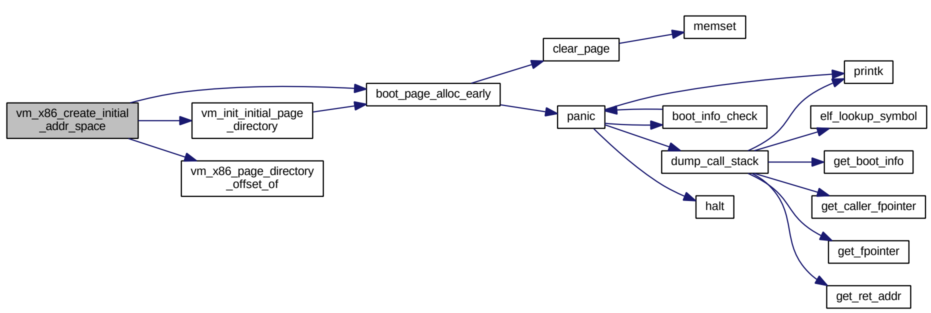
Definition at line 59 of file vm_x86.c.

References boot_page_alloc_early(), addr_space_t::cr3, EARLY_PTR_TO_PHYS_ADDR, EARLY_VIRT_TO_PHYS, initial_addr_space, KERNEL_PREALLOC_LIMIT, KLIMIT, addr_space_t::pd, addr_space_t::top_level, vm_init_initial_page_directory(), and vm_x86_page_directory_offset_of().

Referenced by vm_create_initial_addr_space().

```
59                                                                          {
60      pte_t *page_directory = (pte_t *)boot_page_alloc_early(boot_alloc);
61 
62      vm_init_initial_page_directory(
63              page_directory,
64              boot_alloc,
65              vm_x86_page_directory_offset_of((addr_t)KLIMIT),
66              vm_x86_page_directory_offset_of((addr_t)KERNEL_PREALLOC_LIMIT),
67              true);
68 
69      initial_addr_space.top_level.pd = EARLY_PTR_TO_PHYS_ADDR(page_directory);
70      initial_addr_space.cr3          = EARLY_VIRT_TO_PHYS((uintptr_t)page_directory);
71 
72      return &initial_addr_space;
73 }
```

Here is the call graph for this function:



4.47.1.6 void vm_x86_destroy_addr_space (addr_space_t ∗ *addr_space*)

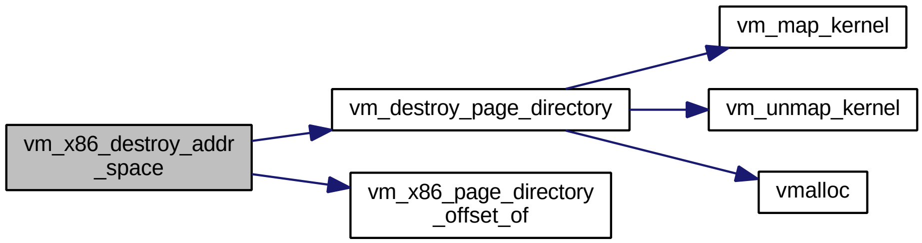
Definition at line 75 of file vm_x86.c.

References KLIMIT, addr_space_t::pd, addr_space_t::top_level, vm_destroy_page_directory(), and vm_x86_page_directory_offset_of().

Referenced by vm_destroy_addr_space().

```
75                                                            {
76      vm_destroy_page_directory(
77              addr_space->top_level.pd,
78              /* Free page tables for addresses 0..KLIMIT, be careful not to free
79               * the kernel page tables starting at KLIMIT. */
80              0,
81              vm_x86_page_directory_offset_of((addr_t)KLIMIT));
82 }
```

Here is the call graph for this function:



4.47.1.7 int vm_x86_get_pte_flags (const pte_t * *pte*)

Definition at line 125 of file vm_x86.c.

References pte_t::entry, and PAGE_MASK.

```
125                                        {
126     return pte->entry & PAGE_MASK;
127 }
```

4.47.1.8 uint32_t vm_x86_get_pte_paddr (const pte_t * *pte*)

Definition at line 129 of file vm_x86.c.

References pte_t::entry, and PAGE_MASK.

```
129                                             {
130     return pte->entry & ~PAGE_MASK;
131 }
```

4.47.1.9 pte_t* vm_x86_get_pte_with_offset (pte_t * *pte,* unsigned int *offset*)

Definition at line 113 of file vm_x86.c.

```
113                                                                {
114     return &pte[offset];
115 }
```

4.47.1.10 pte_t* vm_x86_lookup_page_directory (addr_space_t * *addr_space*)

Lookup and map the page directory for a specified address and address space.

This is the implementation for standard 32-bit (i.e. non-PAE) paging. This means that there is only one preallocated page directory, so the addr and create_as_needed arguments are both irrelevant.

Important note: it is the caller's responsibility to unmap and free the returned page directory when it is done with it.

Parameters

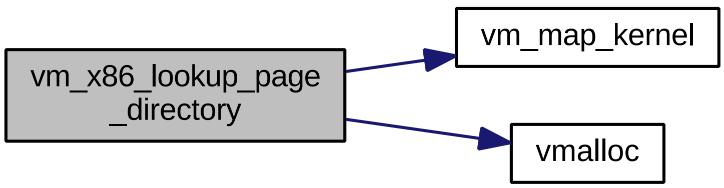
addr_space	address space in which the address is looked up.
addr	address to look up
create_as_need	Whether a page table is allocated if it does not exist

Definition at line 106 of file vm_x86.c.

References addr_space_t::pd, addr_space_t::top_level, VM_FLAG_READ_WRITE, vm_map_kernel(), and vmalloc().

```
106                                                               {
107     pte_t *page_directory = (pte_t *)vmalloc();
108     vm_map_kernel((addr_t)page_directory, addr_space->top_level.pd,
      VM_FLAG_READ_WRITE);
109
110     return page_directory;
111 }
```

Here is the call graph for this function:



4.47.1.11 unsigned int vm_x86_page_directory_offset_of (addr_t *addr*)

Definition at line 88 of file vm_x86.c.

References PAGE_DIRECTORY_OFFSET_OF.

Referenced by vm_x86_create_addr_space(), vm_x86_create_initial_addr_space(), and vm_x86_destroy_addr_-space().

```
88                                                           {
89      return PAGE_DIRECTORY_OFFSET_OF(addr);
90 }
```

4.47.1.12 unsigned int vm_x86_page_table_offset_of (addr_t *addr*)

Definition at line 84 of file vm_x86.c.

References PAGE_TABLE_OFFSET_OF.

```
84                                                       {
85      return PAGE_TABLE_OFFSET_OF(addr);
86 }
```

4.47.1.13 void vm_x86_set_pte (pte_t * *pte*, uint32_t *paddr*, int *flags*)

Definition at line 117 of file vm_x86.c.

References pte_t::entry.

```
                                                                    {
    pte->entry = paddr | flags;
}
```

4.47.1.14 void vm_x86_set_pte_flags (**pte_t** ∗ *pte,* int *flags*)

Definition at line 121 of file vm_x86.c.

References pte_t::entry, and PAGE_MASK.

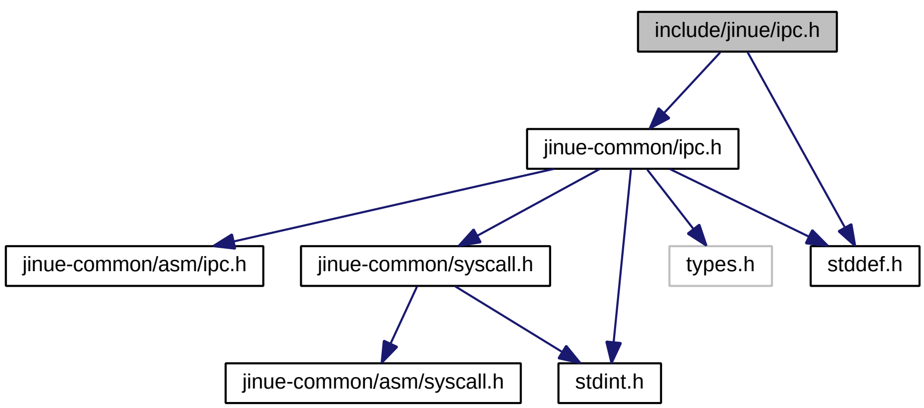
```
121                                                     {
122     pte->entry = (pte->entry & ~PAGE_MASK) | flags;
123 }
```

4.48 include/ipc.h File Reference

4.49 include/jinue/ipc.h File Reference

```
#include <jinue-common/ipc.h>
#include <stddef.h>
```

Include dependency graph for ipc.h:



Data Structures

- struct **jinue_message_t**
- struct **jinue_reply_t**

Functions

- int **jinue_send** (int function, int fd, char ∗buffer, **size_t** buffer_size, **size_t** data_size, unsigned int n_desc, int ∗perrno)
- int **jinue_receive** (int fd, char ∗buffer, **size_t** buffer_size, **jinue_message_t** ∗message, int ∗perrno)
- int **jinue_reply** (char ∗buffer, **size_t** buffer_size, **size_t** data_size, unsigned int n_desc, int ∗perrno)
- int **jinue_create_ipc** (int flags, int ∗perrno)

4.49.1 Function Documentation

4.49.1.1 int jinue_create_ipc (int *flags,* int ∗ *perrno*)

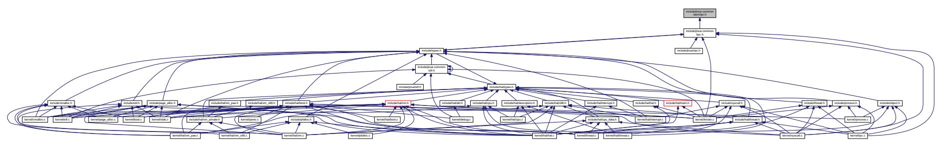
4.49.1.2 int jinue_receive (int *fd,* char ∗ *buffer,* size_t *buffer_size,* jinue_message_t ∗ *message,* int ∗ *perrno*)

4.49.1.3 int jinue_reply (char ∗ *buffer,* size_t *buffer_size,* size_t *data_size,* unsigned int *n_desc,* int ∗ *perrno*)

4.49.1.4 int jinue_send (int *function,* int *fd,* char ∗ *buffer,* size_t *buffer_size,* size_t *data_size,* unsigned int *n_desc,* int ∗ *perrno*)

4.50 include/jinue-common/asm/ipc.h File Reference

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



Macros

- #define **JINUE_SEND_SIZE_BITS** 12

    number of bits reserved for the message buffer size and data size fields
- #define **JINUE_SEND_N_DESC_BITS** 8

    number of bits reserved for the number of message descriptors
- #define **JINUE_SEND_MAX_SIZE** (1 << (**JINUE_SEND_SIZE_BITS** - 1))

    maximum size of a message buffer and of the data inside that buffer
- #define **JINUE_SEND_MAX_N_DESC** ((1 << **JINUE_SEND_N_DESC_BITS**) - 1)

    maximum number of descriptors inside a message
- #define **JINUE_SEND_SIZE_MASK** ((1 << **JINUE_SEND_SIZE_BITS**) - 1)

    mask to extract the message buffer or data size fields
- #define **JINUE_SEND_N_DESC_MASK JINUE_SEND_MAX_N_DESC**

    mask to extract the number of descriptors inside a message
- #define **JINUE_SEND_BUFFER_SIZE_OFFSET** (**JINUE_SEND_N_DESC_BITS** + **JINUE_SEND_SIZE_BITS**)

    offset of buffer size within arg3
- #define **JINUE_SEND_DATA_SIZE_OFFSET JINUE_SEND_N_DESC_BITS**

    offset of data size within arg3
- #define **JINUE_SEND_N_DESC_OFFSET** 0

    offset of number of descriptors within arg3
- #define **JINUE_ARGS_PACK_BUFFER_SIZE**(s) ((s) << **JINUE_SEND_BUFFER_SIZE_OFFSET**)
- #define **JINUE_ARGS_PACK_DATA_SIZE**(s) ((s) << **JINUE_SEND_DATA_SIZE_OFFSET**)
- #define **JINUE_ARGS_PACK_N_DESC**(n) ((n) << **JINUE_SEND_N_DESC_OFFSET**)

4.50.1 Macro Definition Documentation

4.50.1.1 #define JINUE_ARGS_PACK_BUFFER_SIZE(*s*) ((s) << JINUE_SEND_BUFFER_SIZE_OFFSET)

Definition at line 68 of file ipc.h.

4.50.1.2 #define JINUE_ARGS_PACK_DATA_SIZE(*s*) ((s) << JINUE_SEND_DATA_SIZE_OFFSET)

Definition at line 70 of file ipc.h.

4.50.1.3 #define JINUE_ARGS_PACK_N_DESC(*n*) ((n) << JINUE_SEND_N_DESC_OFFSET)

Definition at line 72 of file ipc.h.

4.50.1.4 #define JINUE_SEND_BUFFER_SIZE_OFFSET (JINUE_SEND_N_DESC_BITS + JINUE_SEND_SIZE_BITS)

offset of buffer size within arg3

Definition at line 59 of file ipc.h.

Referenced by ipc_reply().

4.50.1.5 #define JINUE_SEND_DATA_SIZE_OFFSET JINUE_SEND_N_DESC_BITS

offset of data size within arg3

Definition at line 62 of file ipc.h.

4.50.1.6 #define JINUE_SEND_MAX_N_DESC ((1 << JINUE_SEND_N_DESC_BITS) - 1)

maximum number of descriptors inside a message

Definition at line 50 of file ipc.h.

Referenced by ipc_reply(), and ipc_send().

4.50.1.7 #define JINUE_SEND_MAX_SIZE (1 << (JINUE_SEND_SIZE_BITS - 1))

maximum size of a message buffer and of the data inside that buffer

Definition at line 47 of file ipc.h.

Referenced by ipc_reply(), and ipc_send().

4.50.1.8 #define JINUE_SEND_N_DESC_BITS 8

number of bits reserved for the number of message descriptors

Definition at line 44 of file ipc.h.

4.50.1.9 #define JINUE_SEND_N_DESC_MASK JINUE_SEND_MAX_N_DESC

mask to extract the number of descriptors inside a message

Definition at line 56 of file ipc.h.

4.50.1.10 #define JINUE_SEND_N_DESC_OFFSET 0

offset of number of descriptors within arg3

Definition at line 65 of file ipc.h.

4.50.1.11 #define JINUE_SEND_SIZE_BITS 12

number of bits reserved for the message buffer size and data size fields

Definition at line 41 of file ipc.h.

4.50.1.12 #define JINUE_SEND_SIZE_MASK ((1 << JINUE_SEND_SIZE_BITS) - 1)

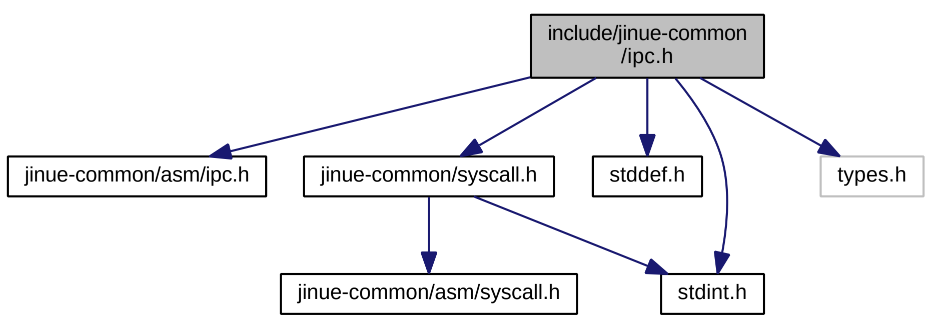
mask to extract the message buffer or data size fields

Definition at line 53 of file ipc.h.

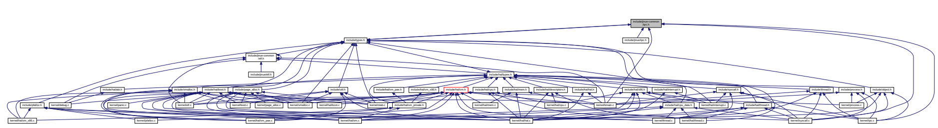
Referenced by ipc_reply().

4.51 include/jinue-common/ipc.h File Reference

```
#include <jinue-common/asm/ipc.h>
#include <jinue-common/syscall.h>
#include <stddef.h>
#include <stdint.h>
#include <types.h>
```
Include dependency graph for ipc.h:



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



Macros

- #define **IPC_FLAG_NONE** 0

- #define **IPC_FLAG_SYSTEM** (1<<8)
- #define **JINUE_IPC_NONE** 0
- #define **JINUE_IPC_SYSTEM** (1<<0)
- #define **JINUE_IPC_PROC** (1<<1)

Typedefs

- typedef int **jinue_ipc_descriptor_t**

Functions

- void **ipc_boot_init** (**boot_alloc_t** *boot_alloc)
- **ipc_t** * **ipc_object_create** (int flags)
- **ipc_t** * **ipc_get_proc_object** (void)
- void **ipc_send** (**jinue_syscall_args_t** *args)
- void **ipc_receive** (**jinue_syscall_args_t** *args)
- void **ipc_reply** (**jinue_syscall_args_t** *args)

4.51.1 Macro Definition Documentation

4.51.1.1 #define IPC_FLAG_NONE 0

Definition at line 41 of file ipc.h.

Referenced by dispatch_syscall().

4.51.1.2 #define IPC_FLAG_SYSTEM (1<<8)

Definition at line 43 of file ipc.h.

Referenced by dispatch_syscall().

4.51.1.3 #define JINUE_IPC_NONE 0

Definition at line 41 of file ipc.h.

4.51.1.4 #define JINUE_IPC_PROC (1<<1)

Definition at line 45 of file ipc.h.

Referenced by dispatch_syscall().

4.51.1.5 #define JINUE_IPC_SYSTEM (1<<0)

Definition at line 43 of file ipc.h.

Referenced by dispatch_syscall().

4.51.2 Typedef Documentation

4.51.2.1 typedef int jinue_ipc_descriptor_t

Definition at line 48 of file ipc.h.

4.51.3 Function Documentation

4.51.3.1 void ipc_boot_init (boot_alloc_t * *boot_alloc*)

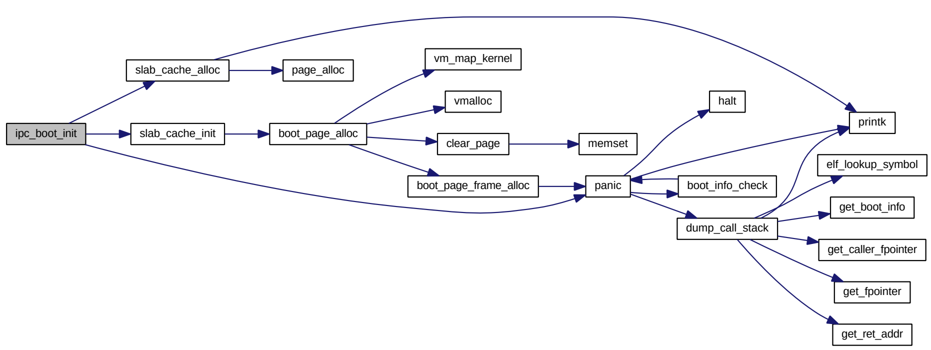
Definition at line 58 of file ipc.c.

References NULL, panic(), slab_cache_alloc(), slab_cache_init(), and SLAB_DEFAULTS.

Referenced by kmain().

```
58                                              {
59      slab_cache_init(
60              &ipc_object_cache,
61              "ipc_object_cache",
62              sizeof(ipc_t),
63              0,
64              ipc_object_ctor,
65              NULL,
66              SLAB_DEFAULTS,
67              boot_alloc);
68 
69      proc_ipc = slab_cache_alloc(&ipc_object_cache);
70 
71      if(proc_ipc == NULL) {
72          panic("Cannot create process manager IPC object.");
73      }
74 }
```

Here is the call graph for this function:



4.51.3.2 ipc_t* ipc_get_proc_object (void)

Definition at line 86 of file ipc.c.

Referenced by dispatch_syscall().

```
86                                {
87      return proc_ipc;
88 }
```

4.51.3.3 ipc_t∗ ipc_object_create (int *flags*)

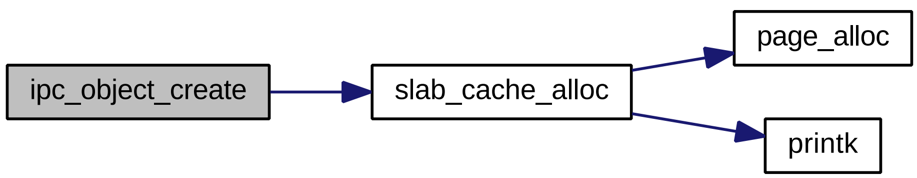
Definition at line 76 of file ipc.c.

References object_header_t::flags, ipc_t::header, NULL, and slab_cache_alloc().

Referenced by dispatch_syscall().

```
76                                 {
77     ipc_t *ipc = slab_cache_alloc(&ipc_object_cache);
78 
79     if(ipc != NULL) {
80         ipc->header.flags = flags;
81     }
82 
83     return ipc;
84 }
```

Here is the call graph for this function:



4.51.3.4 void ipc_receive (jinue_syscall_args_t ∗ *args*)

Definition at line 205 of file ipc.c.

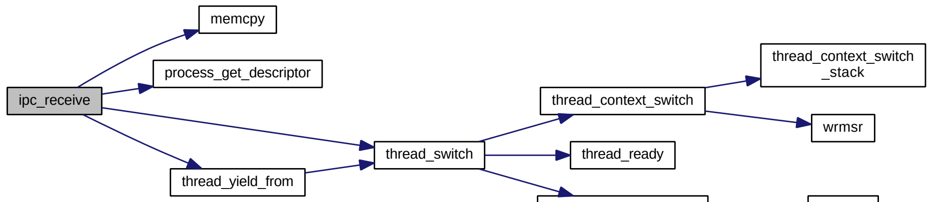
References jinue_syscall_args_t::arg0, jinue_syscall_args_t::arg1, jinue_syscall_args_t::arg2, jinue_syscall_args_t::arg3, message_info_t::data_size, object_header_t::flags, thread_t::header, JINUE_E2BIG, JINUE_EBADF, JINUE_EINVAL, JINUE_EIO, JINUE_EPERM, jinue_node_entry, memcpy(), thread_t::message_args, thread_t::message_buffer, thread_t::message_info, NULL, OBJECT_REF_FLAG_CLOSED, OBJECT_TYPE_IPC, thread_t::process, process_get_descriptor(), ipc_t::recv_list, ipc_t::send_list, thread_t::sender, thread_t::thread_list, thread_switch(), thread_yield_from(), message_info_t::total_size, and object_header_t::type.

Referenced by dispatch_syscall().

```
205                                          {
206     thread_t *thread = get_current_thread();
207 
208     int fd = (int)args->arg1;
209 
210     object_ref_t *ref = process_get_descriptor(thread->process, fd);
211 
212     if(! object_ref_is_valid(ref)) {
213         syscall_args_set_error(args, JINUE_EBADF);
214         return;
215     }
216 
217     if(object_ref_is_closed(ref)) {
218         syscall_args_set_error(args, JINUE_EIO);
219         return;
220     }
221 
222     if(! object_ref_is_owner(ref)) {
223         syscall_args_set_error(args, JINUE_EPERM);
224         return;
225     }
226 
227     object_header_t *header = ref->object;
228 
```

```
    if(object_is_destroyed(header)) {
        ref->flags |= OBJECT_REF_FLAG_CLOSED;
        object_subref(header);

        syscall_args_set_error(args, JINUE_EIO);
        return;
    }

    if(header->type != OBJECT_TYPE_IPC) {
        syscall_args_set_error(args, JINUE_EBADF);
        return;
    }

    ipc_t *ipc = (ipc_t *)header;

    char *user_ptr = (char *)args->arg2;
    size_t buffer_size = jinue_args_get_buffer_size(args);

    if(! user_buffer_check(user_ptr, buffer_size)) {
        syscall_args_set_error(args, JINUE_EINVAL);
        return;
    }

    thread_t *send_thread = jinue_node_entry(
        jinue_list_dequeue(&ipc->send_list),
        thread_t,
        thread_list);

    if(send_thread == NULL) {
        /* No thread is waiting to send a message, so we must wait on the receive
         * list. */
        jinue_list_enqueue(&ipc->recv_list, &thread->thread_list);

        thread_yield_from(
                thread,
                true,       /* make thread block */
                false);     /* don't destroy */

        /* set by sending thread */
        send_thread = thread->sender;
    }
    else {
        object_addref(&send_thread->header);
        thread->sender = send_thread;
    }

    if(send_thread->message_info.total_size > buffer_size) {
        /* message is too big for receive buffer */
        object_subref(&send_thread->header);
        thread->sender = NULL;

        syscall_args_set_error(send_thread->message_args, JINUE_E2BIG);
        syscall_args_set_error(args, JINUE_E2BIG);

        /* switch back to sender thread to return from call immediately */
        thread_switch(
                thread,
                send_thread,
                false,      /* don't block (put this thread back in ready queue) */
                false);     /* don't destroy */

        return;
    }

    memcpy(
        user_ptr,
        send_thread->message_buffer,
        send_thread->message_info.data_size);

    args->arg0 = send_thread->message_args->arg0;
    args->arg1 = ref->cookie;
    /* argument 2 is left intact (buffer pointer) */
    args->arg3 = send_thread->message_args->arg3;
}
```

Here is the call graph for this function:



4.51.3.5 void ipc_reply (jinue_syscall_args_t * *args*)

TODO is there a better error number for this situation?

TODO remove this check when descriptor passing is implemented

TODO copy descriptors

TODO set return value and error number

Definition at line 304 of file ipc.c.

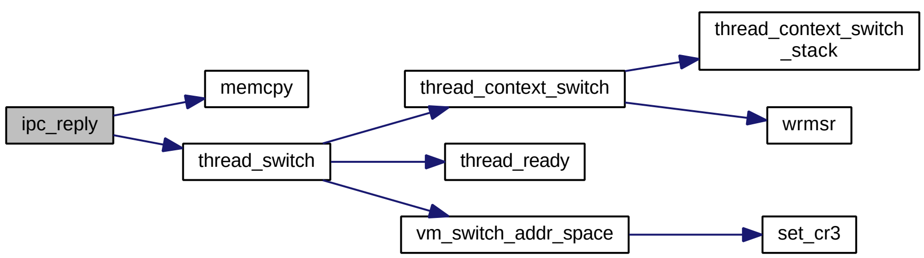
References jinue_syscall_args_t::arg2, jinue_syscall_args_t::arg3, message_info_t::buffer_size, message_info_t::data-_size, message_info_t::desc_n, thread_t::header, JINUE_EINVAL, JINUE_ENOSYS, JINUE_SEND_BUFFER_SIZE_-OFFSET, JINUE_SEND_MAX_N_DESC, JINUE_SEND_MAX_SIZE, JINUE_SEND_SIZE_MASK, memcpy(), thread_-t::message_args, thread_t::message_buffer, thread_t::message_info, NULL, thread_t::sender, and thread_switch().

Referenced by dispatch_syscall().

```
304                                             {
305     thread_t *thread        = get_current_thread();
306     thread_t *send_thread   = thread->sender;
307 
308     if(send_thread == NULL) {
310         syscall_args_set_error(args, JINUE_EINVAL);
311         return;
312     }
313 
314     size_t buffer_size   = jinue_args_get_buffer_size(args);
315     size_t data_size     = jinue_args_get_data_size(args);
316     size_t desc_n        = jinue_args_get_n_desc(args);
317     size_t total_size    =
318             data_size +
319             desc_n * sizeof(jinue_ipc_descriptor_t);
320 
321     if(buffer_size > JINUE_SEND_MAX_SIZE) {
322         syscall_args_set_error(args, JINUE_EINVAL);
323         return;
324     }
325 
326     if(total_size > buffer_size) {
327         syscall_args_set_error(args, JINUE_EINVAL);
328         return;
329     }
330 
331     if(desc_n > JINUE_SEND_MAX_N_DESC) {
332         syscall_args_set_error(args, JINUE_EINVAL);
333         return;
334     }
335 
336     /* the reply must fit in the sender's buffer */
337     if(total_size > send_thread->message_info.buffer_size) {
338         syscall_args_set_error(args, JINUE_EINVAL);
339         return;
340     }
341 
343     if(desc_n > 0) {
```

```
        syscall_args_set_error(args, JINUE_ENOSYS);
        return;
    }

    const char *user_ptr = (const char *)args->arg2;

    if(! user_buffer_check(user_ptr, buffer_size)) {
        syscall_args_set_error(args, JINUE_EINVAL);
        return;
    }

    memcpy(&send_thread->message_buffer, user_ptr, data_size);

    syscall_args_set_return(send_thread->message_args, 0);
    send_thread->message_args->arg3 =
            args->arg3 & ~(JINUE_SEND_SIZE_MASK << JINUE_SEND_BUFFER_SIZE_OFFSET);

    send_thread->message_info.data_size = data_size;
    send_thread->message_info.desc_n    = desc_n;

    object_subref(&send_thread->header);
    thread->sender = NULL;

    syscall_args_set_return(args, 0);

    /* switch back to sender thread to return from call immediately */
    thread_switch(
            thread,
            send_thread,
            false,      /* don't block (put this thread back in ready queue) */
            false);     /* don't destroy */
}
```

Here is the call graph for this function:



4.51.3.6 void ipc_send (jinue_syscall_args_t * *args*)

TODO remove this check when descriptor passing is implemented

TODO copy descriptors

TODO copy descriptors

Definition at line 90 of file ipc.c.

References jinue_syscall_args_t::arg0, jinue_syscall_args_t::arg1, jinue_syscall_args_t::arg2, message_info_t::buffer_size, message_info_t::cookie, message_info_t::data_size, message_info_t::desc_n, object_header_t::flags, message_info_t::function, thread_t::header, JINUE_EBADF, JINUE_EINVAL, JINUE_EIO, JINUE_ENOSYS, jinue_node_entry, JINUE_SEND_MAX_N_DESC, JINUE_SEND_MAX_SIZE, memcpy(), thread_t::message_args, thread_t::message_buffer, thread_t::message_info, NULL, OBJECT_REF_FLAG_CLOSED, OBJECT_TYPE_IPC, thread_t::process, process_get_descriptor(), ipc_t::recv_list, ipc_t::send_list, thread_t::sender, thread_t::thread_list, thread_switch(), thread_yield_from(), message_info_t::total_size, and object_header_t::type.

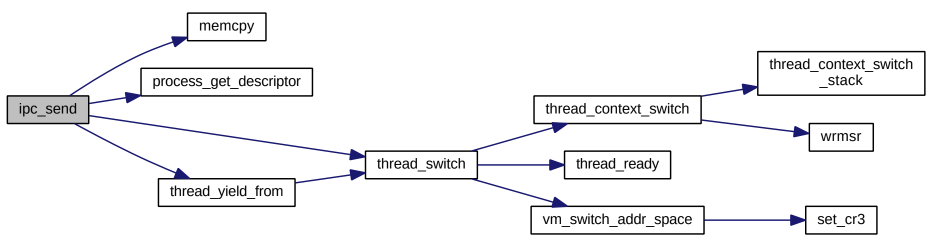
Referenced by dispatch_syscall().

```
90                                                      {
```

```
    thread_t *thread = get_current_thread();

    message_info_t *message_info = &thread->message_info;

    message_info->function      = args->arg0;
    message_info->buffer_size   = jinue_args_get_buffer_size(args);
    message_info->data_size     = jinue_args_get_data_size(args);
    message_info->desc_n        = jinue_args_get_n_desc(args);
    message_info->total_size    =
             message_info->data_size +
             message_info->desc_n * sizeof(jinue_ipc_descriptor_t);

    if(message_info->buffer_size > JINUE_SEND_MAX_SIZE) {
        syscall_args_set_error(args, JINUE_EINVAL);
        return;
    }

    if(message_info->total_size > message_info->buffer_size) {
        syscall_args_set_error(args, JINUE_EINVAL);
        return;
    }

    if(message_info->desc_n > JINUE_SEND_MAX_N_DESC) {
        syscall_args_set_error(args, JINUE_EINVAL);
        return;
    }

    if(message_info->desc_n > 0) {
        syscall_args_set_error(args, JINUE_ENOSYS);
        return;
    }

    int fd = (int)args->arg1;

    object_ref_t *ref = process_get_descriptor(thread->process, fd);

    if(! object_ref_is_valid(ref)) {
        syscall_args_set_error(args, JINUE_EBADF);
        return;
    }

    if(object_ref_is_closed(ref)) {
        syscall_args_set_error(args, JINUE_EIO);
        return;
    }

    message_info->cookie = ref->cookie;

    object_header_t *header = ref->object;

    if(object_is_destroyed(header)) {
        ref->flags |= OBJECT_REF_FLAG_CLOSED;
        object_subref(header);

        syscall_args_set_error(args, JINUE_EIO);
        return;
    }

    if(header->type != OBJECT_TYPE_IPC) {
        syscall_args_set_error(args, JINUE_EBADF);
        return;
    }

    ipc_t *ipc = (ipc_t *)header;

    char *user_ptr = (char *)args->arg2;

    if(! user_buffer_check(user_ptr, message_info->buffer_size)) {
        syscall_args_set_error(args, JINUE_EINVAL);
        return;
    }

    memcpy(&thread->message_buffer, user_ptr, message_info->data_size);

    /* return values are set by ipc_reply() (or by ipc_receive() if the call
     * fails because the message is too big for the receiver's buffer) */
    thread->message_args = args;

    thread_t *recv_thread = jinue_node_entry(
            jinue_list_dequeue(&ipc->recv_list),
            thread_t,
```

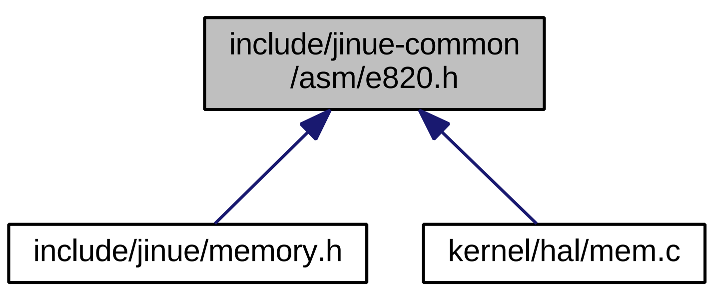
```
            thread_list);

    if(recv_thread == NULL) {
        /* No thread is waiting to receive this message, so we must wait on the
         * sender list. */
        jinue_list_enqueue(&ipc->send_list, &thread->thread_list);

        thread_yield_from(
                thread,
                true,       /* make thread block */
                false);     /* don't destroy */
    }
    else {
        object_addref(&thread->header);
        recv_thread->sender = thread;

        /* switch to receiver thread, which will resume inside syscall_receive() */
        thread_switch(
                thread,
                recv_thread,
                true,       /* block sender thread */
                false);     /* don't destroy sender */
    }

    /* copy reply to user space buffer */
    memcpy(user_ptr, &thread->message_buffer, message_info->data_size);

}
```

Here is the call graph for this function:



4.52 include/jinue-common/asm/e820.h File Reference

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



Macros

- #define **E820_RAM** 1
- #define **E820_RESERVED** 2
- #define **E820_ACPI** 3

- #define **E820_SMAP** 0x534d4150

4.52.1 Macro Definition Documentation

4.52.1.1 #define E820_ACPI 3

Definition at line 39 of file e820.h.

4.52.1.2 #define E820_RAM 1

Definition at line 35 of file e820.h.

Referenced by mem_check_memory().

4.52.1.3 #define E820_RESERVED 2

Definition at line 37 of file e820.h.

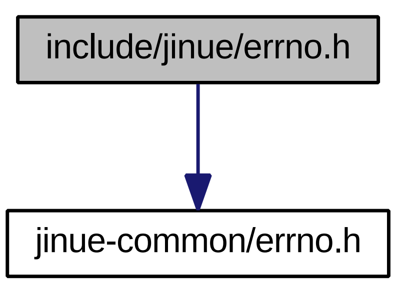
4.52.1.4 #define E820_SMAP 0x534d4150

Definition at line 41 of file e820.h.

4.53 include/jinue/errno.h File Reference

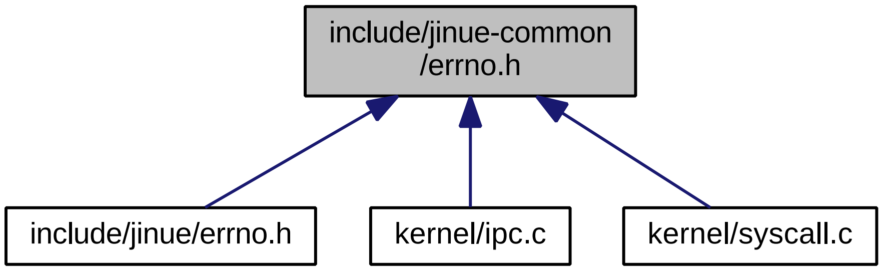
```
#include <jinue-common/errno.h>
```

Include dependency graph for errno.h:



4.54 include/jinue-common/errno.h File Reference

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



Macros

- #define **JINUE_EMORE** 1
- #define **JINUE_ENOMEM** 2
- #define **JINUE_ENOSYS** 3
- #define **JINUE_EINVAL** 4
- #define **JINUE_EAGAIN** 5
- #define **JINUE_EBADF** 6
- #define **JINUE_EIO** 7
- #define **JINUE_EPERM** 8
- #define **JINUE_E2BIG** 9

4.54.1 Macro Definition Documentation

4.54.1.1 #define JINUE_E2BIG 9

Definition at line 51 of file errno.h.

Referenced by ipc_receive().

4.54.1.2 #define JINUE_EAGAIN 5

Definition at line 43 of file errno.h.

Referenced by dispatch_syscall().

4.54.1.3 #define JINUE_EBADF 6

Definition at line 45 of file errno.h.

Referenced by ipc_receive(), and ipc_send().

4.54.1.4 #define JINUE_EINVAL 4

Definition at line 41 of file errno.h.

Referenced by dispatch_syscall(), ipc_receive(), ipc_reply(), and ipc_send().

4.54.1.5 #define JINUE_EIO 7

Definition at line 47 of file errno.h.

Referenced by ipc_receive(), and ipc_send().

4.54.1.6 #define JINUE_EMORE 1

Definition at line 35 of file errno.h.

4.54.1.7 #define JINUE_ENOMEM 2

Definition at line 37 of file errno.h.

4.54.1.8 #define JINUE_ENOSYS 3

Definition at line 39 of file errno.h.

Referenced by dispatch_syscall(), ipc_reply(), and ipc_send().

4.54.1.9 #define JINUE_EPERM 8

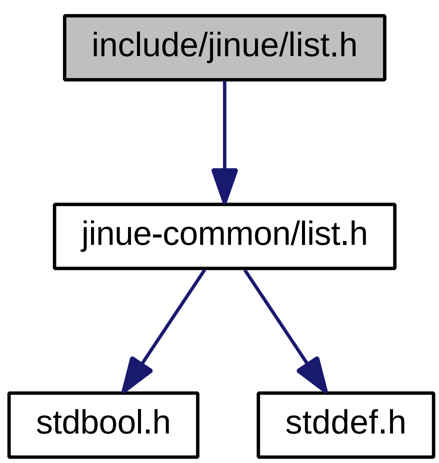
Definition at line 49 of file errno.h.

Referenced by ipc_receive().

4.55 include/jinue/list.h File Reference

```
#include <jinue-common/list.h>
```

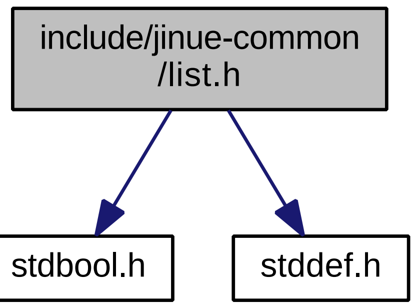
Include dependency graph for list.h:



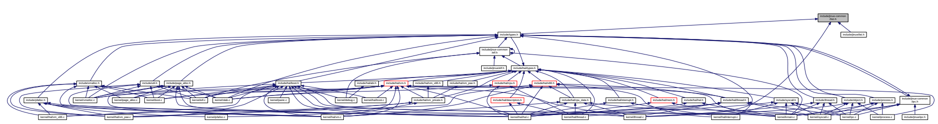
4.56 include/jinue-common/list.h File Reference

```
#include <stdbool.h>
#include <stddef.h>
```

Include dependency graph for list.h:



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



Data Structures

- struct **jinue_node_t**
- struct **jinue_list_t**

Macros

- #define **JINUE_LIST_STATIC** {.head = **NULL**, .tail = **NULL**}
- #define **jinue_list_pop**(l) (jinue_list_dequeue((l)))
- #define **JINUE_OFFSETOF**(type, member) ((**size_t**)(&((type ∗)0)->member))

  TODO move this to a more general-purpose header file.
- #define **jinue_node_entry**(node, type, member) (jinue_node_entry_by_offset(node, **JINUE_OFFSETOF**(type, member)))
- #define **jinue_cursor_entry**(cur, type, member) (jinue_cursor_entry_by_offset(cur, **JINUE_OFFSETOF**(type, member)))

Typedefs

- typedef struct **jinue_node_t** **jinue_node_t**
- typedef **jinue_node_t** ∗∗ **jinue_cursor_t**

4.56.1 Macro Definition Documentation

4.56.1.1 #define jinue_cursor_entry(*cur,* *type,* *member*) (jinue_cursor_entry_by_offset(cur, JINUE_OFFSETOF(type, member)))

Definition at line 158 of file list.h.

4.56.1.2 #define jinue_list_pop(*l*) (jinue_list_dequeue((l)))

Definition at line 121 of file list.h.

4.56.1.3 #define JINUE_LIST_STATIC {.head = NULL, .tail = NULL}

Definition at line 62 of file list.h.

4.56.1.4 #define jinue_node_entry(*node,* *type,* *member*) (jinue_node_entry_by_offset(node, JINUE_OFFSETOF(type, member)))

Definition at line 144 of file list.h.

Referenced by ipc_receive(), and ipc_send().

4.56.1.5 #define JINUE_OFFSETOF(*type,* *member*) ((size_t)(&((type ∗)0)->member))

TODO move this to a more general-purpose header file.

Definition at line 142 of file list.h.

4.56.2 Typedef Documentation

4.56.2.1 typedef jinue_node_t∗∗ jinue_cursor_t

Definition at line 60 of file list.h.

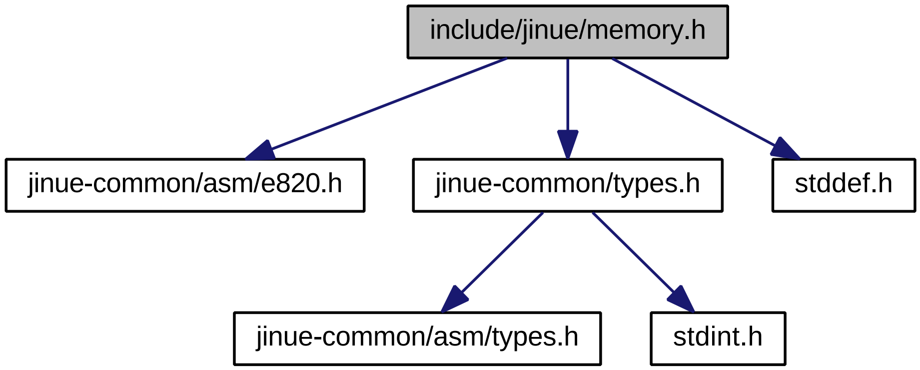
4.56.2.2 typedef struct jinue_node_t jinue_node_t

Definition at line 42 of file list.h.

4.57 include/jinue/memory.h File Reference

```
#include <jinue-common/asm/e820.h>
#include <jinue-common/types.h>
#include <stddef.h>
```

Include dependency graph for memory.h:



Functions

- const char * **jinue_pys_mem_type_description** (**uint32_t** type)
- int **jinue_get_phys_memory** (**jinue_mem_map_t** *buffer, **size_t** buffer_size, int *perrno)

4.57.1 Function Documentation

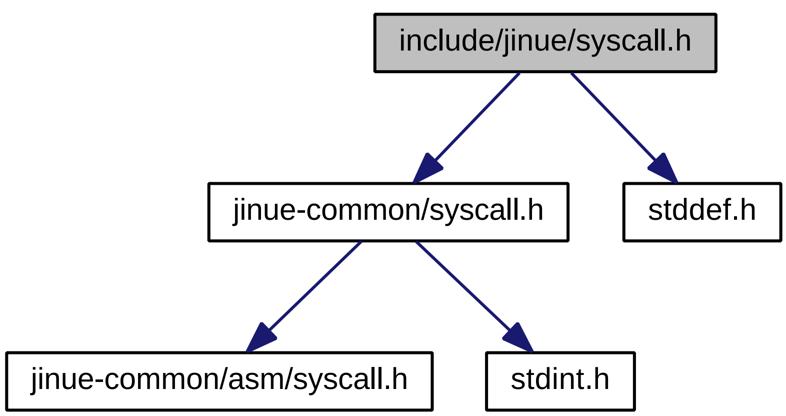
4.57.1.1 int jinue_get_phys_memory (**jinue_mem_map_t** * *buffer,* **size_t** *buffer_size,* int * *perrno*)

4.57.1.2 const char* jinue_pys_mem_type_description (**uint32_t** *type*)

4.58 include/jinue/syscall.h File Reference

```
#include <jinue-common/syscall.h>
#include <stddef.h>
```

Include dependency graph for syscall.h:



Functions

- void **jinue_call_raw** (**jinue_syscall_args_t** *args)

- int **jinue_call** (**jinue_syscall_args_t** ∗args, int ∗perrno)
- void **jinue_get_syscall_implementation** (void)
- const char ∗ **jinue_get_syscall_implementation_name** (void)
- void **jinue_set_thread_local_storage** (void ∗addr, **size_t** size)
- void ∗ **jinue_get_thread_local_storage** (void)
- int **jinue_thread_create** (void(∗entry)(), void ∗stack, int ∗perrno)
- int **jinue_yield** (void)
- void **jinue_thread_exit** (void)

4.58.1 Function Documentation

4.58.1.1 int jinue_call (**jinue_syscall_args_t** ∗ *args*, int ∗ *perrno*)

4.58.1.2 void jinue_call_raw (**jinue_syscall_args_t** ∗ *args*)

4.58.1.3 void jinue_get_syscall_implementation (void)

4.58.1.4 const char∗ jinue_get_syscall_implementation_name (void)

4.58.1.5 void∗ jinue_get_thread_local_storage (void)

4.58.1.6 void jinue_set_thread_local_storage (void ∗ *addr*, **size_t** *size*)

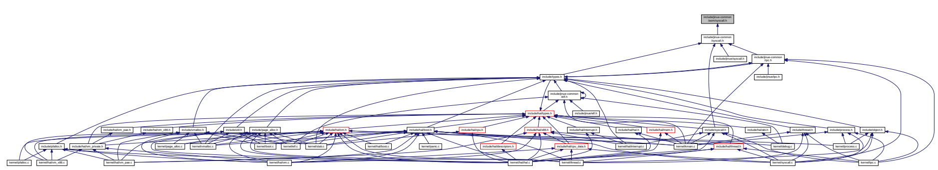
4.58.1.7 int jinue_thread_create (void(∗)() *entry*, void ∗ *stack*, int ∗ *perrno*)

4.58.1.8 void jinue_thread_exit (void)

4.58.1.9 int jinue_yield (void)

4.59 include/jinue-common/asm/syscall.h File Reference

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



Macros

- #define **SYSCALL_IRQ** 0x80

  interrupt vector for system call software interrupt
- #define **SYSCALL_FUNCT_SYSCALL_METHOD** 1

  get best system call implementation number based on CPU features
- #define **SYSCALL_FUNCT_CONSOLE_PUTC** 2

  send a character to in-kernel console driver
- #define **SYSCALL_FUNCT_CONSOLE_PUTS** 3

send a fixed-length string to in-kernel console driver

- #define **SYSCALL_FUNCT_THREAD_CREATE** 4
  create a new thread
- #define **SYSCALL_FUNCT_THREAD_YIELD** 5
  relinquish the CPU and allow the next thread to run
- #define **SYSCALL_FUNCT_SET_THREAD_LOCAL_ADDR** 6
  set address and size of thread local storage for current thread
- #define **SYSCALL_FUNCT_GET_THREAD_LOCAL_ADDR** 7
  get address of thread local storage for current thread
- #define **SYSCALL_FUNCT_GET_PHYS_MEMORY** 8
  get free memory block list for management by process manager
- #define **SYSCALL_FUNCT_CREATE_IPC** 9
  create an IPC object to receive messages
- #define **SYSCALL_FUNCT_RECEIVE** 10
  receive a message on an IPC object
- #define **SYSCALL_FUNCT_REPLY** 11
  reply to current message
- #define **SYSCALL_FUNCT_PROC_BASE** 0x400
  start of function numbers for process manager system calls
- #define **SYSCALL_FUNCT_SYSTEM_BASE** 0x1000
  start of function numbers for system IPC objects
- #define **SYSCALL_FUNCT_USER_BASE** 0x4000
  start of function numbers for user IPC objects
- #define **SYSCALL_METHOD_FAST_INTEL** 0
  Intel's fast system call method (SYSENTER/SYSEXIT)
- #define **SYSCALL_METHOD_FAST_AMD** 1
  AMD's fast system call method (SYSCALL/SYSLEAVE)
- #define **SYSCALL_METHOD_INTR** 2
  slow/safe system call method using interrupts

4.59.1 Macro Definition Documentation

4.59.1.1 #define SYSCALL_FUNCT_CONSOLE_PUTC 2

send a character to in-kernel console driver

Definition at line 42 of file syscall.h.

Referenced by dispatch_syscall().

4.59.1.2 #define SYSCALL_FUNCT_CONSOLE_PUTS 3

send a fixed-length string to in-kernel console driver

Definition at line 45 of file syscall.h.

Referenced by dispatch_syscall().

4.59.1.3 #define SYSCALL_FUNCT_CREATE_IPC 9

create an IPC object to receive messages

Definition at line 63 of file syscall.h.

Referenced by dispatch_syscall().

4.59.1.4 #define SYSCALL_FUNCT_GET_PHYS_MEMORY 8

get free memory block list for management by process manager

Definition at line 60 of file syscall.h.

Referenced by dispatch_syscall().

4.59.1.5 #define SYSCALL_FUNCT_GET_THREAD_LOCAL_ADDR 7

get address of thread local storage for current thread

Definition at line 57 of file syscall.h.

Referenced by dispatch_syscall().

4.59.1.6 #define SYSCALL_FUNCT_PROC_BASE 0x400

start of function numbers for process manager system calls

Definition at line 72 of file syscall.h.

Referenced by dispatch_syscall().

4.59.1.7 #define SYSCALL_FUNCT_RECEIVE 10

receive a message on an IPC object

Definition at line 66 of file syscall.h.

Referenced by dispatch_syscall().

4.59.1.8 #define SYSCALL_FUNCT_REPLY 11

reply to current message

Definition at line 69 of file syscall.h.

Referenced by dispatch_syscall().

4.59.1.9 #define SYSCALL_FUNCT_SET_THREAD_LOCAL_ADDR 6

set address and size of thread local storage for current thread

Definition at line 54 of file syscall.h.

Referenced by dispatch_syscall().

4.59.1.10 #define SYSCALL_FUNCT_SYSCALL_METHOD 1

get best system call implementation number based on CPU features

Definition at line 39 of file syscall.h.

Referenced by dispatch_syscall().

4.59.1.11 #define SYSCALL_FUNCT_SYSTEM_BASE 0x1000

start of function numbers for system IPC objects

Definition at line 75 of file syscall.h.

Referenced by dispatch_syscall().

4.59.1.12 #define SYSCALL_FUNCT_THREAD_CREATE 4

create a new thread

Definition at line 48 of file syscall.h.

Referenced by dispatch_syscall().

4.59.1.13 #define SYSCALL_FUNCT_THREAD_YIELD 5

relinquish the CPU and allow the next thread to run

Definition at line 51 of file syscall.h.

Referenced by dispatch_syscall().

4.59.1.14 #define SYSCALL_FUNCT_USER_BASE 0x4000

start of function numbers for user IPC objects

Definition at line 78 of file syscall.h.

4.59.1.15 #define SYSCALL_IRQ 0x80

interrupt vector for system call software interrupt

Definition at line 36 of file syscall.h.

Referenced by dispatch_interrupt().

4.59.1.16 #define SYSCALL_METHOD_FAST_AMD 1

AMD's fast system call method (SYSCALL/SYSLEAVE)

Definition at line 85 of file syscall.h.

4.59.1.17 #define SYSCALL_METHOD_FAST_INTEL 0

Intel's fast system call method (SYSENTER/SYSEXIT)

Definition at line 82 of file syscall.h.

4.59.1.18 #define SYSCALL_METHOD_INTR 2

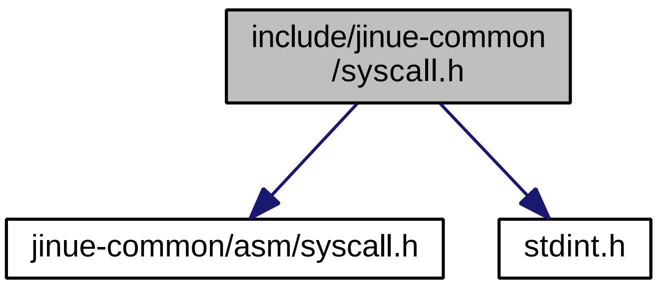
slow/safe system call method using interrupts

Definition at line 88 of file syscall.h.

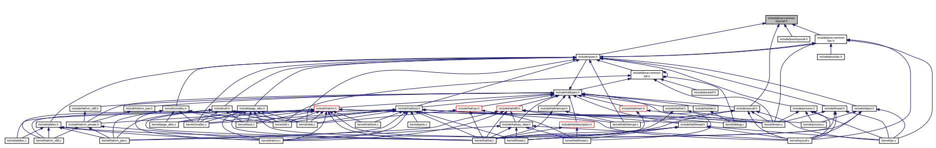
4.60 include/jinue-common/syscall.h File Reference

```
#include <jinue-common/asm/syscall.h>
#include <stdint.h>
```

Include dependency graph for syscall.h:



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



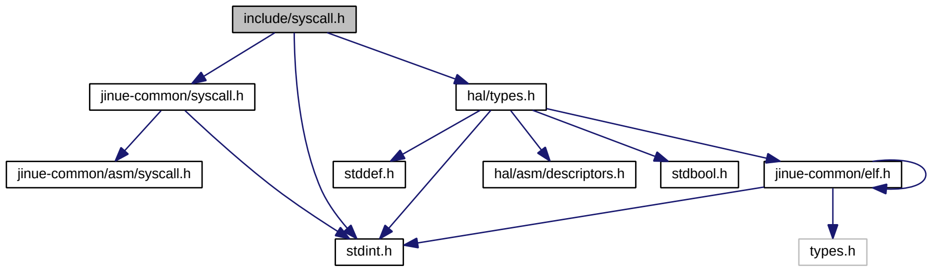
Data Structures

- struct **jinue_syscall_args_t**

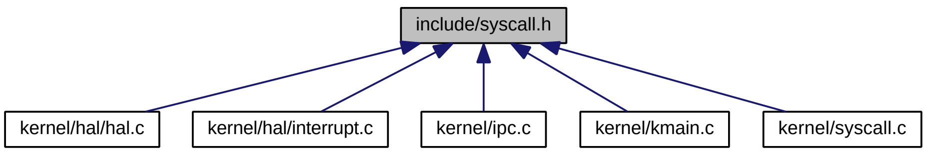
4.61 include/syscall.h File Reference

```
#include <jinue-common/syscall.h>
#include <hal/types.h>
#include <stdint.h>
```

Include dependency graph for syscall.h:



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



Functions

- void **dispatch_syscall** (**trapframe_t** ∗trapframe)

4.61.1 Function Documentation

4.61.1.1 void dispatch_syscall (trapframe_t ∗ *trapframe*)

TODO for check negative values (especially -1)

TODO: permission check

TODO: permission check, sanity check (data size vs buffer size)

TODO: check user pointer

Definition at line 48 of file syscall.c.

References jinue_mem_entry_t::addr, e820_t::addr, jinue_syscall_args_t::arg0, jinue_syscall_args_t::arg1, jinue_syscall_args_t::arg2, jinue_syscall_args_t::arg3, boot_info, CONSOLE_DEFAULT_COLOR, console_printn(), console_putc(), object_ref_t::cookie, boot_info_t::e820_entries, boot_info_t::e820_map, jinue_mem_map_t::entry, object_ref_t::flags, get_boot_info(), ipc_t::header, IPC_FLAG_NONE, IPC_FLAG_SYSTEM, ipc_get_proc_object(), ipc_object_create(), ipc_receive(), ipc_reply(), ipc_send(), JINUE_EAGAIN, JINUE_EINVAL, JINUE_ENOSYS, JINUE_IPC_PROC, JINUE_IPC_SYSTEM, NULL, jinue_mem_map_t::num_entries, object_ref_t::object, OBJECT_REF_FLAG_OWNER, OBJECT_REF_FLAG_VALID, printk(), thread_t::process, process_get_descriptor(), process_unused_descriptor(), jinue_mem_entry_t::size, e820_t::size, SYSCALL_FUNCT_CONSOLE_PUTC, SYSCALL_FUNCT_CONSOLE_PUTS, SYSCALL_FUNCT_CREATE_IPC, SYSCALL_FUNCT_GET_PHYS_MEMORY, SYSCALL_FUNCT_GET_THREAD_LOCAL_ADDR, SYSCALL_FUNCT_PROC_BASE, SYSCALL_FUNCT_RECEIVE, SYSCALL_FUNCT_REPLY, SYSCALL_FUNCT_SET_THREAD_LOCAL_ADDR, SYSCALL_FUNCT_SYSCALL_METHOD, SYSCALL_FUNCT_SYSTEM_BASE, SYSCALL_FUNCT_THREAD_CREATE, SYSCALL_FUNCT_THREAD_YIELD, syscall_method, thread_create(), thread_yield_from(), jinue_mem_entry_t::type, and e820_t::type.

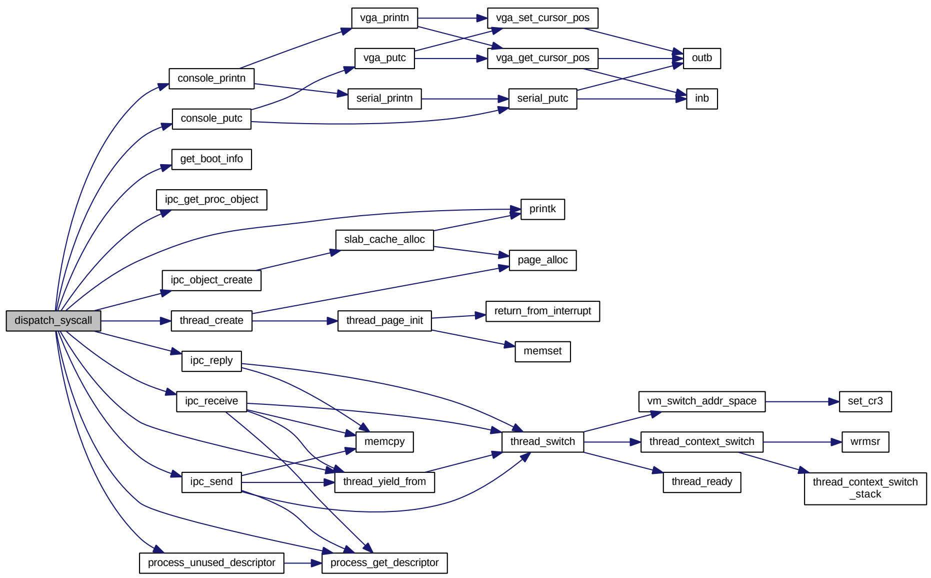
Referenced by dispatch_interrupt().

```
                                          {
    jinue_syscall_args_t *args = (jinue_syscall_args_t *)&trapframe->msg_arg0;

    uintptr_t function_number = args->arg0;

    if(function_number < SYSCALL_FUNCT_PROC_BASE) {
        /* microkernel system calls */
        switch(function_number) {

        case SYSCALL_FUNCT_SYSCALL_METHOD:
            syscall_args_set_return(args, syscall_method);
            break;

        case SYSCALL_FUNCT_CONSOLE_PUTC:
            console_putc(
                    (char)args->arg1,
                    CONSOLE_DEFAULT_COLOR);
            syscall_args_set_return(args, 0);
            break;

        case SYSCALL_FUNCT_CONSOLE_PUTS:
            console_printn(
                    (char *)args->arg2,
                    jinue_args_get_data_size(args),
                    CONSOLE_DEFAULT_COLOR);
            syscall_args_set_return(args, 0);
            break;

        case SYSCALL_FUNCT_THREAD_CREATE:
        {
            thread_t *thread = thread_create(
                    /* TODO use arg1 as an address space reference if specified */
                    get_current_thread()->process,
                    (addr_t)args->arg2,
                    (addr_t)args->arg3);

            if(thread == NULL) {
                syscall_args_set_error(args, JINUE_EAGAIN);
            }
            else {
                syscall_args_set_return(args, 0);
            }
        }
            break;

        case SYSCALL_FUNCT_THREAD_YIELD:
            thread_yield_from(
                    get_current_thread(),
                    false,          /* don't block */
                    args->arg1);    /* destroy (aka. exit) thread if true */
            syscall_args_set_return(args, 0);
            break;

        case SYSCALL_FUNCT_SET_THREAD_LOCAL_ADDR:
            thread_context_set_local_storage(
                    &get_current_thread()->thread_ctx,
                    (addr_t)args->arg1,
                    (size_t)args->arg2);
            syscall_args_set_return(args, 0);
            break;

        case SYSCALL_FUNCT_GET_THREAD_LOCAL_ADDR:
            syscall_args_set_return_ptr(
                    args,
                    thread_context_get_local_storage(
                            &get_current_thread()->thread_ctx));
            break;

        case SYSCALL_FUNCT_GET_PHYS_MEMORY:
        {
            unsigned int idx;

            size_t buffer_size = jinue_args_get_buffer_size(args);
            jinue_mem_map_t *map = (jinue_mem_map_t *)jinue_args_get_buffer_ptr(args);
            const boot_info_t *boot_info = get_boot_info();

            if(buffer_size < sizeof(jinue_mem_map_t) + boot_info->e820_entries * sizeof(jinue_mem_entry_t) ) {
                syscall_args_set_error(args, JINUE_EINVAL);
            }
            else {
```

```
                map->num_entries = boot_info->e820_entries;

                for(idx = 0; idx < map->num_entries; ++idx) {
                    map->entry[idx].addr = boot_info->e820_map[idx].addr;
                    map->entry[idx].size = boot_info->e820_map[idx].size;
                    map->entry[idx].type = boot_info->e820_map[idx].type;
                }

                syscall_args_set_return(args, 0);
            }
        }
            break;

        case SYSCALL_FUNCT_CREATE_IPC:
        {
            ipc_t *ipc;

            thread_t *thread = get_current_thread();

            int fd = process_unused_descriptor(thread->process);

            if(fd < 0) {
                syscall_args_set_error(args, JINUE_EAGAIN);
                break;
            }

            if(args->arg1 & JINUE_IPC_PROC) {
                ipc = ipc_get_proc_object();
            }
            else {
                int flags = IPC_FLAG_NONE;

                if(args->arg1 & JINUE_IPC_SYSTEM) {
                    flags |= IPC_FLAG_SYSTEM;
                }

                ipc = ipc_object_create(flags);

                if(ipc == NULL) {
                    syscall_args_set_error(args, JINUE_EAGAIN);
                    break;
                }
            }

            object_ref_t *ref = process_get_descriptor(thread->process, fd);

            object_addref(&ipc->header);

            ref->object = &ipc->header;
            ref->flags  = OBJECT_REF_FLAG_VALID | OBJECT_REF_FLAG_OWNER;
            ref->cookie = 0;

            syscall_args_set_return(args, fd);

        }
            break;
        case SYSCALL_FUNCT_RECEIVE:
            ipc_receive(args);
            break;

        case SYSCALL_FUNCT_REPLY:
            ipc_reply(args);
            break;

        default:
            printk("SYSCALL: function %u arg1=%u(0x%x) arg2=%u(0x%x) arg3=%u(0x%x)\n",
                function_number,
                args->arg1, args->arg1,
                args->arg2, args->arg2,
                args->arg3, args->arg3 );

            syscall_args_set_error(args, JINUE_ENOSYS);
        }
    }
    else if(function_number < SYSCALL_FUNCT_SYSTEM_BASE) {
        /* process manager system calls */
        printk("PROC SYSCALL: function %u arg1=%u(0x%x) arg2=%u(0x%x) arg3=%u(0x%x)\n",
                function_number,
                args->arg1, args->arg1,
                args->arg2, args->arg2,
                args->arg3, args->arg3 );
```

```
        syscall_args_set_error(args, JINUE_ENOSYS);
    }
    else {
        /* inter-process message */
        ipc_send(args);
    }
}
```

Here is the call graph for this function:



4.62 include/kbd.h File Reference

Functions

- void **any_key** (void)

4.62.1 Function Documentation

4.62.1.1 void any_key (void)

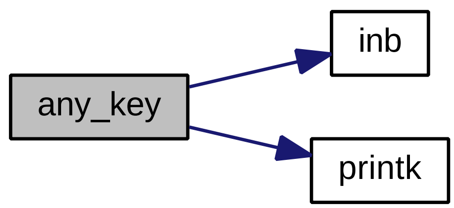
Definition at line 36 of file kbd.c.

References inb(), and printk().

```
36                      {
37     unsigned char buffer;
38     bool ignore;
39 
40     /* prompt */
41     printk("(press enter)");
42 
43     /* wait for key, ignore break codes */
44     ignore = false;
45     while(1) {
46         do {
47             buffer = inb(0x64);
48         } while( (buffer & 1) == 0 );
```

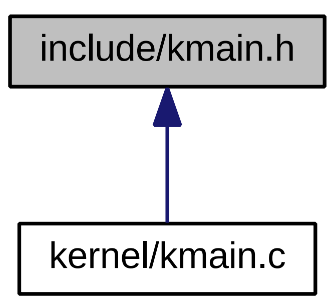
```

        buffer = inb(0x60);

        if(buffer == 0x0e || buffer == 0x0f) {
            ignore = true;
            continue;
        }

        if(ignore) {
            ignore = false;
            continue;
        }

        if(buffer == 0x1c || buffer == 0x5a) {
            break;
        }
    }

    /* advance cursor */
    printk("\n");
}
```

Here is the call graph for this function:



4.63 include/kmain.h File Reference

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



Functions

- void **kmain** (void)

4.63.1 Function Documentation

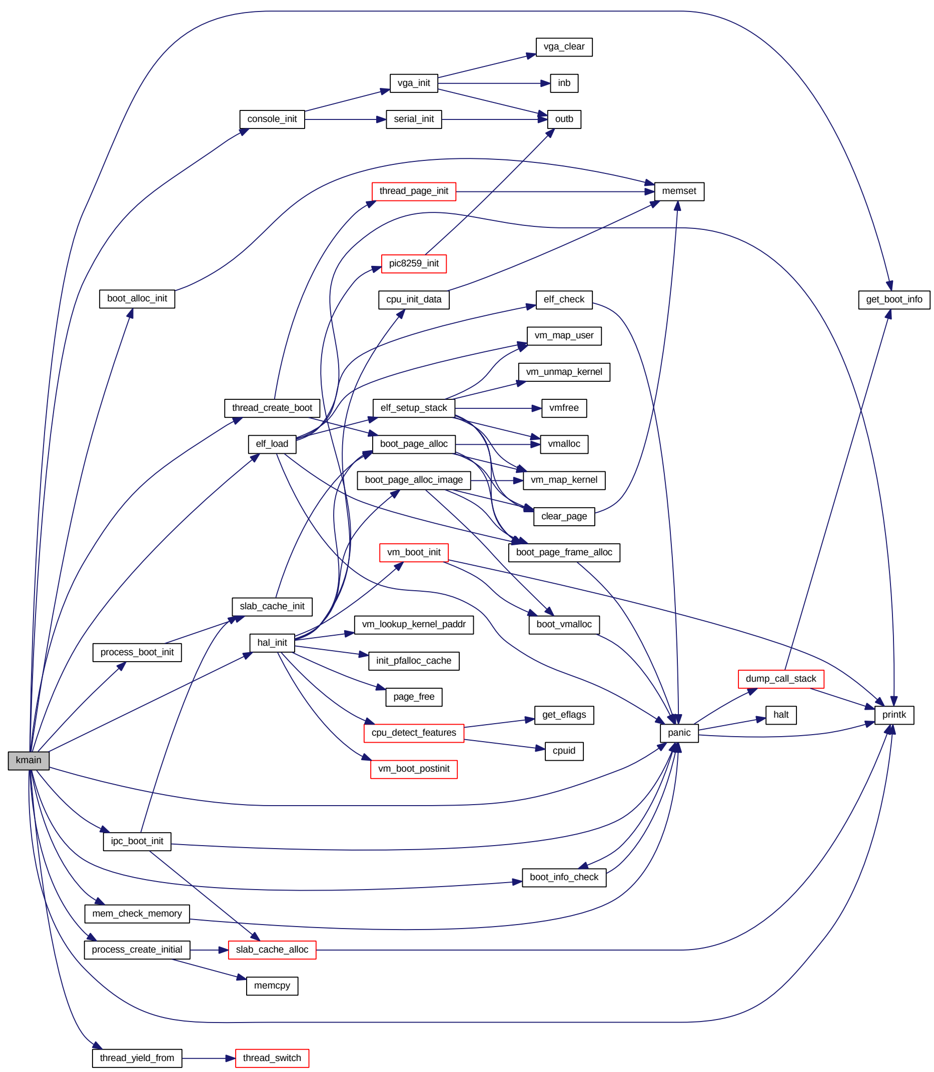
4.63.1.1 void kmain (void)

Definition at line 67 of file kmain.c.

References process_t::addr_space, boot_alloc_init(), boot_info_t::boot_heap, boot_info_check(), BUILD_HOST, BUILD_TIME, boot_info_t::cmdline, console_init(), elf_load(), elf_info_t::entry, get_boot_info(), GIT_REVISION, hal_init(), ipc_boot_init(), boot_info_t::kernel_size, mem_check_memory(), NULL, panic(), printk(), process_boot_init(), process_create_initial(), boot_info_t::ramdisk_size, boot_info_t::ramdisk_start, elf_info_t::stack_addr, thread_create_boot(), thread_yield_from(), and VGA_COLOR_YELLOW.

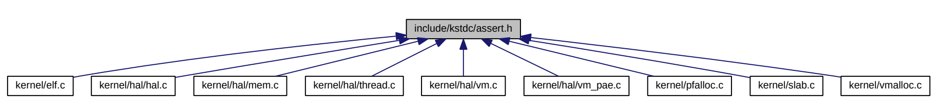
```
{
    elf_info_t elf_info;

    /* initialize console and say hello */
    console_init();

    /* Say hello. */
    printk("Kernel revision " GIT_REVISION " built " BUILD_TIME " on " BUILD_HOST "\n");

    const boot_info_t *boot_info = get_boot_info();
    (void)boot_info_check(true);

    printk("Kernel size is %u bytes.\n", boot_info->kernel_size);

    if(boot_info->ramdisk_start == 0 || boot_info->ramdisk_size == 0) {
        printk("%kWarning: no initial RAM disk loaded.\n", VGA_COLOR_YELLOW);
    }
    else {
        printk("RAM disk with size %u bytes loaded at address %x.\n", boot_info->ramdisk_size, boot_info->ramdisk_start);
    }

    printk("Kernel command line:\n", boot_info->kernel_size);
    printk("    %s\n", boot_info->cmdline);

    /* Initialize the boot allocator. */
    boot_alloc_t boot_alloc;
    boot_alloc_init(&boot_alloc, boot_info->boot_heap);
    mem_check_memory(&boot_alloc, boot_info);

    /* initialize hardware abstraction layer */
    hal_init(&boot_alloc, boot_info);

    /* initialize caches */
    ipc_boot_init(&boot_alloc);
    process_boot_init(&boot_alloc);

    /* create process for process manager */
    process_t *process = process_create_initial();

    if(process == NULL) {
        panic("Could not create initial process.");
    }

    /* load process manager binary */
    Elf32_Ehdr *elf = find_process_manager();
    elf_load(&elf_info, elf, &process->addr_space, &boot_alloc);

    /* create initial thread */
    thread_t *thread = thread_create_boot(
            process,
            elf_info.entry,
            elf_info.stack_addr,
            &boot_alloc);

    if(thread == NULL) {
        panic("Could not create initial thread.");
    }

    /* start process manager
     *
     * We switch from NULL since this is the first thread. */
    thread_yield_from(
            NULL,
            false,      /* don't block */
            false);     /* don't destroy */
                        /* just be nice */

    /* should never happen */
    panic("thread_yield_from() returned in kmain()");
}
```

Here is the call graph for this function:



4.64 include/kstdc/assert.h File Reference

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



Macros

- #define **assert**(expr)

Functions

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

4.64.1 Macro Definition Documentation

4.64.1.1 #define assert(*expr*)

Value:

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

Definition at line 46 of file assert.h.

Referenced by hal_init(), mem_check_memory(), slab_cache_alloc(), slab_cache_init(), thread_context_switch(), vm_-change_flags(), vm_destroy_addr_space(), vm_lookup_kernel_paddr(), and vm_map_early().

4.64.2 Function Documentation

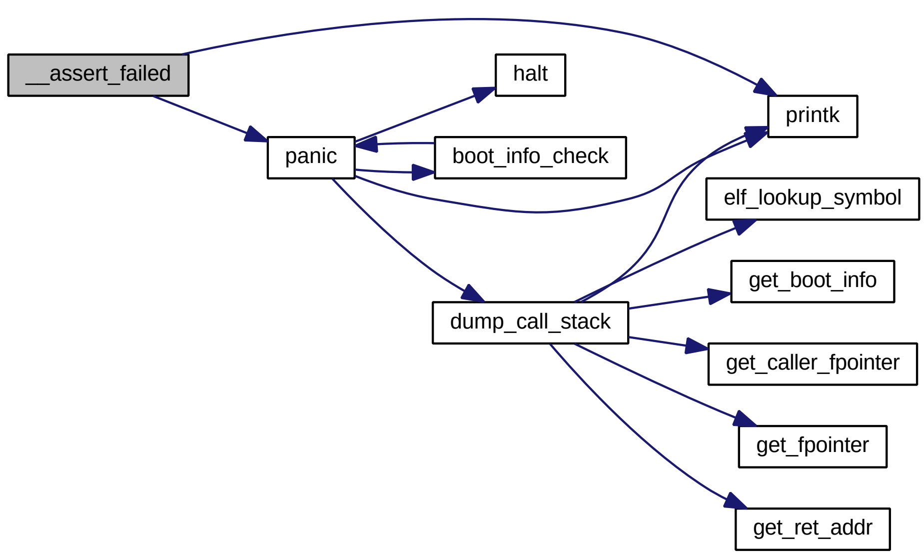
4.64.2.1 void __assert_failed (const char ∗ *expr,* const char ∗ *file,* unsigned int *line,* const char ∗ *func*)

Definition at line 36 of file c-assert.c.

References panic(), and printk().

```
                         {

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

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

Here is the call graph for this function:



4.65 include/kstdc/stdarg.h File Reference

Macros

- #define **va_start**(ap, parmN) __builtin_va_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**

4.65.1 Macro Definition Documentation

4.65.1.1 #define va_arg __builtin_va_arg

Definition at line 38 of file stdarg.h.

4.65.1.2 #define va_copy(*dest*, *src*) __builtin_va_copy((dest), (src))

Definition at line 40 of file stdarg.h.

4.65.1.3 #define va_end __builtin_va_end

Definition at line 39 of file stdarg.h.

4.65.1.4 #define va_start(*ap*, *parmN*) __builtin_va_start((ap), (parmN))

Definition at line 37 of file stdarg.h.

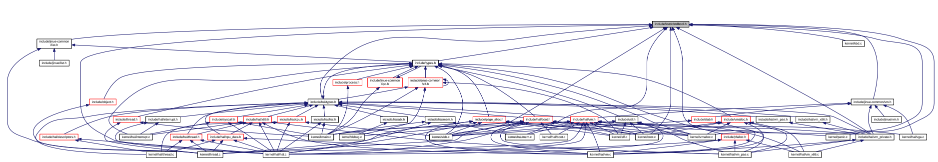
4.65.2 Typedef Documentation

4.65.2.1 typedef __builtin_va_list va_list

Definition at line 35 of file stdarg.h.

4.66 include/kstdc/stdbool.h File Reference

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



Macros

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

4.66.1 Macro Definition Documentation

4.66.1.1 #define __bool_true_false_are_defined 1

Definition at line 39 of file stdbool.h.

4.66.1.2 #define bool _Bool

Definition at line 35 of file stdbool.h.

4.66.1.3 #define false 0

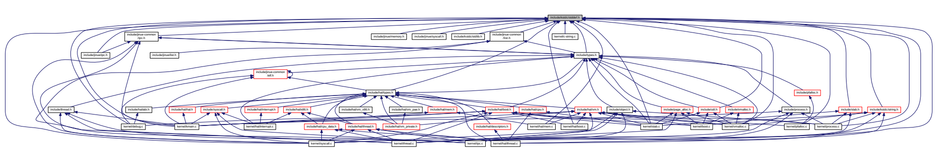
Definition at line 37 of file stdbool.h.

4.66.1.4 #define true 1

Definition at line 36 of file stdbool.h.

4.67 include/kstdc/stddef.h File Reference

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



Macros

- #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**

4.67.1 Macro Definition Documentation

4.67.1.1 #define NULL 0

Definition at line 40 of file stddef.h.

Referenced by add_page_frame(), boot_heap_pop(), boot_info_check(), boot_page_alloc_early(), boot_vmalloc(), cpu_init_data(), dispatch_syscall(), dump_call_stack(), elf_lookup_symbol(), ipc_boot_init(), ipc_object_create(), ipc_receive(), ipc_reply(), ipc_send(), kmain(), page_alloc(), page_alloc_is_empty(), process_boot_init(), process_create(), process_create_initial(), process_get_descriptor(), remove_page_frame(), slab_cache_alloc(), slab_cache_free(), slab_cache_init(), thread_context_switch(), thread_create(), thread_page_init(), thread_switch(), vm_change_flags(), vm_destroy_addr_space(), vm_lookup_kernel_paddr(), vm_map_kernel(), vm_pae_create_addr_space(), vm_pae_create_pdpt_cache(), vm_pae_lookup_page_directory(), and vm_unmap_kernel().

4.67.1.2 #define offsetof(*type, member*) ((size_t) &(((type ∗)0)->member))

Definition at line 43 of file stddef.h.

4.67.2 Typedef Documentation

4.67.2.1 typedef signed long ptrdiff_t

Definition at line 35 of file stddef.h.

4.67.2.2 typedef unsigned long size_t

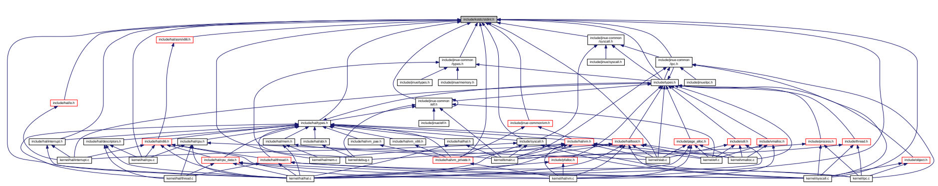
Definition at line 36 of file stddef.h.

4.67.2.3 typedef int wchar_t

Definition at line 37 of file stddef.h.

4.68 include/kstdc/stdint.h File Reference

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



Macros

- #define **INT64_C**(x) (x##LL)
- #define **UINT64_C**(x) (x##ULL)

Typedefs

- typedef signed char **int8_t**
- typedef short int **int16_t**
- typedef int **int32_t**
- typedef long long int **int64_t**
- typedef unsigned char **uint8_t**
- typedef unsigned short int **uint16_t**
- typedef unsigned int **uint32_t**
- typedef unsigned long long int **uint64_t**
- typedef int **intptr_t**
- typedef unsigned int **uintptr_t**

4.68.1 Macro Definition Documentation

4.68.1.1 #define INT64_C(*x*) (x##LL)

Definition at line 35 of file stdint.h.

4.68.1.2 #define UINT64_C(*x*) (x##ULL)

Definition at line 37 of file stdint.h.

4.68.2 Typedef Documentation

4.68.2.1 typedef short int int16_t

Definition at line 41 of file stdint.h.

4.68.2.2 typedef int int32_t

Definition at line 43 of file stdint.h.

4.68.2.3 typedef long long int int64_t

Definition at line 45 of file stdint.h.

4.68.2.4 typedef signed char int8_t

Definition at line 39 of file stdint.h.

4.68.2.5 typedef int intptr_t

Definition at line 57 of file stdint.h.

4.68.2.6 typedef unsigned short int uint16_t

Definition at line 50 of file stdint.h.

4.68.2.7 typedef unsigned int uint32_t

Definition at line 52 of file stdint.h.

4.68.2.8 typedef unsigned long long int uint64_t

Definition at line 54 of file stdint.h.

4.68.2.9 typedef unsigned char uint8_t

Definition at line 48 of file stdint.h.

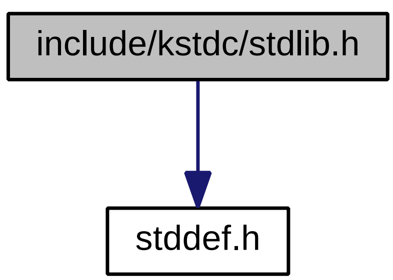
4.68.2.10 typedef unsigned int uintptr_t

Definition at line 59 of file stdint.h.

4.69 include/kstdc/stdlib.h File Reference

```
#include <stddef.h>
```

Include dependency graph for stdlib.h:



Macros

- #define **EXIT_SUCCESS** 0
- #define **EXIT_FAILURE** 1

4.69.1 Macro Definition Documentation

4.69.1.1 #define EXIT_FAILURE 1

Definition at line 39 of file stdlib.h.

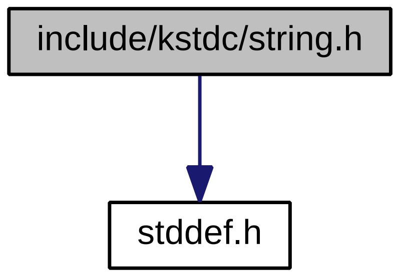
4.69.1.2 #define EXIT_SUCCESS 0

Definition at line 37 of file stdlib.h.

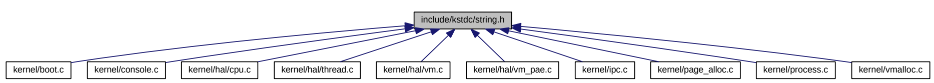
4.70 include/kstdc/string.h File Reference

`#include <stddef.h>`

Include dependency graph for string.h:



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



Functions

- void ∗ **memset** (void ∗s, int c, **size_t** n)
- void ∗ **memcpy** (void ∗dest, const void ∗src, **size_t** n)
- **size_t strlen** (const char ∗s)

4.70.1 Function Documentation

4.70.1.1 void∗ memcpy (void ∗ *dest*, const void ∗ *src*, size_t *n*)

Definition at line 45 of file c-string.c.

Referenced by ipc_receive(), ipc_reply(), ipc_send(), and process_create_initial().

```
45                                                        {
46     size_t       idx;
47     char        *cdest   = dest;
48     const char  *csrc    = src;
49
50     for(idx = 0; idx < n; ++idx) {
51         cdest[idx] = csrc[idx];
52     }
53
54     return dest;
55 }
```

4.70.1.2 void∗ memset (void ∗ *s*, int *c*, size_t *n*)

Definition at line 34 of file c-string.c.

Referenced by boot_alloc_init(), clear_page(), cpu_init_data(), thread_page_init(), and vm_pae_lookup_page_-directory().

```
34                                     {
35     size_t   idx;
36     char    *cs = s;
37 
38     for(idx = 0; idx < n; ++idx) {
39         cs[idx] = c;
40     }
41 
42     return s;
43 }
```

4.70.1.3 size_t strlen (const char ∗ *s*)

Definition at line 57 of file c-string.c.

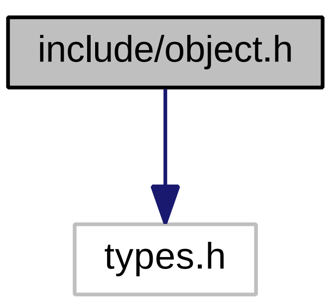
Referenced by console_print().

```
57                           {
58     size_t count = 0;
59 
60     while(*s != 0) {
61         ++s;
62         ++count;
63     }
64 
65     return count;
66 }
```

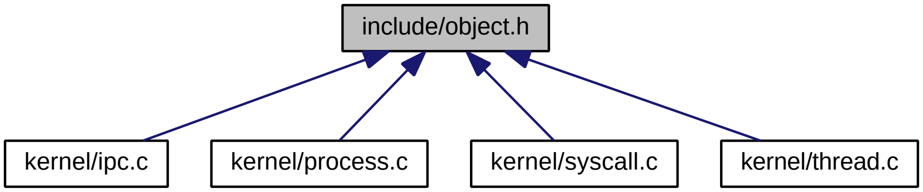
4.71 include/object.h File Reference

`#include <types.h>`

Include dependency graph for object.h:



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



Macros

- #define **OBJECT_FLAG_NONE** 0
- #define **OBJECT_FLAG_DESTROYED** (1<<0)
- #define **OBJECT_REF_FLAG_NONE** 0
- #define **OBJECT_REF_FLAG_VALID** (1<<0)
- #define **OBJECT_REF_FLAG_CLOSED** (1<<1)
- #define **OBJECT_REF_FLAG_OWNER** (1<<2)
- #define **OBJECT_TYPE_THREAD** 1
- #define **OBJECT_TYPE_IPC** 2
- #define **OBJECT_TYPE_PROCESS** 3

4.71.1 Macro Definition Documentation

4.71.1.1 #define OBJECT_FLAG_DESTROYED (1<<0)

Definition at line 42 of file object.h.

4.71.1.2 #define OBJECT_FLAG_NONE 0

Definition at line 40 of file object.h.

4.71.1.3 #define OBJECT_REF_FLAG_CLOSED (1<<1)

Definition at line 49 of file object.h.

Referenced by ipc_receive(), and ipc_send().

4.71.1.4 #define OBJECT_REF_FLAG_NONE 0

Definition at line 45 of file object.h.

4.71.1.5 #define OBJECT_REF_FLAG_OWNER (1<<2)

Definition at line 51 of file object.h.

Referenced by dispatch_syscall().

4.71.1.6 #define OBJECT_REF_FLAG_VALID (1<<0)

Definition at line 47 of file object.h.

Referenced by dispatch_syscall().

4.71.1.7 #define OBJECT_TYPE_IPC 2

Definition at line 56 of file object.h.

Referenced by ipc_receive(), and ipc_send().

4.71.1.8 #define OBJECT_TYPE_PROCESS 3

Definition at line 58 of file object.h.

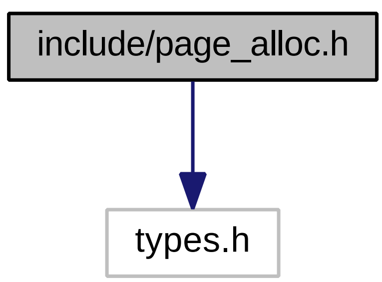
4.71.1.9 #define OBJECT_TYPE_THREAD 1

Definition at line 54 of file object.h.

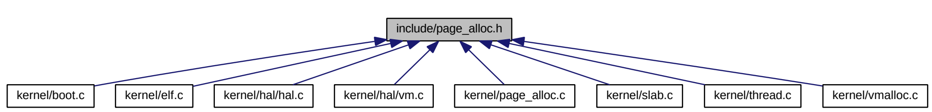
4.72 include/page_alloc.h File Reference

```
#include <types.h>
```

Include dependency graph for page_alloc.h:



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



Functions

- void ∗ **page_alloc** (void)

  Allocate a page of kernel memory.
- void **page_free** (void ∗page)

  Free a page of kernel memory.

- **bool page_alloc_is_empty** (void)

  Check that pages are available to be allocated.

- **bool add_page_frame** (**kern_paddr_t** paddr)

  Map a page frame and add it to the page allocator.

- **kern_paddr_t remove_page_frame** (void)

  Remove a page frame from the allocator.

- void **clear_page** (void ∗page)

  Clear a page by writing all bytes to zero.

4.72.1 Function Documentation

4.72.1.1 bool add_page_frame (kern_paddr_t *paddr*)

Map a page frame and add it to the page allocator.

This function is used to implement a system call that allows userspace to provide additional page frames to the kernel. This function fails when no more pages of kernel address space can be allocated with **vmalloc()** (p. 275) to map the provided page frame.

Parameters

paddr	physical address of the provided page frame

Returns

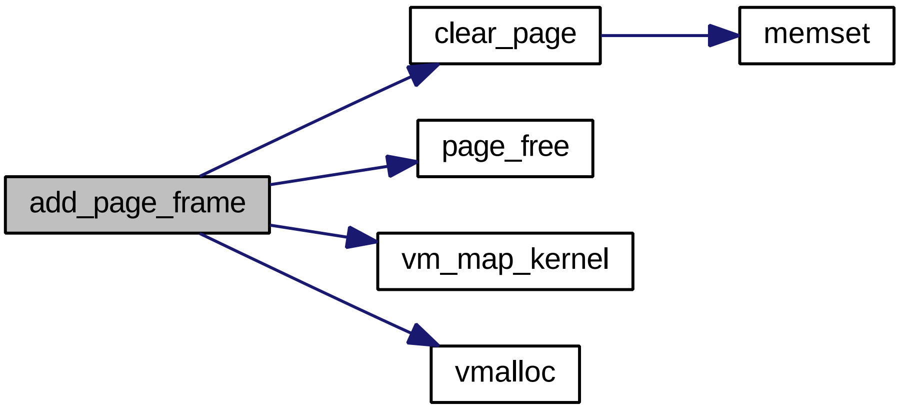
true if the function succeeded

Definition at line 111 of file page_alloc.c.

References clear_page(), NULL, page_free(), VM_FLAG_READ_WRITE, vm_map_kernel(), and vmalloc().

```
111                                         {
112     void *page = vmalloc();
113 
114     if(page == NULL) {
115         return false;
116     }
117 
118     vm_map_kernel(page, paddr, VM_FLAG_READ_WRITE);
119 
120     /* Since this page is coming from userspace, is is important to clear it:
121      * 1) The page may contain sensitive information, which we don't want to
122      *    leak through Meltdown-like vulnerabilities; and
123      * 2) Since the content is userspace-chosen, it could be used for kernel
124      *    vulnerability exploits. */
125     clear_page(page);
126     page_free(page);
127 
128     return true;
129 }
```

Here is the call graph for this function:



4.72.1.2 void clear_page (void * *page*)

Clear a page by writing all bytes to zero.

Parameters

page	the page to clear

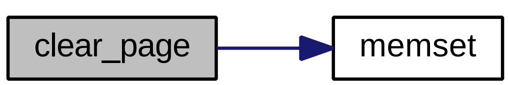
Definition at line 173 of file page_alloc.c.

References memset(), and PAGE_SIZE.

Referenced by add_page_frame(), boot_page_alloc(), boot_page_alloc_early(), boot_page_alloc_image(), elf_setup_-stack(), and remove_page_frame().

```
173                              {
174     memset(page, 0, PAGE_SIZE);
175 }
```

Here is the call graph for this function:



4.72.1.3 void* page_alloc (void)

Allocate a page of kernel memory.

Pages allocated by this function can be used for any purpose in the kernel, e.g. as slabs for the slab allocator or as page tables.

Pages allocated by this function are not guaranteed to be mapped in the allocations region of the kernel address space (that is, the region managed by **vmalloc()** (p. 275)). While most will be, pages originally allocated in the image region during initialization by calling **boot_page_alloc_image()** (p. 72) can be reclaimed with **page_free()** (p. 252) and then re-allocated by this function.

Returns

allocated page

Definition at line 59 of file page_alloc.c.

References alloc_page::next, and NULL.

Referenced by remove_page_frame(), slab_cache_alloc(), thread_create(), and vm_clone_page_directory().

```
59                          {
60      struct alloc_page *alloc_page = head_page;
61
62      if(alloc_page != NULL) {
63          head_page = alloc_page->next;
64      }
65
66      return alloc_page;
67  }
```

4.72.1.4 bool page_alloc_is_empty (void)

Check that pages are available to be allocated.

Page availability can be checked with this function before calling either **page_alloc()** (p. 251) or **remove_page_frame()** (p. 253).

Returns

true if pages are available (one or more)

Definition at line 95 of file page_alloc.c.

References NULL.

```
95                                  {
96      return head_page == NULL;
97  }
```

4.72.1.5 void page_free (void * *page*)

Free a page of kernel memory.

Pages freed by calling this function are available to be re-allocated by the **page_alloc()** (p. 251) function. This function can be used to free pages allocated by **page_alloc()** (p. 251) or to reclaim pages allocated during kernel initialization by **boot_page_alloc()** (p. 70) or **boot_page_alloc_image()** (p. 72).

Parameters

page	the page to free

Definition at line 80 of file page_alloc.c.

References alloc_page::next.

Referenced by add_page_frame(), hal_init(), and thread_destroy().

```
80                              {
81      struct alloc_page *alloc_page = page;
82      alloc_page->next    = head_page;
83      head_page           = alloc_page;
84  }
```

4.72.1.6 kern_paddr_t remove_page_frame (void)

Remove a page frame from the allocator.

This function is used implement a system call that allows userspace to reclaim free kernel memory for its own use. The address space page is freed with **vmfree()** (p. 276) and the physical address of the underlying page frame is returned.

Returns

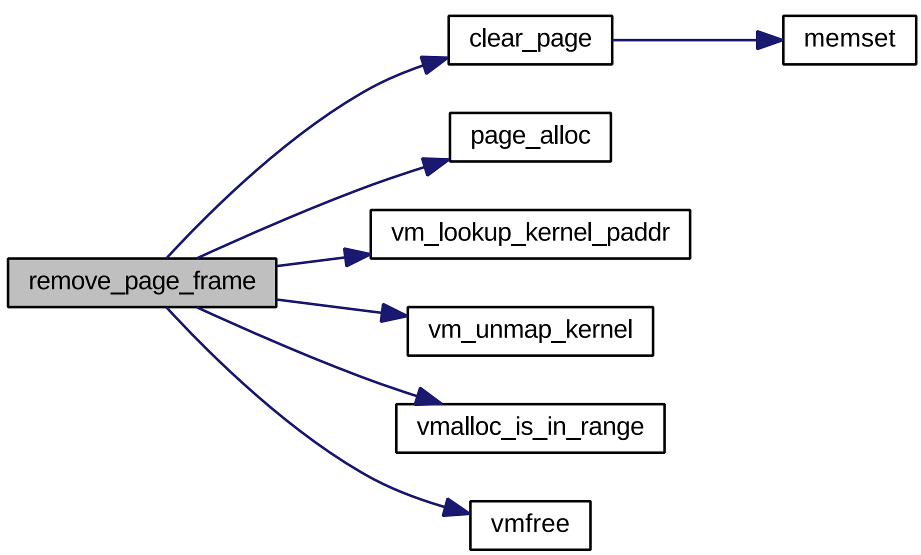
physical address of the freed page frame, or PFNULL if none is available

Definition at line 142 of file page_alloc.c.

References clear_page(), NULL, page_alloc(), PFNULL, vm_lookup_kernel_paddr(), vm_unmap_kernel(), vmalloc_is_in_range(), and vmfree().

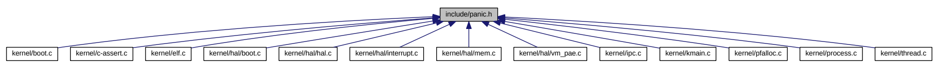
```
142                                     {
143     void *page = page_alloc();
144
145     if(page == NULL) {
146         return PFNULL;
147     }
148
149     /* This page is going to userspace. Let's clear its content so we don't
150      * leak information about the kernel's internal state that could be useful
151      * for exploiting vulnerabilities. */
152     clear_page(page);
153
154     kern_paddr_t paddr = vm_lookup_kernel_paddr(page);
155
156     vm_unmap_kernel(page);
157
158     /* The page may be in the image region instead of the allocations region if
159      * it was allocated during kernel initialization. */
160     if(vmalloc_is_in_range(page)) {
161         vmfree(page);
162     }
163
164     return paddr;
165 }
```

Here is the call graph for this function:



4.73 include/panic.h File Reference

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



Functions

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

4.73.1 Function Documentation

4.73.1.1 void panic (const char ∗ *message*)

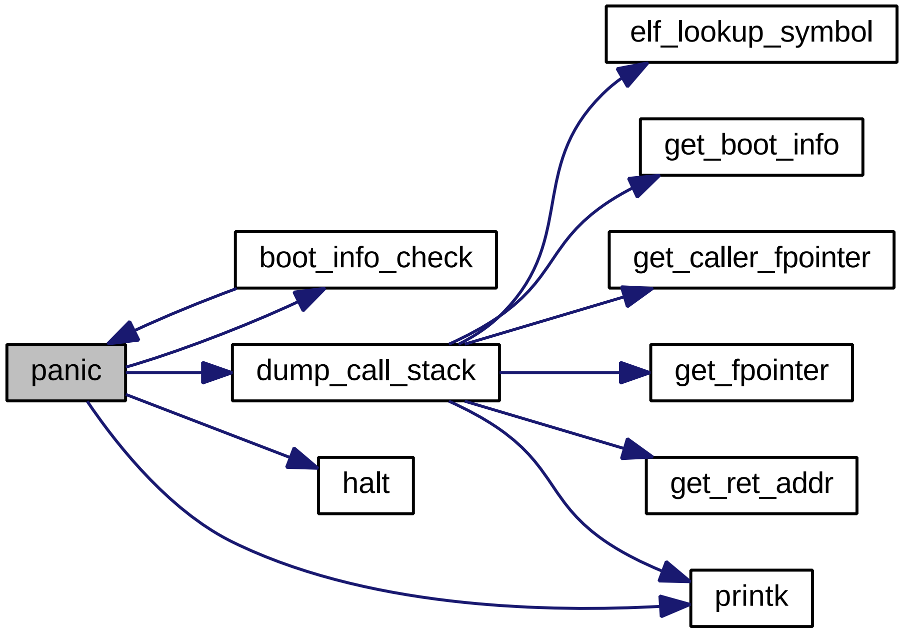
Definition at line 40 of file panic.c.

References boot_info_check(), dump_call_stack(), halt(), printk(), and VGA_COLOR_RED.

Referenced by __assert_failed(), boot_heap_pop(), boot_info_check(), boot_page_alloc_early(), boot_page_frame_-alloc(), boot_vmalloc(), dispatch_interrupt(), elf_check(), elf_load(), ipc_boot_init(), kmain(), mem_check_memory(), and pfalloc_from().

```
40                              {
41     static int enter_count = 0;
42
43     ++enter_count;
44
45     /* When things go seriously wrong, things that panic does itself can create
46      * a further panic, for example by triggering a hardware exception. The
47      * enter_count static variable keeps count of the number of times panic()
48      * is entered. */
49     switch(enter_count) {
50     case 1:
51     case 2:
52         /* The first two times panic() is entered, a panic message is displayed
53          * along with a full call sack dump. */
54         printk( "%kKERNEL PANIC%s: %s\n",
55                 VGA_COLOR_RED,
56                 enter_count==1?"":" (recursive)",
57                 message);
58
59         if( boot_info_check(false) ) {
60             dump_call_stack();
61         }
62         else {
63             printk("Cannot dump call stack because boot information structure is invalid.\n");
64         }
65         break;
66     case 3:
67         /* The third time, a "recursive count exceeded" message is displayed. We
68          * try to limit the number of actions we take to limit the chances of a
69          * further panic. */
70         printk("%kKERNEL PANIC (recursive count exceeded)\n", VGA_COLOR_RED);
71         break;
72     default:
73         /* The fourth time, we do nothing but halt the CPU. */
74         break;
75     }
76
77     halt();
78 }
```

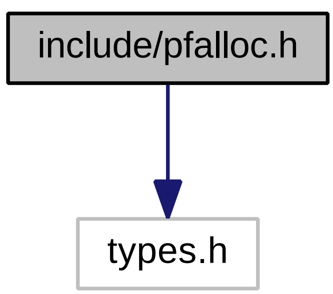
Here is the call graph for this function:



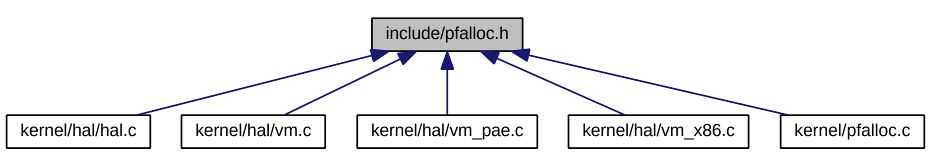
4.74 include/pfalloc.h File Reference

`#include <types.h>`

Include dependency graph for pfalloc.h:



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



Data Structures

- struct **pfalloc_cache_t**

Macros

- #define **KERNEL_PAGE_STACK_SIZE** 1024
- #define **KERNEL_PAGE_STACK_INIT** 32
- #define **pfalloc**() **pfalloc_from**(&**global_pfalloc_cache**)
- #define **pffree**(p) **pffree_to**(&**global_pfalloc_cache**, (p))

Functions

- void **init_pfalloc_cache** (**pfalloc_cache_t** ∗pfcache, **kern_paddr_t** ∗stack_page)
- **kern_paddr_t** **pfalloc_from** (**pfalloc_cache_t** ∗pfcache)
- void **pffree_to** (**pfalloc_cache_t** ∗pfcache, **kern_paddr_t** paddr)

Variables

- **pfalloc_cache_t global_pfalloc_cache**

4.74.1 Macro Definition Documentation

4.74.1.1 #define KERNEL_PAGE_STACK_INIT 32

Definition at line 39 of file pfalloc.h.

Referenced by hal_init().

4.74.1.2 #define KERNEL_PAGE_STACK_SIZE 1024

Definition at line 37 of file pfalloc.h.

Referenced by init_pfalloc_cache(), and pffree_to().

4.74.1.3 #define pfalloc( ) pfalloc_from(&global_pfalloc_cache)

Definition at line 50 of file pfalloc.h.

Referenced by vm_pae_lookup_page_directory().

4.74.1.4 #define pffree(*p*) pffree_to(&global_pfalloc_cache, (p))

Definition at line 52 of file pfalloc.h.

Referenced by hal_init(), and vm_destroy_page_directory().

4.74.2 Function Documentation

4.74.2.1 void init_pfalloc_cache (pfalloc_cache_t * *pfcache*, kern_paddr_t * *stack_page*)

Definition at line 40 of file pfalloc.c.

References pfalloc_cache_t::count, KERNEL_PAGE_STACK_SIZE, PFNULL, and pfalloc_cache_t::ptr.

Referenced by hal_init().

```
40                                                                              {
41      kern_paddr_t    *ptr;
42      unsigned int     idx;
43
44      ptr = stack_page;
45
46      for(idx = 0;idx < KERNEL_PAGE_STACK_SIZE; ++idx) {
47          ptr[idx] = PFNULL;
48      }
49
50      pfcache->ptr   = stack_page;
51      pfcache->count = 0;
52  }
```

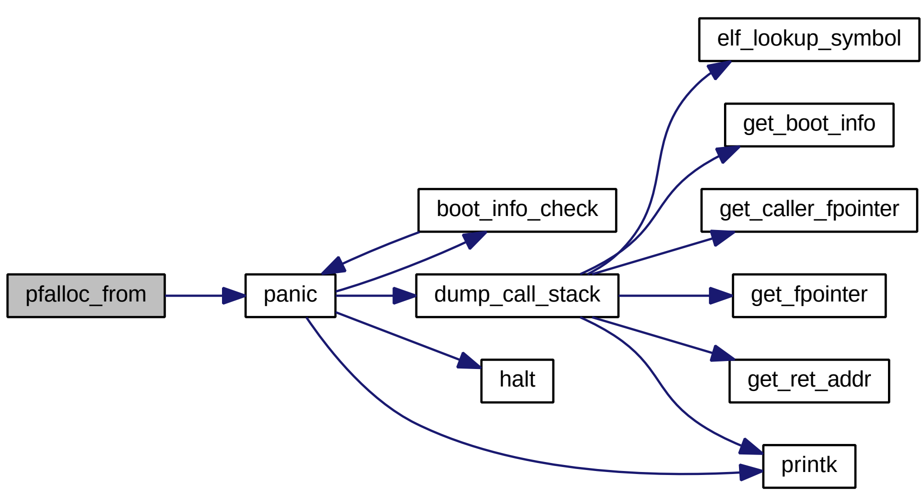
4.74.2.2 kern_paddr_t pfalloc_from (pfalloc_cache_t * *pfcache*)

Definition at line 54 of file pfalloc.c.

References pfalloc_cache_t::count, panic(), and pfalloc_cache_t::ptr.

```
54                                                        {
55      if(pfcache->count == 0) {
56          panic("pfalloc_from(): no more pages to allocate");
57      }
58
59      --pfcache->count;
60
61      return *(--pfcache->ptr);
62  }
```

Here is the call graph for this function:



4.74.2.3 void pffree_to (pfalloc_cache_t ∗ *pfcache,* kern_paddr_t *paddr*)

We are leaking memory here. Should we panic instead?

Definition at line 64 of file pfalloc.c.

References pfalloc_cache_t::count, KERNEL_PAGE_STACK_SIZE, and pfalloc_cache_t::ptr.

```
64                                                          {
65     if(pfcache->count >= KERNEL_PAGE_STACK_SIZE) {
67         return;
68     }
69
70     ++pfcache->count;
71
72     (pfcache->ptr++)[0] = paddr;
73 }
```

4.74.3 Variable Documentation

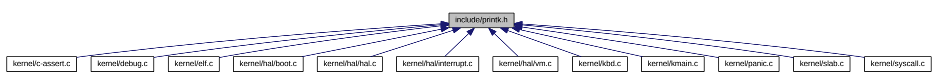
4.74.3.1 pfalloc_cache_t global_pfalloc_cache

Definition at line 38 of file pfalloc.c.

Referenced by hal_init().

4.75 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, int colour)
- void **print_hex_nibble** (unsigned char byte, int colour)
- void **print_hex_b** (unsigned char byte, int colour)
- void **print_hex_w** (unsigned short word, int colour)
- void **print_hex_l** (unsigned long dword, int colour)
- void **print_hex_q** (unsigned long long qword, int colour)

4.75.1 Function Documentation

4.75.1.1 void print_hex_b (unsigned char *byte,* int *colour*)

4.75.1.2 void print_hex_l (unsigned long *dword,* int *colour*)

4.75.1.3 void print_hex_nibble (unsigned char *byte,* int *colour*)

4.75.1.4 void print_hex_q (unsigned long long *qword,* int *colour*)

4.75.1.5 void print_hex_w (unsigned short *word,* int *colour*)

4.75.1.6 void print_unsigned_int (unsigned int *n,* int *colour*)

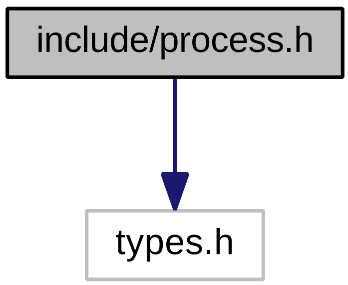
4.75.1.7 void printk (const char ∗ *format,* ...)

Referenced by __assert_failed(), any_key(), boot_info_dump(), dispatch_interrupt(), dispatch_syscall(), dump_call_-stack(), elf_load(), kmain(), panic(), slab_cache_alloc(), slab_cache_free(), and vm_boot_init().

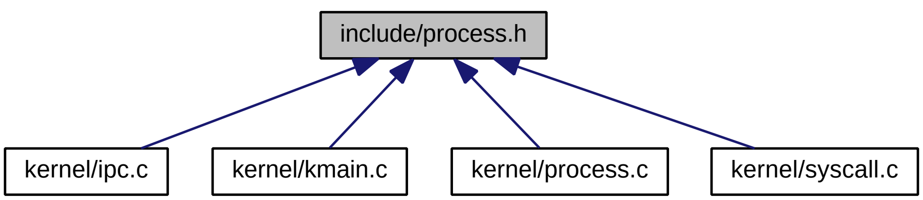
4.76 include/process.h File Reference

`#include <types.h>`

Include dependency graph for process.h:



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



Functions

- void **process_boot_init** (**boot_alloc_t** ∗boot_alloc)
- **process_t** ∗ **process_create** (void)
- **process_t** ∗ **process_create_initial** (void)
- **object_ref_t** ∗ **process_get_descriptor** (**process_t** ∗process, int fd)
- int **process_unused_descriptor** (**process_t** ∗process)

4.76.1 Function Documentation

4.76.1.1 void process_boot_init (boot_alloc_t ∗ *boot_alloc*)

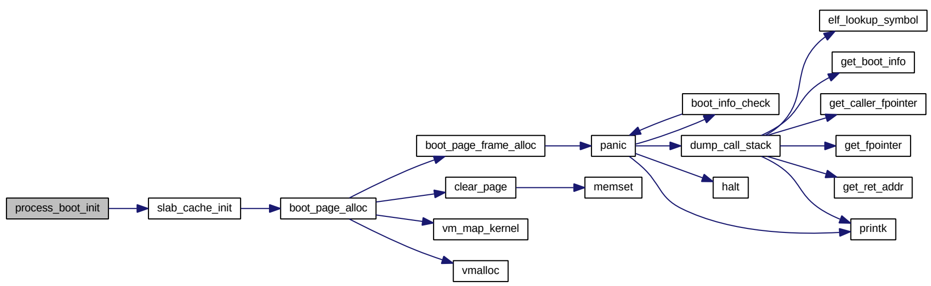
Definition at line 49 of file process.c.

References NULL, slab_cache_init(), and SLAB_DEFAULTS.

Referenced by kmain().

```
{
    slab_cache_init(
            &process_cache,
            "process_cache",
            sizeof(process_t),
            0,
            process_ctor,
            NULL,
            SLAB_DEFAULTS,
            boot_alloc);
}
```

Here is the call graph for this function:



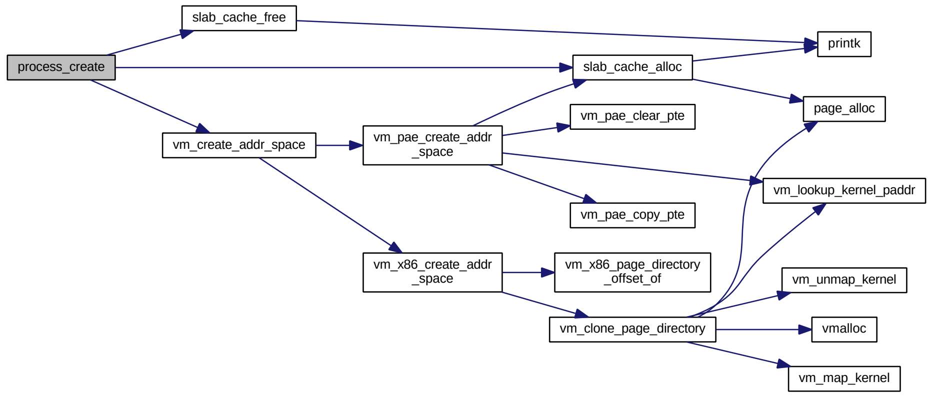
4.76.1.2 process_t∗ process_create (void)

Definition at line 65 of file process.c.

References process_t::addr_space, NULL, slab_cache_alloc(), slab_cache_free(), and vm_create_addr_space().

```
65                              {
66     process_t *process = slab_cache_alloc(&process_cache);
67
68     if(process != NULL) {
69         addr_space_t *addr_space = vm_create_addr_space(&process->addr_space);
70
71         /* The address space object is located inside the process object but the
72          * call to vm_create_addr_space() above can still fail if we cannot
73          * allocate the initial page directory/tables or, when PAE is enabled,
74          * if we cannot allocate a PDPT. */
75         if(addr_space == NULL) {
76             slab_cache_free(process);
77             return NULL;
78         }
79
80         process_init(process);
81     }
82
83     return process;
84 }
```

Here is the call graph for this function:



4.76.1.3 process_t∗ process_create_initial (void)

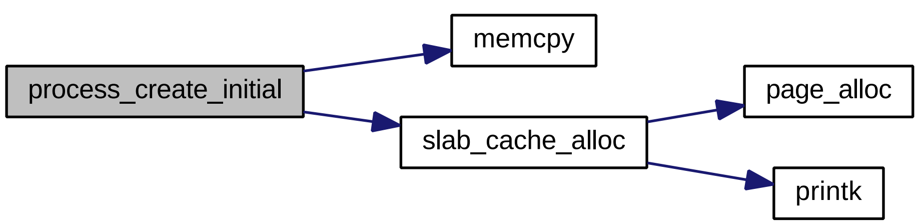
Definition at line 86 of file process.c.

References process_t::addr_space, initial_addr_space, memcpy(), NULL, and slab_cache_alloc().

Referenced by kmain().

```
86                                     {
87     process_t *process = slab_cache_alloc(&process_cache);
88
89     if(process != NULL) {
90         memcpy(&process->addr_space, &initial_addr_space, sizeof(addr_space_t));
91         process_init(process);
92     }
93
94     return process;
95 }
```

Here is the call graph for this function:



4.76.1.4 object_ref_t∗ process_get_descriptor (process_t ∗ *process,* int *fd*)

Definition at line 97 of file process.c.

References process_t::descriptors, NULL, and PROCESS_MAX_DESCRIPTORS.

Referenced by dispatch_syscall(), ipc_receive(), ipc_send(), and process_unused_descriptor().

```
97                                                               {
```

```
98      if(fd < 0 || fd > PROCESS_MAX_DESCRIPTORS) {
99          return NULL;
100     }
101 
102     return &process->descriptors[fd];
103 }
```

4.76.1.5 int process_unused_descriptor (process_t ∗ *process*)

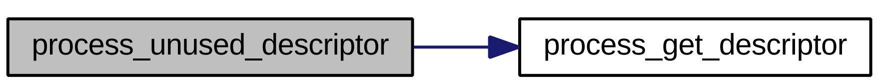
Definition at line 105 of file process.c.

References process_get_descriptor(), and PROCESS_MAX_DESCRIPTORS.

Referenced by dispatch_syscall().

```
105                                                   {
106     int idx;
107 
108     for(idx = 0; idx < PROCESS_MAX_DESCRIPTORS; ++idx) {
109         object_ref_t *ref = process_get_descriptor(process, idx);
110 
111         if(! object_ref_is_valid(ref)) {
112             return idx;
113         }
114     }
115 
116     return -1;
117 }
```

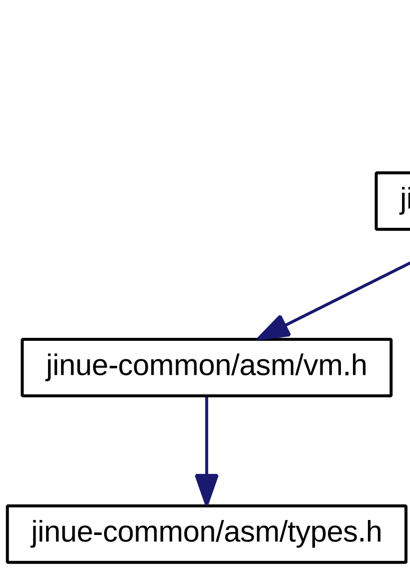
Here is the call graph for this function:



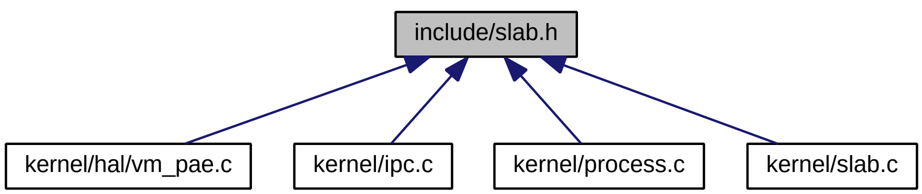
4.77 include/slab.h File Reference

```
#include <jinue-common/vm.h>
#include <stddef.h>
```

Include dependency graph for slab.h:



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



Data Structures

- struct **slab_cache_t**
- struct **slab_bufctl_t**
- struct **slab_t**

Macros

- #define **SLAB_SIZE PAGE_SIZE**
- #define **SLAB_POISON_ALIVE_VALUE** 0x0BADCAFE
- #define **SLAB_POISON_DEAD_VALUE** 0xDEADBEEF
- #define **SLAB_RED_ZONE_VALUE** 0x5711600D
- #define **SLAB_DEFAULT_WORKING_SET** 2
- #define **SLAB_DEFAULTS** (0)
- #define **SLAB_RED_ZONE** (1<<0)
- #define **SLAB_POISON** (1<<1)
- #define **SLAB_HWCACHE_ALIGN** (1<<2)
- #define **SLAB_COMPACT** (1<<3)

Typedefs

- typedef void(∗ **slab_ctor_t**)(void ∗, **size_t**)
- typedef struct **slab_cache_t** **slab_cache_t**
- typedef struct **slab_bufctl_t** **slab_bufctl_t**
- typedef struct **slab_t** **slab_t**

Functions

- void **slab_cache_init** (**slab_cache_t** ∗cache, char ∗name, **size_t** size, **size_t** alignment, **slab_ctor_t** ctor, **slab_ctor_t** dtor, int flags, **boot_alloc_t** ∗boot_alloc)

  Initialize an object cache.
- void ∗ **slab_cache_alloc** (**slab_cache_t** ∗cache)

  Allocate an object from the specified cache.
- void **slab_cache_free** (void ∗buffer)

  Free an object.
- void **slab_cache_reap** (**slab_cache_t** ∗cache)

  Return memory to the page allocator.
- void **slab_cache_set_working_set** (**slab_cache_t** ∗cache, unsigned int n)

  Set a cache's working set.

Variables

- **slab_cache_t** ∗ **slab_cache_list**

4.77.1 Macro Definition Documentation

4.77.1.1 #define SLAB_COMPACT (1<<3)

Definition at line 57 of file slab.h.

Referenced by slab_cache_init().

4.77.1.2 #define SLAB_DEFAULT_WORKING_SET 2

Definition at line 46 of file slab.h.

Referenced by slab_cache_init().

4.77.1.3 #define SLAB_DEFAULTS (0)

Definition at line 49 of file slab.h.

Referenced by ipc_boot_init(), process_boot_init(), and vm_pae_create_pdpt_cache().

4.77.1.4 #define SLAB_HWCACHE_ALIGN (1<<2)

Definition at line 55 of file slab.h.

Referenced by slab_cache_init().

4.77.1.5 #define SLAB_POISON (1<<1)

Definition at line 53 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_free(), and slab_cache_init().

4.77.1.6 #define SLAB_POISON_ALIVE_VALUE 0x0BADCAFE

Definition at line 40 of file slab.h.

Referenced by slab_cache_alloc().

4.77.1.7 #define SLAB_POISON_DEAD_VALUE 0xDEADBEEF

Definition at line 42 of file slab.h.

Referenced by slab_cache_alloc(), and slab_cache_free().

4.77.1.8 #define SLAB_RED_ZONE (1<<0)

Definition at line 51 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_free(), and slab_cache_init().

4.77.1.9 #define SLAB_RED_ZONE_VALUE 0x5711600D

Definition at line 44 of file slab.h.

Referenced by slab_cache_alloc(), and slab_cache_free().

4.77.1.10 #define SLAB_SIZE PAGE_SIZE

Definition at line 38 of file slab.h.

Referenced by slab_cache_free(), and slab_cache_init().

4.77.2 Typedef Documentation

4.77.2.1 typedef struct slab_bufctl_t slab_bufctl_t

Definition at line 88 of file slab.h.

4.77.2.2 typedef struct slab_cache_t slab_cache_t

Definition at line 82 of file slab.h.

4.77.2.3 typedef void(∗ slab_ctor_t)(void ∗, size_t)

Definition at line 60 of file slab.h.

4.77.2.4 typedef struct slab_t slab_t

Definition at line 101 of file slab.h.

4.77.3 Function Documentation

4.77.3.1 void* slab_cache_alloc (slab_cache_t * *cache*)

Allocate an object from the specified cache.

The cache must have been initialized with **slab_cache_init()** (p. 270). If no more space is available on existing slabs, this function tries to allocate a new slab using the kernel's page allocator (i.e. **page_alloc()** (p. 251)). It page allocation fails, this function fails by returning NULL.

Parameters

cache	the cache from which to allocate an object

Returns

the address of the allocated object, or NULL if allocation failed

ASSERTION: now that slab_cache_grow() has run, we should have found at least one empty slab

Important note regarding the slab lists: The empty, partial and full slab lists are doubly-linked lists. This is done to allow the deletion of an arbitrary link given a pointer to it. We do not allow reverse traversal: we do not maintain a tail pointer and, more importantly: we do *NOT* maintain the previous pointer of the first link in the list (i.e. it is garbage data, not NULL).

ASSERTION: there is at least one buffer on the free list

ASSERT: the slab is the head of the partial list

Definition at line 232 of file slab.c.

References assert, slab_cache_t::bufctl_offset, slab_cache_t::ctor, slab_cache_t::empty_count, slab_cache_t::flags, slab_t::free_list, slab_cache_t::name, slab_bufctl_t::next, slab_t::next, NULL, slab_t::obj_count, slab_cache_t::obj_size, page_alloc(), slab_t::prev, printk(), SLAB_POISON, SLAB_POISON_ALIVE_VALUE, SLAB_POISON_DEAD_VALUE, SLAB_RED_ZONE, SLAB_RED_ZONE_VALUE, slab_cache_t::slabs_empty, slab_cache_t::slabs_full, and slab_cache_t::slabs_partial.

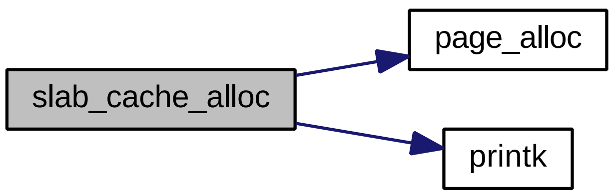
Referenced by ipc_boot_init(), ipc_object_create(), process_create(), process_create_initial(), and vm_pae_create_addr_space().

```
232                                             {
233     slab_t          *slab;
234 
235     if(cache->slabs_partial != NULL) {
236         slab = cache->slabs_partial;
237     }
238     else {
239         if(cache->slabs_empty == NULL) {
240             void *slab_addr = page_alloc();
241 
242             if(slab_addr == NULL) {
243                 return NULL;
244             }
245 
246             init_and_add_slab(cache, slab_addr);
247         }
248 
249         slab = cache->slabs_empty;
250 
252         assert(slab != NULL);
```

```
        /* We are about to allocate one object from this slab, so it will
         *  not be empty anymore...*/
        cache->slabs_empty      = slab->next;

        --(cache->empty_count);

        slab->next              = cache->slabs_partial;
        if(slab->next != NULL) {
            slab->next->prev = slab;
        }
        cache->slabs_partial    = slab;
    }

    slab_bufctl_t *bufctl = slab->free_list;

    assert(bufctl != NULL);

    slab->free_list  = bufctl->next;
    slab->obj_count += 1;

    /* If we just allocated the last buffer, move the slab to the full
     * list */
    if(slab->free_list == NULL) {
        /* remove from the partial slabs list */

        assert(cache->slabs_partial == slab);

        cache->slabs_partial = slab->next;

        if(slab->next != NULL) {
            slab->next->prev = slab->prev;
        }

        /* add to the full slabs list */
        slab->next        = cache->slabs_full;
        cache->slabs_full = slab;

        if(slab->next != NULL) {
            slab->next->prev = slab;
        }
    }

    uint32_t *buffer = (uint32_t *)( (char *)bufctl - cache->bufctl_offset );

    if(cache->flags & SLAB_POISON) {
        unsigned int idx;
        unsigned int dump_lines = 0;

        for(idx = 0; idx < cache->obj_size / sizeof(uint32_t); ++idx) {
            if(buffer[idx] != SLAB_POISON_DEAD_VALUE) {
                if(dump_lines == 0) {
                    printk("detected write to freed object, cache: %s buffer: 0x%x:\n",
                        cache->name,
                        (unsigned int)buffer
                    );
                }

                if(dump_lines < 4) {
                    printk(" value 0x%x at byte offset %u\n", buffer[idx], idx * sizeof(
uint32_t));
                }

                ++dump_lines;
            }

            buffer[idx] = SLAB_POISON_ALIVE_VALUE;
        }

        /* If both SLAB_POISON and SLAB_RED_ZONE are enabled, we perform
         * redzone checking even on freed objects. */
        if(cache->flags & SLAB_RED_ZONE) {
            if(buffer[idx] != SLAB_RED_ZONE_VALUE) {
                printk("detected write past the end of freed object, cache: %s buffer: 0x%x value: 0x%x\n",
                    cache->name,
                    (unsigned int)buffer,
                    buffer[idx]
                );
            }

            buffer[idx] = SLAB_RED_ZONE_VALUE;
```

```
344         }
345 
346         if(cache->ctor != NULL) {
347             cache->ctor((void *)buffer, cache->obj_size);
348         }
349     }
350     else if(cache->flags & SLAB_RED_ZONE) {
351         buffer[cache->obj_size / sizeof(uint32_t)] = SLAB_RED_ZONE_VALUE;
352     }
353 
354     return (void *)buffer;
355 }
```

Here is the call graph for this function:



4.77.3.2 void slab_cache_free (void ∗ *buffer*)

Free an object.

Parameters

buffer	the object to free

Definition at line 363 of file slab.c.

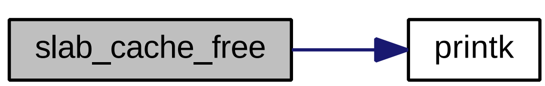
References ALIGN_START_PTR, slab_cache_t::bufctl_offset, slab_t::cache, slab_cache_t::dtor, slab_cache_t::empty_count, slab_cache_t::flags, slab_t::free_list, slab_cache_t::name, slab_bufctl_t::next, slab_t::next, NULL, slab_t::obj_count, slab_cache_t::obj_size, slab_t::prev, printk(), SLAB_POISON, SLAB_POISON_DEAD_VALUE, SLAB_RED_ZONE, SLAB_RED_ZONE_VALUE, SLAB_SIZE, slab_cache_t::slabs_empty, slab_cache_t::slabs_full, and slab_cache_t::slabs_partial.

Referenced by process_create(), and vm_pae_destroy_addr_space().

```
363                                    {
364     /* compute address of slab data structure */
365     addr_t slab_start       = ALIGN_START_PTR(buffer, SLAB_SIZE);
366     slab_t *slab            = (slab_t *)(slab_start + SLAB_SIZE - sizeof(slab_t) );
367 
368     /* obtain address of cache and bufctl */
369     slab_cache_t *cache     = slab->cache;
370     slab_bufctl_t *bufctl   = (slab_bufctl_t *)((char *)buffer + cache->
      bufctl_offset);
371 
372     /* If slab is on the full slabs list, move it to the partial list
373      * since we are about to return a buffer to it. */
374     if(slab->free_list == NULL) {
375         /* remove from full slabs list */
376         if(cache->slabs_full == slab) {
377             cache->slabs_full = slab->next;
378         }
379         else {
380             slab->prev->next = slab->next;
381         }
382 
383         if(slab->next != NULL) {
384             slab->next->prev = slab->prev;
385         }
386 
```

```c
        /* add to partial slabs list */
        slab->next           = cache->slabs_partial;
        cache->slabs_partial = slab;

        if(slab->next != NULL) {
            slab->next->prev = slab;
        }
    }

    if(cache->flags & SLAB_RED_ZONE) {
        uint32_t *rz_word = (uint32_t *)((char *)buffer + cache->obj_size);

        if(*rz_word != SLAB_RED_ZONE_VALUE) {
            printk("detected write past the end of object, cache: %s buffer: 0x%x value: 0x%x\n",
                cache->name,
                (unsigned int)buffer,
                *rz_word
            );
        }

        *rz_word = SLAB_RED_ZONE_VALUE;
    }

    if(cache->flags & SLAB_POISON) {
        unsigned int idx;

        if(cache->dtor != NULL) {
            cache->dtor(buffer, cache->obj_size);
        }

        uint32_t *buffer32 = (uint32_t *)buffer;

        for(idx = 0; idx < cache->obj_size / sizeof(uint32_t); ++idx) {
            buffer32[idx] = SLAB_POISON_DEAD_VALUE;
        }
    }

    /* link buffer into slab free list */
    bufctl->next     = slab->free_list;
    slab->free_list  = bufctl;
    slab->obj_count -= 1;

    /* If we just returned the last object to the slab, move the slab to
     * the empty list. */
    if(slab->obj_count == 0) {
        /* remove from partial slabs list */
        if(cache->slabs_partial == slab) {
            cache->slabs_partial = slab->next;
        }
        else {
            slab->prev->next = slab->next;
        }

        if(slab->next != NULL) {
            slab->next->prev = slab->prev;
        }

        /* add to empty slabs list */
        slab->next         = cache->slabs_empty;
        cache->slabs_empty = slab;

        if(slab->next != NULL) {
            slab->next->prev = slab;
        }

        ++(cache->empty_count);
    }
}
```

Here is the call graph for this function:



4.77.3.3 void slab_cache_init (**slab_cache_t** ∗ *cache,* char ∗ *name,* **size_t** *size,* **size_t** *alignment,* **slab_ctor_t** *ctor,* **slab_ctor_t** *dtor,* int *flags,* **boot_alloc_t** ∗ *boot_alloc*)

Initialize an object cache.

The following flags are supported:

- SLAB_HWCACHE_ALIGN Align objects on at least the line size of the CPU's data cache.
- SLAB_COMPACT the bufctl can safely be put inside the object without destroying the constructed state. If not set, additional space is reserved specifically for the bufctl to prevent corruption of the constructed state.
- SLAB_RED_ZONE (redzone checking - debugging) Add a guard word at the end of each object and use this to detect writes past the end of the object.
- SLAB_POISON (debugging) Fill uninitialized objects with a recognizable pattern before calling the constructor function to help identify members that do not get initialized. Do the same when freeing objects and use this to detect writes to freed objects.

This function uses the kernel's boot-time page allocator to allocate an initial slab. This helps with bootstrapping because it allows a few objects (up to s slab's worth) to be allocated before the main page allocator has been initialized and then replenished by user space. It also means this function can only be called during kernel initialization (it would not make sense to call it later).

Parameters

cache	the cache to initialize
name	a human-readable name for the cache, used in debugging messages
size	the size of objects allocated on this cache
alignment	the minimum object alignment, or zero for no constraint
ctor	the object constructor function
dtor	the object destructor function
flags	see description
boot_alloc	the kernel boot-time page allocator structure

ASSERTION: buffer size is at least the size of a pointer

ASSERTION: name is not NULL string

Definition at line 137 of file slab.c.

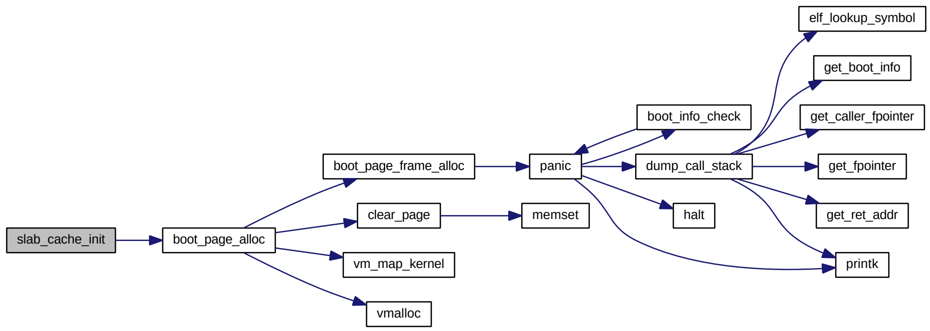
References ALIGN_END, slab_cache_t::alignment, slab_cache_t::alloc_size, assert, boot_page_alloc(), slab_cache_t::bufctl_offset, cpu_info, slab_cache_t::ctor, cpu_info_t::dcache_alignment, slab_cache_t::dtor, slab_cache_t::empty_count, slab_cache_t::flags, slab_cache_t::max_colour, slab_cache_t::name, slab_cache_t::next_colour, NULL, slab_cache_t::obj_size, SLAB_COMPACT, SLAB_DEFAULT_WORKING_SET, SLAB_HWCACHE_ALIGN, SLAB_POISON, SLAB_RED_ZONE, SLAB_SIZE, slab_cache_t::slabs_empty, slab_cache_t::slabs_full, slab_cache_t::slabs_partial, and slab_cache_t::working_set.

Referenced by ipc_boot_init(), process_boot_init(), and vm_pae_create_pdpt_cache().

```
                                         {

    assert(size >= sizeof(void *));

    assert(name != NULL);

    cache->name             = name;
    cache->ctor             = ctor;
    cache->dtor             = dtor;
    cache->slabs_empty      = NULL;
    cache->slabs_partial    = NULL;
    cache->slabs_full       = NULL;
    cache->empty_count      = 0;
```

```
    cache->flags            = flags;
    cache->next_colour      = 0;
    cache->working_set      = SLAB_DEFAULT_WORKING_SET;

    /* Compute actual alignment. */
    if(alignment == 0) {
        cache->alignment = sizeof(uint32_t);
    }
    else {
        cache->alignment = alignment;
    }

    if((flags & SLAB_HWCACHE_ALIGN) && cache->alignment < cpu_info.dcache_alignment) {
        cache->alignment = cpu_info.dcache_alignment;
    }

    cache->alignment = ALIGN_END(cache->alignment, sizeof(uint32_t));

    /* Reserve space for bufctl and/or redzone word. */
    cache->obj_size = ALIGN_END(size, sizeof(uint32_t));

    if((flags & SLAB_POISON) && (flags & SLAB_RED_ZONE)) {
        /* bufctl and redzone word appended to buffer */
        cache->alloc_size = cache->obj_size + sizeof(uint32_t) + sizeof(slab_bufctl_t);
    }
    else if((flags & SLAB_POISON) || (flags & SLAB_RED_ZONE)) {
        /* bufctl or redzone word appended to buffer (can be shared) */
        cache->alloc_size = cache->obj_size + sizeof(uint32_t);
    }
    else if(ctor != NULL && ! (flags & SLAB_COMPACT)) {
        /* If a constructor is defined, we cannot put the bufctl inside
         * the object because that could overwrite constructed state,
         * unless client explicitly says it's ok (SLAB_COMPACT flag). */
        cache->alloc_size = cache->obj_size + sizeof(slab_bufctl_t);
    }
    else {
        cache->alloc_size = cache->obj_size;
    }

    if(cache->alloc_size % cache->alignment != 0) {
        cache->alloc_size += cache->alignment - cache->alloc_size % cache->alignment;
    }

    size_t avail_space = SLAB_SIZE - sizeof(slab_t);

    unsigned int buffers_per_slab = avail_space / cache->alloc_size;

    size_t wasted_space = avail_space - buffers_per_slab * cache->alloc_size;

    cache->max_colour = (wasted_space / cache->alignment) * cache->alignment;

    cache->bufctl_offset = cache->alloc_size - sizeof(slab_bufctl_t);

    /* Allocate first slab.
     *
     * This is needed to allow a few objects to be allocated during kernel
     * initialization. */
    init_and_add_slab(cache, boot_page_alloc(boot_alloc));
}
```

Here is the call graph for this function:



4.77.3.4 void slab_cache_reap (slab_cache_t ∗ *cache*)

Return memory to the page allocator.

Free slabs in excess to the cache's working set are finalized and freed.

Parameters

cache	the cache from which to reclaim memory

Definition at line 544 of file slab.c.

References slab_cache_t::empty_count, slab_t::next, slab_cache_t::slabs_empty, and slab_cache_t::working_set.

```
544                                         {
545     while(cache->empty_count > cache->working_set) {
546         /* select the first empty slab */
547         slab_t *slab = cache->slabs_empty;
548 
549         /* unlink it and update count */
550         cache->slabs_empty  = slab->next;
551         cache->empty_count -= 1;
552 
553         /* destroy slab */
554         destroy_slab(cache, slab);
555     }
556 }
```

4.77.3.5 void slab_cache_set_working_set (slab_cache_t ∗ *cache*, unsigned int *n*)

Set a cache's working set.

The working set is defined as the number of free slabs the cache keeps for itself when pages are reclaimed from it. (This is terminology used in the Bonwick paper.) This provides some hysteresis to prevent slabs from being continuously created and destroyed, which requires calling the constructor and destructor functions on individual objects on the slabs.

Parameters

cache	the cache for which to set the working set
n	the size of the working set (number of pages)

Definition at line 571 of file slab.c.

References slab_cache_t::working_set.

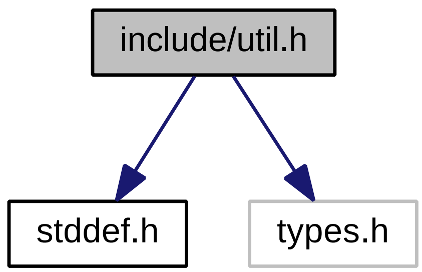
```
                                                                     {
        cache->working_set = n;
}
```

4.77.4 Variable Documentation

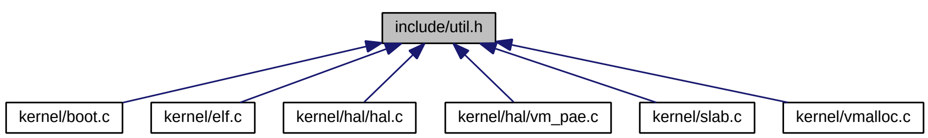
4.77.4.1 slab_cache_t∗ slab_cache_list

4.78 include/util.h File Reference

`#include <stddef.h>`
`#include <types.h>`
Include dependency graph for util.h:



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



Macros

- #define **ALIGN_START**(x, s) ((x) & ∼((s)-1))
- #define **ALIGN_END**(x, s) (**ALIGN_START**((x) + s - 1, (s)))
- #define **OFFSET_OF_PTR**(x, s) ((**uintptr_t**)(x) & ((s)-1))
- #define **ALIGN_START_PTR**(x, s) ((void ∗)**ALIGN_START**((**uintptr_t**)(x), (s)))
- #define **ALIGN_END_PTR**(x, s) ((void ∗)**ALIGN_END**((**uintptr_t**)(x), (s)))
- #define **alloc_forward**(T, p) ((T ∗)alloc_forward_func(sizeof(T), &(p)))
- #define **alloc_backward**(T, p) ((T ∗)alloc_forward_func(sizeof(T), &(p)))

4.78.1 Macro Definition Documentation

4.78.1.1 #define ALIGN_END(*x*, *s*) (**ALIGN_START**((x) + s - 1, (s)))

Definition at line 40 of file util.h.

Referenced by slab_cache_init().

4.78.1.2 #define ALIGN_END_PTR(*x,* *s*) ((void ∗)ALIGN_END((uintptr_t)(x), (s)))

Definition at line 46 of file util.h.

Referenced by boot_heap_alloc_size(), and elf_load().

4.78.1.3 #define ALIGN_START(*x,* *s*) ((x) & ∼((s)-1))

Definition at line 38 of file util.h.

4.78.1.4 #define ALIGN_START_PTR(*x,* *s*) ((void ∗)ALIGN_START((uintptr_t)(x), (s)))

Definition at line 44 of file util.h.

Referenced by elf_load(), and slab_cache_free().

4.78.1.5 #define alloc_backward(*T,* *p*) ((T ∗)alloc_forward_func(sizeof(T), &(p)))

Definition at line 67 of file util.h.

4.78.1.6 #define alloc_forward(*T,* *p*) ((T ∗)alloc_forward_func(sizeof(T), &(p)))

Definition at line 65 of file util.h.

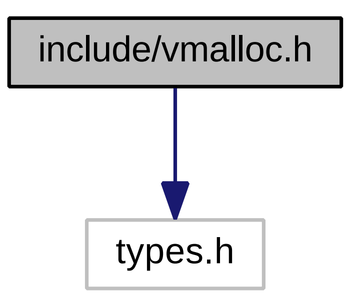
4.78.1.7 #define OFFSET_OF_PTR(*x,* *s*) ((uintptr_t)(x) & ((s)-1))

Definition at line 42 of file util.h.

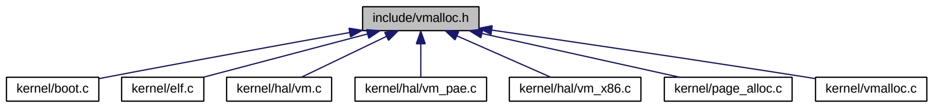
4.79 include/vmalloc.h File Reference

`#include <types.h>`

Include dependency graph for vmalloc.h:



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



Functions

- **addr_t vmalloc** (void)

  Allocate a page of virtual address space.
- void **vmfree** (**addr_t** page)

  Free a page of virtual address space.
- void **vmalloc_init** (**addr_t** start_addr, **addr_t** end_addr, **addr_t** preinit_limit, **boot_alloc_t** ∗boot_alloc)

  Basic initialization of the virtual memory allocator.
- **bool vmalloc_is_in_range** (**addr_t** page)

  Check whether the specified page is in the region managed by the allocator.

4.79.1 Function Documentation

4.79.1.1 addr_t vmalloc (void)

Allocate a page of virtual address space.

Returns

address of allocated page or NULL if allocation failed

Definition at line 150 of file vmalloc.c.

Referenced by add_page_frame(), boot_page_alloc(), elf_setup_stack(), vm_clone_page_directory(), vm_destroy_-page_directory(), vm_pae_lookup_page_directory(), and vm_x86_lookup_page_directory().

```
150                        {
151     return vmalloc_from(&kernel_vmallocator);
152 }
```

4.79.1.2 void vmalloc_init (addr_t *start_addr,* addr_t *end_addr,* addr_t *preinit_limit,* boot_alloc_t ∗ *boot_alloc*)

Basic initialization of the virtual memory allocator.

This function initializes the allocator structure, and then initializes the first few blocks up to the limit set by the preinit_limit argument (more precisely, up to and including the block that contains preinit_limit - 1). TODO mention how to initialize the rest once this is implemented.

Parameters

start_addr	the start address of the region managed by the allocator

end_addr	the end address of the region managed by the allocator
preinit_limit	the limit address for preinitialized blocks
boot_alloc	the initialization-time page allocator structure

Definition at line 178 of file vmalloc.c.

Referenced by vm_boot_postinit().

```
182                                     {
183
184     init_allocator(
185             &kernel_vmallocator,
186             start_addr,
187             end_addr,
188             preinit_limit,
189             boot_alloc);
190 }
```

4.79.1.3 bool vmalloc_is_in_range (addr_t *page*)

Check whether the specified page is in the region managed by the allocator.

Parameters

page	the address of the page

Returns

true if it is in the region, false otherwise

Definition at line 199 of file vmalloc.c.

Referenced by remove_page_frame().

```
199                                      {
200     return addr_is_in_initialized_range(&kernel_vmallocator, page);
201 }
```

4.79.1.4 void vmfree (addr_t *page*)

Free a page of virtual address space.

Parameters

page	the address of the page to free

Definition at line 160 of file vmalloc.c.

Referenced by elf_setup_stack(), and remove_page_frame().

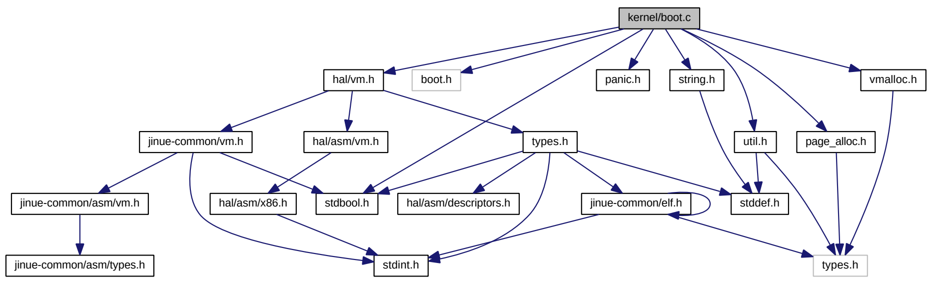
```
160                         {
161     vmfree_to(&kernel_vmallocator, page);
162 }
```

4.80 kernel/boot.c File Reference

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

```
#include <boot.h>
#include <page_alloc.h>
#include <panic.h>
#include <stdbool.h>
#include <string.h>
#include <vmalloc.h>
#include <util.h>
```

Include dependency graph for boot.c:



Functions

- void **boot_alloc_init** (**boot_alloc_t** ∗boot_alloc, void ∗heap_ptr)

  Initialize the boot allocator.
- void ∗ **boot_heap_alloc_size** (**boot_alloc_t** ∗boot_alloc, **size_t** size, **size_t** align)

  Allocate an object on the boot heap.
- void **boot_heap_push** (**boot_alloc_t** ∗boot_alloc)

  Push the current state of the boot allocator heap.
- void **boot_heap_pop** (**boot_alloc_t** ∗boot_alloc)

  Pop the last pushed boot allocator heap.
- **addr_t boot_page_alloc_early** (**boot_alloc_t** ∗boot_alloc)

  Early page allocation.
- **kern_paddr_t boot_page_frame_alloc** (**boot_alloc_t** ∗boot_alloc)

  Allocate a page frame, that is, a page of physical memory.
- **addr_t boot_vmalloc** (**boot_alloc_t** ∗boot_alloc)

  Allocate a page of address space.
- **addr_t boot_page_alloc** (**boot_alloc_t** ∗boot_alloc)

  Allocate a page in the allocations region of the kernel address space.
- **addr_t boot_page_alloc_image** (**boot_alloc_t** ∗boot_alloc)

  Allocate a page in the image region of the kernel address space.

4.80.1 Function Documentation

4.80.1.1 void boot_alloc_init (boot_alloc_t ∗ *boot_alloc*, void ∗ *heap_ptr*)

Initialize the boot allocator.

The boot allocator is used for heap and page allocation during kernel initialization. After this function is called, the boot heap is ready to use (see **boot_heap_alloc()** (p. 66)). However, the page and page frame allocators require additional initialization by the machine-dependent code before they can be used.

Parameters

boot_alloc	the allocator state initialized by this function
heap_ptr	the current top of the boot heap

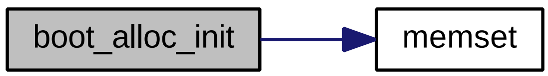
Definition at line 54 of file boot.c.

References boot_alloc_t::heap_ptr, boot_alloc_t::its_early, and memset().

Referenced by kmain().

```
54                                                                 {
55     memset(boot_alloc, 0, sizeof(boot_alloc_t));
56     boot_alloc->heap_ptr    = heap_ptr;
57     boot_alloc->its_early   = true;
58     /* TODO handle heap limit. */
59 }
```

Here is the call graph for this function:



4.80.1.2 void∗ boot_heap_alloc_size (boot_alloc_t ∗ *boot_alloc,* size_t *size,* size_t *align*)

Allocate an object on the boot heap.

Callers do not call this function directly but instead use the **boot_heap_alloc()** (p. 66) macro that takes a type as the second argument instead of an object size.

Parameters

boot_alloc	the boot allocator state
size	the size of the object to allocate, in bytes
align	the required start address alignment of the object, zero for no constraint

Returns

the allocated object

Definition at line 73 of file boot.c.

References ALIGN_END_PTR, and boot_alloc_t::heap_ptr.

```
73                                                                                  {
74     if(align != 0) {
75         boot_alloc->heap_ptr = ALIGN_END_PTR(boot_alloc->heap_ptr, align);
76     }
77
78     void *object            = boot_alloc->heap_ptr;
79     boot_alloc->heap_ptr    = (char *)boot_alloc->heap_ptr + size;
80
81     return object;
82 }
```

4.80.1.3 void boot_heap_pop (boot_alloc_t ∗ *boot_alloc*)

Pop the last pushed boot allocator heap.

This function frees all heap allocations performed since the matching call to **boot_heap_push()** (p. 68).

Parameters

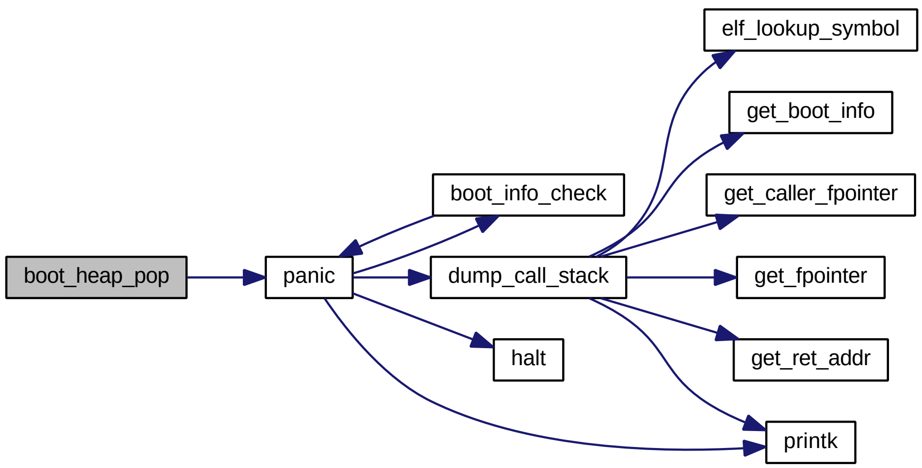
boot_alloc	the boot allocator state

Definition at line 112 of file boot.c.

References boot_alloc_t::heap_ptr, boot_alloc_t::heap_pushed_state, boot_heap_pushed_state::next, NULL, and panic().

```
112                                             {
113     if(boot_alloc->heap_pushed_state == NULL) {
114         panic("No more boot heap pushed state to pop.");
115     }
116 
117     boot_alloc->heap_ptr             = boot_alloc->heap_pushed_state;
118     boot_alloc->heap_pushed_state    = boot_alloc->heap_pushed_state->next;
119 }
```

Here is the call graph for this function:



4.80.1.4 void boot_heap_push (**boot_alloc_t** ∗ *boot_alloc*)

Push the current state of the boot allocator heap.

All heap allocations performed after calling this function are freed by the matching call to **boot_heap_pop()** (p. 67). This function can be called multiple times before calling **boot_heap_pop()** (p. 67). Heap states pushed by this function are popped by **boot_heap_pop()** (p. 67)() in the reverse order they were pushed.

Parameters

boot_alloc	the boot allocator state

Definition at line 95 of file boot.c.

References boot_heap_alloc, boot_alloc_t::heap_pushed_state, and boot_heap_pushed_state::next.

```
95                                              {
96     struct boot_heap_pushed_state *pushed_state =
97             boot_heap_alloc(boot_alloc, struct boot_heap_pushed_state, 0);
98 
99     pushed_state->next               = boot_alloc->heap_pushed_state;
100     boot_alloc->heap_pushed_state    = pushed_state;
101 }
```

4.80.1.5 addr_t boot_page_alloc (boot_alloc_t ∗ *boot_alloc*)

Allocate a page in the allocations region of the kernel address space.

The physical memory is allocated just after the kernel image and other initialization-time allocations by calling **boot_page_frame_alloc()** (p. 73) whereas the address space page is allocated in the allocations region by calling **vmalloc()** (p. 275).

If either of these two conditions is met, you must use boot_pgalloc_image() instead of this function: 1) The address space page allocator has not not yet been initialized by calling **vmalloc_init()** (p. 275); or 2) It is necessary to allocate multiple contiguous pages.

Parameters

boot_alloc	the boot allocator state

Returns

address of allocated page

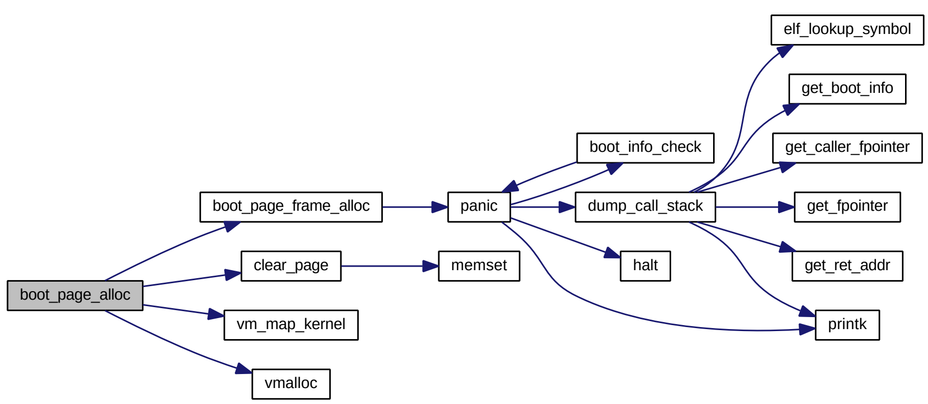
Definition at line 282 of file boot.c.

References boot_page_frame_alloc(), clear_page(), VM_FLAG_READ_WRITE, vm_map_kernel(), and vmalloc().

Referenced by hal_init(), slab_cache_init(), and thread_create_boot().

```
282                                              {
283     kern_paddr_t paddr  = boot_page_frame_alloc(boot_alloc);
284     addr_t vaddr        = vmalloc();
285
286     vm_map_kernel(vaddr, paddr, VM_FLAG_READ_WRITE);
287
288     /* This newly allocated page may have data left from a previous boot which
289      * may contain sensitive information. Let's clear it. */
290     clear_page(vaddr);
291
292     return vaddr;
293 }
```

Here is the call graph for this function:



4.80.1.6 addr_t boot_page_alloc_early (boot_alloc_t ∗ *boot_alloc*)

Early page allocation.

When the kernel is first entered, the setup code has set up temporary page tables that map a contiguous region of physical memory (RAM) that contains the kernel image at KLIMIT. The setup code itself allocates a few pages, notably for the temporary page tables and for the boot stack and heap. These pages are allocated sequentially just after the kernel image.

This function allocates pages sequentially following the kernel image and the setup code allocations. It is meant to be called early in the initialization process, while the temporary page tables set up by the setup code are still being used, which means before the kernel switches to the initial address space it sets up.

Because the page tables set up by the setup code are being used, there is a fixed relation between the virtual address of the pages allocated by this function and the physical address of the underlying page frames. This relation is expressed by the **EARLY_PTR_TO_PHYS_ADDR()** (p. 142) macro.

This function must not be called once the kernel has switched away from the page tables set up by the setup code to the initial address space it has set up itself. This function checks for this and triggers a kernel panic if it happens.

Parameters

boot_alloc	the boot allocator state

Returns

address of allocated page

Definition at line 150 of file boot.c.

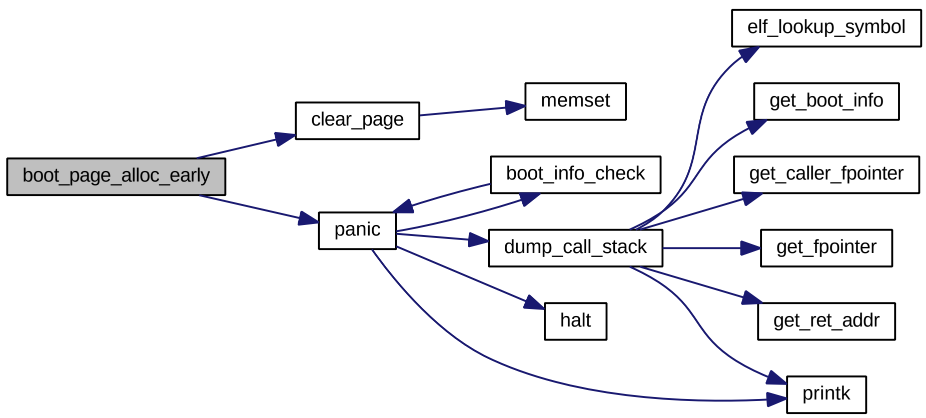
References clear_page(), EARLY_PTR_TO_PHYS_ADDR, boot_alloc_t::its_early, boot_alloc_t::kernel_paddr_limit, boot_alloc_t::kernel_paddr_top, boot_alloc_t::kernel_vm_limit, boot_alloc_t::kernel_vm_top, NULL, PAGE_MASK, P-AGE_SIZE, and panic().

Referenced by vm_init_initial_page_directory(), vm_pae_create_initial_addr_space(), and vm_x86_create_initial_addr-_space().

```
150                                                         {
151     /* Preconditions */
152     if(! boot_alloc->its_early) {
153         panic("boot_pgalloc_early() called too late");
154     }
155 
156     if(boot_alloc->kernel_vm_top == NULL) {
157         panic("boot_pgalloc_early(): allocator is uninitialized");
158     }
159 
160     if(((uintptr_t)boot_alloc->kernel_vm_top & PAGE_MASK) != 0) {
161         panic("boot_pgalloc_early(): bad kernel region top VM address alignment");
162     }
163 
164     if(boot_alloc->kernel_paddr_top != EARLY_PTR_TO_PHYS_ADDR(boot_alloc->
      kernel_vm_top)) {
165         panic("boot_pgalloc_early(): inconsistent allocator state");
166     }
167 
168     /* address of allocated page */
169     addr_t allocated_page = boot_alloc->kernel_vm_top;
170 
171     /* Update allocator state.
172      *
173      * In this early allocator function that is called while the temporary page
174      * tables set up by the setup code are still being used, there is a fixed
175      * relationship between virtual and physical addresses. */
176     boot_alloc->kernel_vm_top      = allocated_page + PAGE_SIZE;
177     boot_alloc->kernel_paddr_top   = boot_alloc->kernel_paddr_top + PAGE_SIZE;
178 
179     /* Check updated state against allocation limits. */
180     if(boot_alloc->kernel_vm_top > boot_alloc->kernel_vm_limit) {
181         panic("vmalloc_early(): kernel address space exhausted");
182     }
183 
184     if(boot_alloc->kernel_paddr_top > boot_alloc->kernel_paddr_limit) {
185         panic("vmalloc_early(): available memory exhausted");
```

```
    }

    /* This newly allocated page may have data left from a previous boot which
     * may contain sensitive information. Let's clear it. */
    clear_page(allocated_page);

    /* Post-condition */
    if(boot_alloc->kernel_paddr_top != EARLY_PTR_TO_PHYS_ADDR(boot_alloc->
  kernel_vm_top)) {
        panic("boot_pgalloc_early(): inconsistent allocator state on return");
    }

    return allocated_page;
}
```

Here is the call graph for this function:



4.80.1.7 addr_t boot_page_alloc_image (boot_alloc_t ∗ *boot_alloc*)

Allocate a page in the image region of the kernel address space.

Since the size of the image region is limited, use boot_pgalloc() instead of this function whenever possible.

The difference between this function and boot_pgalloc() is that the address space page is allocated by this function using **boot_vmalloc()** (p. 74) instead of **vmalloc()** (p. 275). Pages allocated by subsequent calls to this function are allocated sequentially.

Parameters

boot_alloc	the boot allocator state

Returns

address of allocated page

Definition at line 310 of file boot.c.

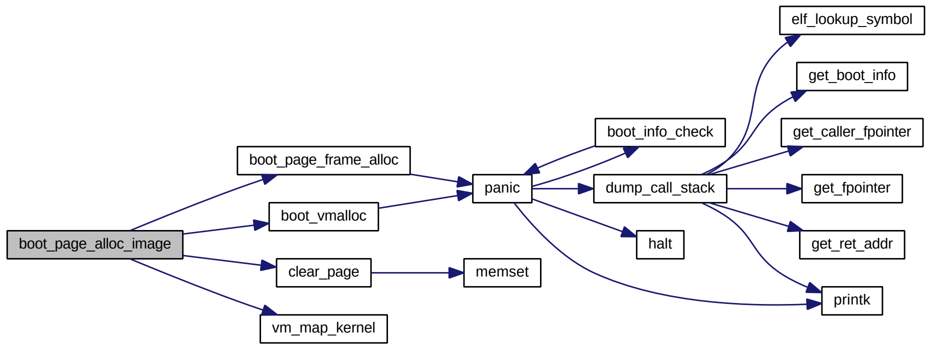
References boot_page_frame_alloc(), boot_vmalloc(), clear_page(), VM_FLAG_READ_WRITE, and vm_map_kernel().

Referenced by hal_init().

```
310                                                         {
311     kern_paddr_t paddr  = boot_page_frame_alloc(boot_alloc);
312     addr_t vaddr        = boot_vmalloc(boot_alloc);
313
```

```
314     vm_map_kernel(vaddr, paddr, VM_FLAG_READ_WRITE);
315
316     /* This newly allocated page may have data left from a previous boot which
317      * may contain sensitive information. Let's clear it. */
318     clear_page(vaddr);
319
320     return vaddr;
321 }
```

Here is the call graph for this function:



4.80.1.8 kern_paddr_t boot_page_frame_alloc (boot_alloc_t ∗ *boot_alloc*)

Allocate a page frame, that is, a page of physical memory.

The allocated page frame is not mapped anywhere. For a mapped page, call boot_pgalloc() or boot_pgalloc_image() instead;

Parameters

boot_alloc	the boot allocator state

Returns

physical address of allocated page frame

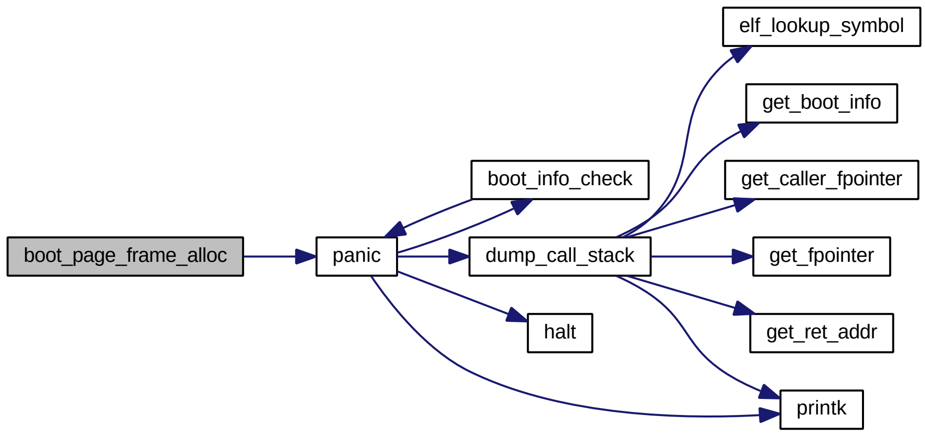
Definition at line 210 of file boot.c.

References boot_alloc_t::its_early, boot_alloc_t::kernel_paddr_limit, boot_alloc_t::kernel_paddr_top, PAGE_SIZE, and panic().

Referenced by boot_page_alloc(), boot_page_alloc_image(), elf_load(), and elf_setup_stack().

```
210                                                             {
211     if(boot_alloc->its_early) {
212         panic("boot_pfalloc() called too soon");
213     }
214
215     /* address of allocated page  */
216     kern_paddr_t paddr = boot_alloc->kernel_paddr_top;
217
218     /* Update allocator state. */
219     boot_alloc->kernel_paddr_top = paddr + PAGE_SIZE;
220
221     /* Check bounds. */
222     if(boot_alloc->kernel_paddr_top > boot_alloc->kernel_paddr_limit) {
223         panic("pfalloc_boot(): available memory exhausted");
224     }
225
226     return paddr;
227 }
```

Here is the call graph for this function:



4.80.1.9 addr_t boot_vmalloc (boot_alloc_t ∗ *boot_alloc*)

Allocate a page of address space.

No memory is mapped to the allocated page. The page is allocated from the image region of the kernel address space, just after the kernel image and other initialization-time page allocations. This function allocates pages sequentially.

Parameters

boot_alloc	the boot allocator state

Returns

address of allocated page

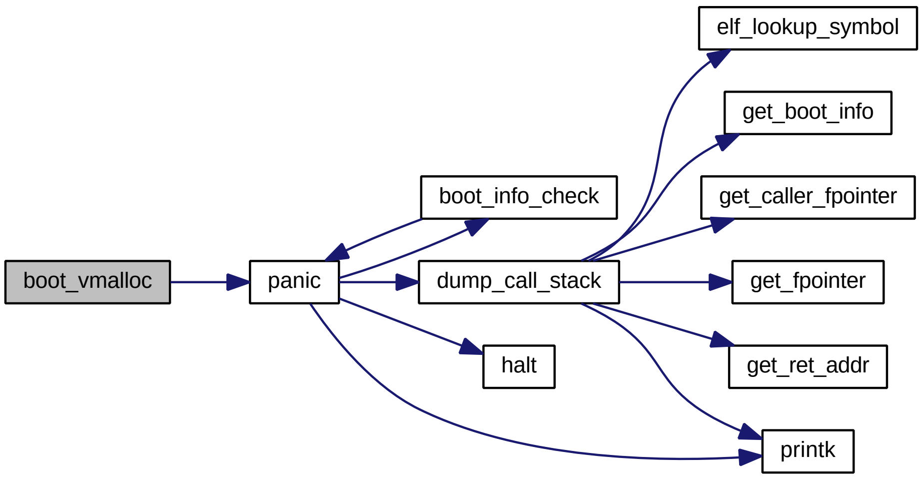
Definition at line 241 of file boot.c.

References boot_alloc_t::its_early, boot_alloc_t::kernel_vm_limit, boot_alloc_t::kernel_vm_top, NULL, PAGE_SIZE, and panic().

Referenced by boot_page_alloc_image(), and vm_boot_init().

```
241                                           {
242     if(boot_alloc->its_early) {
243         panic("boot_vmalloc() called too soon");
244     }
245 
246     if(boot_alloc->kernel_vm_top == NULL) {
247         panic("boot_pgalloc_early(): allocator is uninitialized");
248     }
249 
250     /* address of allocated page */
251     addr_t page = boot_alloc->kernel_vm_top;
252 
253     /* Update allocator state. */
254     boot_alloc->kernel_vm_top = page + PAGE_SIZE;
255 
256     /* Check bounds. */
257     if(boot_alloc->kernel_vm_top > boot_alloc->kernel_vm_limit) {
258         panic("vmalloc_boot(): kernel address space exhausted");
259     }
260 
261     return page;
262 }
```

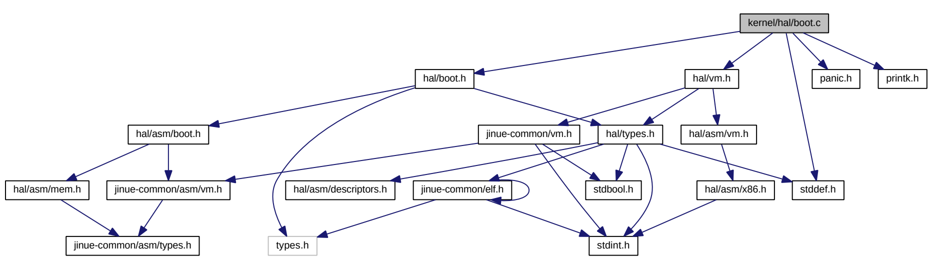
Here is the call graph for this function:



4.81 kernel/hal/boot.c File Reference

```
#include <hal/boot.h>
#include <hal/vm.h>
#include <panic.h>
#include <printk.h>
#include <stddef.h>
```

Include dependency graph for boot.c:



Functions

- **bool boot_info_check** (**bool** panic_on_failure)
- const **boot_info_t** ∗ **get_boot_info** (void)
- void **boot_info_dump** (void)

Variables

- const **boot_info_t** ∗ **boot_info**

4.81.1 Function Documentation

4.81.1.1 bool boot_info_check (bool *panic_on_failure*)

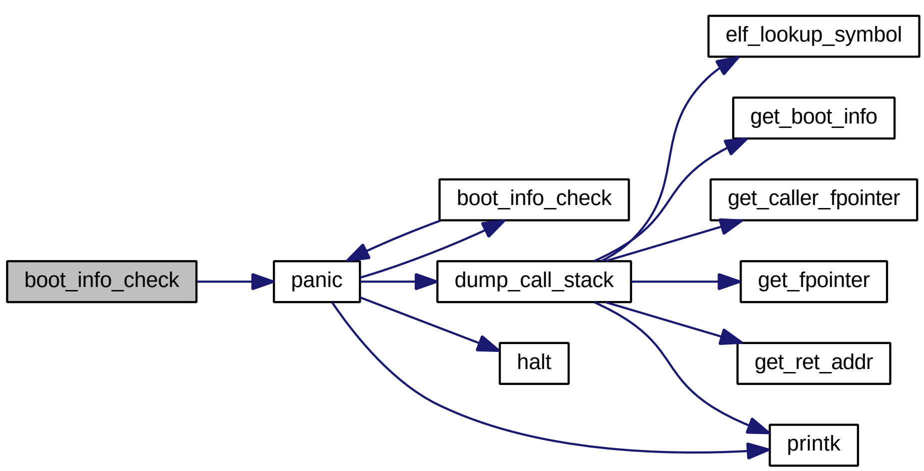
Definition at line 42 of file boot.c.

References BOOT_SETUP_MAGIC, boot_info_t::image_start, boot_info_t::kernel_start, NULL, page_offset_of, panic(), and boot_info_t::setup_signature.

Referenced by kmain(), and panic().

```
42                                                  {
43      const char *error_description = NULL;
44
45      /* This data structure is accessed early during the boot process, before
46       * paging is enabled. What this means is that, if boot_info is NULL and we
47       * dereference it, it does *not* cause a page fault or any other CPU
48       * exception. */
49      if(boot_info == NULL) {
50          error_description = "Boot information structure pointer is NULL.";
51      }
52      else if(boot_info->setup_signature != BOOT_SETUP_MAGIC) {
53          error_description = "Bad setup header signature.";
54      }
55      else if(page_offset_of(boot_info->image_start) != 0) {
56          error_description = "Bad image alignment.";
57      }
58      else if(page_offset_of(boot_info->kernel_start) != 0) {
59          error_description = "Bad kernel alignment.";
60      }
61      else {
62          return true;
63      }
64
65      if(panic_on_failure) {
66          panic(error_description);
67      }
68
69      return false;
70 }
```

Here is the call graph for this function:



4.81.1.2 void boot_info_dump (void)

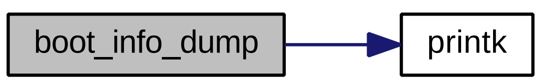
Definition at line 76 of file boot.c.

References boot_info_t::boot_end, boot_info_t::boot_heap, boot_info_t::e820_entries, boot_info_t::e820_map, boot_info_t::image_start, boot_info_t::image_top, boot_info_t::kernel_size, boot_info_t::kernel_start, boot_info_t::page_-

directory, boot_info_t::page_table, printk(), boot_info_t::proc_size, boot_info_t::proc_start, and boot_info_t::setup_-signature.

```
76                          {
77      printk("Boot information structure:\n");
78      printk("    kernel_start    %x  %u\n", boot_info->kernel_start    , boot_info->kernel_start    );
79      printk("    kernel_size     %x  %u\n", boot_info->kernel_size     , boot_info->kernel_size     );
80      printk("    proc_start      %x  %u\n", boot_info->proc_start      , boot_info->proc_start      );
81      printk("    proc_size       %x  %u\n", boot_info->proc_size       , boot_info->proc_size       );
82      printk("    image_start     %x  %u\n", boot_info->image_start     , boot_info->image_start     );
83      printk("    image_top       %x  %u\n", boot_info->image_top       , boot_info->image_top       );
84      printk("    e820_entries    %x  %u\n", boot_info->e820_entries    , boot_info->e820_entries    );
85      printk("    e820_map        %x  %u\n", boot_info->e820_map        , boot_info->e820_map        );
86      printk("    boot_heap       %x  %u\n", boot_info->boot_heap       , boot_info->boot_heap       );
87      printk("    boot_end        %x  %u\n", boot_info->boot_end        , boot_info->boot_end        );
88      printk("    page_table      %x  %u\n", boot_info->page_table      , boot_info->page_table      );
89      printk("    page_directory  %x  %u\n", boot_info->page_directory  , boot_info->page_directory  );
90      printk("    setup_signature %x  %u\n", boot_info->setup_signature, boot_info->setup_signature );
91 }
```

Here is the call graph for this function:



4.81.1.3 const **boot_info_t**∗ get_boot_info (void)

Definition at line 72 of file boot.c.

References boot_info.

Referenced by dispatch_syscall(), dump_call_stack(), and kmain().

```
72                                        {
73      return boot_info;
74 }
```

4.81.2 Variable Documentation

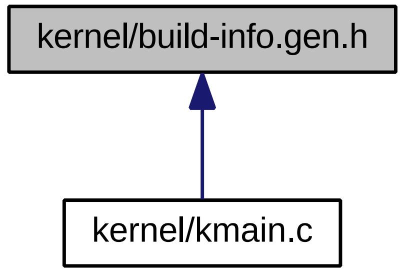
4.81.2.1 const **boot_info_t**∗ boot_info

Definition at line 40 of file boot.c.

Referenced by dispatch_syscall(), dump_call_stack(), and get_boot_info().

4.82 kernel/build-info.gen.h File Reference

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



Macros

- #define **GIT_REVISION** "2a99f29"
- #define **BUILD_TIME** "Wed Mar 20 21:01:51 EDT 2019"
- #define **BUILD_HOST** "raskolnikov"

4.82.1 Macro Definition Documentation

4.82.1.1 #define BUILD_HOST "raskolnikov"

Definition at line 9 of file build-info.gen.h.

Referenced by kmain().

4.82.1.2 #define BUILD_TIME "Wed Mar 20 21:01:51 EDT 2019"

Definition at line 8 of file build-info.gen.h.

Referenced by kmain().

4.82.1.3 #define GIT_REVISION "2a99f29"

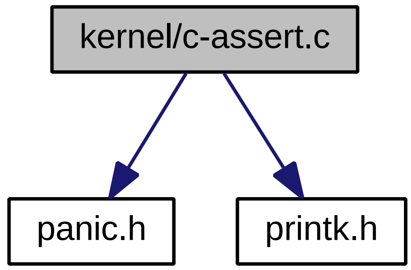
Definition at line 7 of file build-info.gen.h.

Referenced by kmain().

4.83 kernel/c-assert.c File Reference

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

Include dependency graph for c-assert.c:



Functions

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

4.83.1 Function Documentation

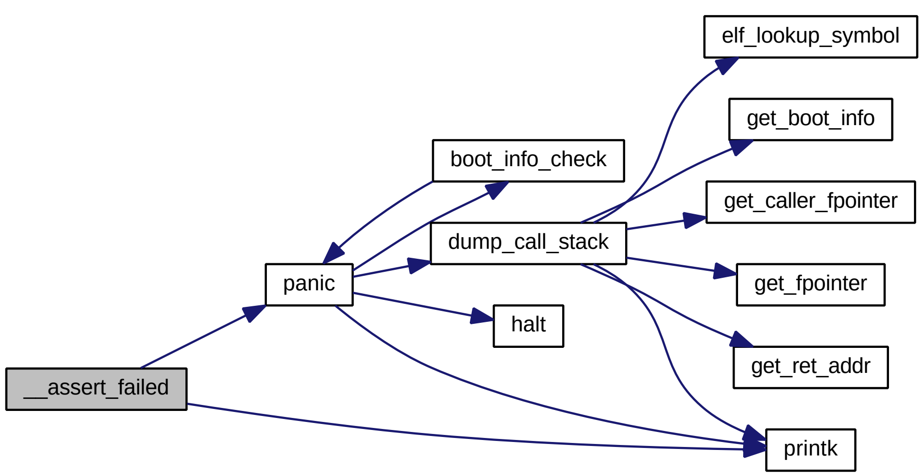
4.83.1.1 void __assert_failed (const char ∗ *expr,* const char ∗ *file,* unsigned int *line,* const char ∗ *func*)

Definition at line 36 of file c-assert.c.

References panic(), and printk().

```
                        {

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

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

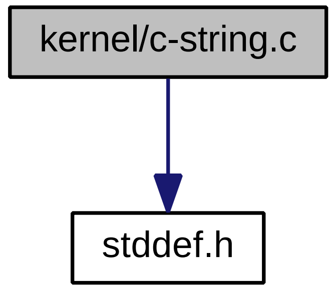
Here is the call graph for this function:



4.84 kernel/c-string.c File Reference

`#include <stddef.h>`

Include dependency graph for c-string.c:



Functions

- void ∗ **memset** (void ∗s, int c, **size_t** n)
- void ∗ **memcpy** (void ∗dest, const void ∗src, **size_t** n)
- **size_t strlen** (const char ∗s)

4.84.1 Function Documentation

4.84.1.1 void∗ memcpy (void ∗ *dest,* const void ∗ *src,* size_t *n*)

Definition at line 45 of file c-string.c.

Referenced by ipc_receive(), ipc_reply(), ipc_send(), and process_create_initial().

```
45                                                      {
46     size_t       idx;
47     char        *cdest  = dest;
48     const char  *csrc   = src;
49
50     for(idx = 0; idx < n; ++idx) {
51         cdest[idx] = csrc[idx];
52     }
53
54     return dest;
55 }
```

4.84.1.2 void∗ memset (void ∗ *s,* int *c,* size_t *n*)

Definition at line 34 of file c-string.c.

Referenced by boot_alloc_init(), clear_page(), cpu_init_data(), thread_page_init(), and vm_pae_lookup_page_-directory().

```
34                                         {
35     size_t   idx;
36     char    *cs = s;
37
38     for(idx = 0; idx < n; ++idx) {
39         cs[idx] = c;
40     }
```

```

    return s;
}
```

4.84.1.3 size_t strlen (const char ∗ *s*)

Definition at line 57 of file c-string.c.

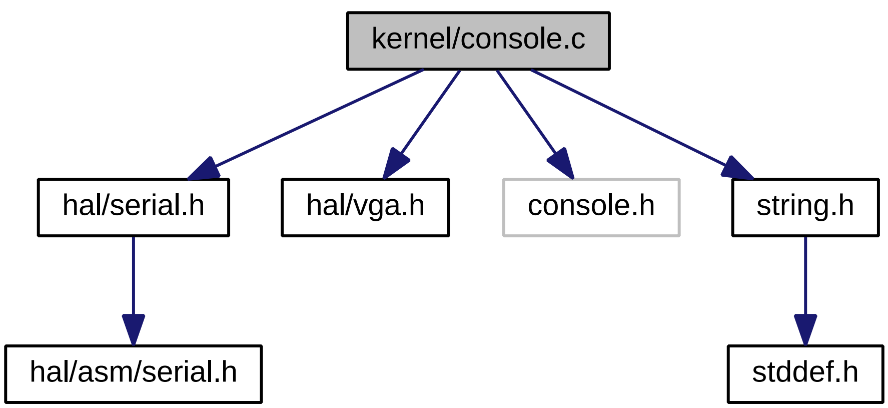
Referenced by console_print().

```
57                               {
58     size_t count = 0;
59
60     while(*s != 0) {
61         ++s;
62         ++count;
63     }
64
65     return count;
66 }
```

4.85 kernel/console.c File Reference

```
#include <hal/serial.h>
#include <hal/vga.h>
#include <console.h>
#include <string.h>
```

Include dependency graph for console.c:



Functions

- void **console_init** (void)
- void **console_printn** (const char ∗message, unsigned int n, int colour)
- void **console_putc** (char c, int colour)
- void **console_print** (const char ∗message, int colour)

4.85.1 Function Documentation

4.85.1.1 void console_init (void)

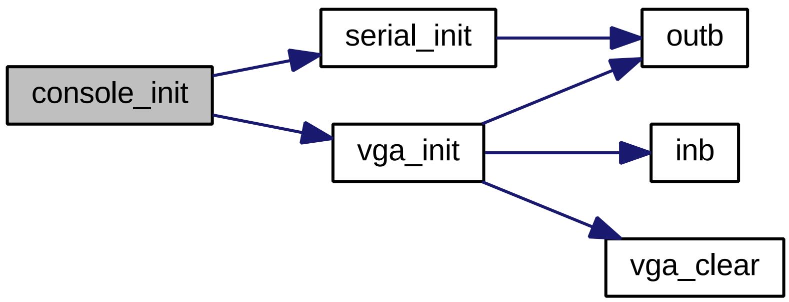
Definition at line 38 of file console.c.

References CONSOLE_SERIAL_BAUD_RATE, CONSOLE_SERIAL_IOPORT, serial_init(), and vga_init().

Referenced by kmain().

```
38                      {
39     vga_init();
40     serial_init(CONSOLE_SERIAL_IOPORT, CONSOLE_SERIAL_BAUD_RATE);
41 }
```

Here is the call graph for this function:



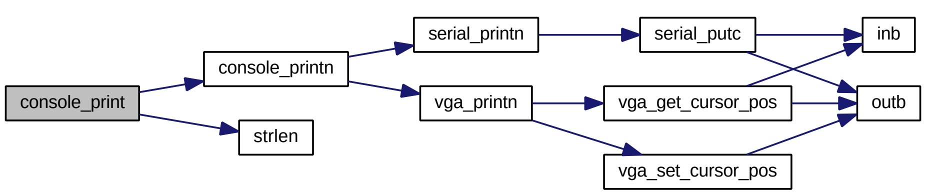
4.85.1.2 void console_print (const char ∗ *message,* int *colour*)

Definition at line 53 of file console.c.

References console_printn(), and strlen().

```
53                                              {
54     console_printn(message, strlen(message), colour);
55 }
```

Here is the call graph for this function:



4.85.1.3 void console_printn (const char ∗ *message,* unsigned int *n,* int *colour*)

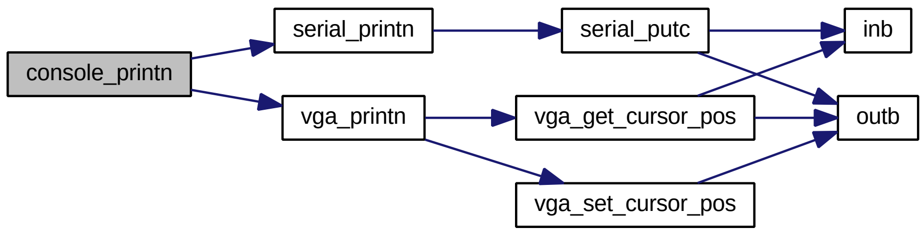
Definition at line 43 of file console.c.

References CONSOLE_SERIAL_IOPORT, serial_printn(), and vga_printn().

Referenced by console_print(), and dispatch_syscall().

```
43                                                                  {
44     vga_printn(message, n, colour);
45     serial_printn(CONSOLE_SERIAL_IOPORT, message, n);
46 }
```

Here is the call graph for this function:



4.85.1.4 void console_putc (char *c*, int *colour*)

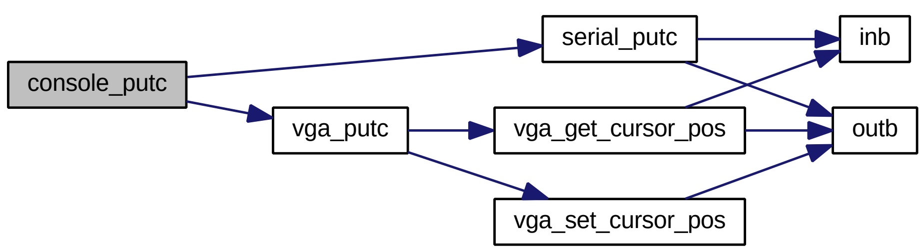
Definition at line 48 of file console.c.

References CONSOLE_SERIAL_IOPORT, serial_putc(), and vga_putc().

Referenced by dispatch_syscall().

```
48                                         {
49     vga_putc(c, colour);
50     serial_putc(CONSOLE_SERIAL_IOPORT, c);
51 }
```

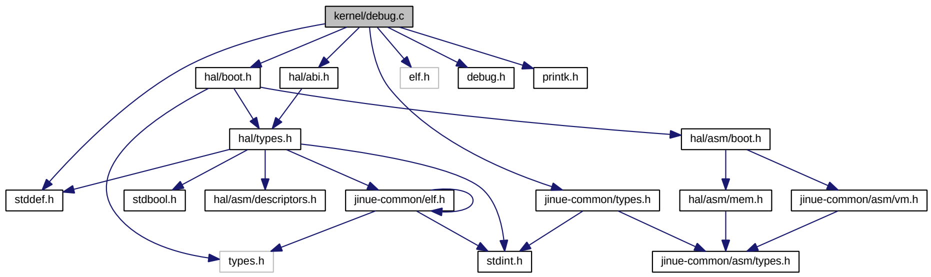
Here is the call graph for this function:



4.86 kernel/debug.c File Reference

```
#include <jinue-common/types.h>
#include <hal/abi.h>
#include <hal/boot.h>
#include <elf.h>
#include <stddef.h>
#include <debug.h>
#include <printk.h>
```

Include dependency graph for debug.c:



Functions

- void **dump_call_stack** (void)

4.86.1 Function Documentation

4.86.1.1 void dump_call_stack (void)

Definition at line 41 of file debug.c.

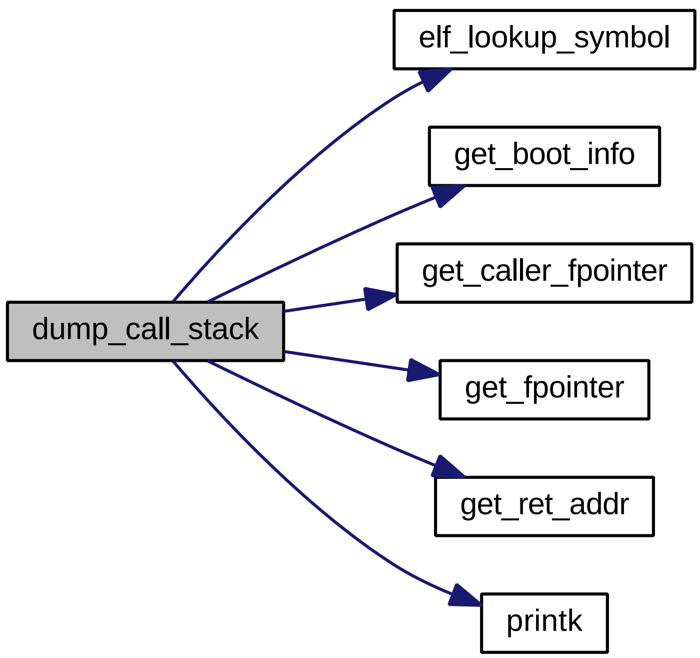
References elf_symbol_t::addr, boot_info, elf_lookup_symbol(), get_boot_info(), get_caller_fpointer(), get_fpointer(), get_ret_addr(), boot_info_t::kernel_start, elf_symbol_t::name, NULL, printk(), and STT_FUNCTION.

Referenced by panic().

```
41                                {
42      addr_t               fptr;
43 
44      const boot_info_t *boot_info = get_boot_info();
45 
46      printk("Call stack dump:\n");
47 
48      fptr = get_fpointer();
49 
50      while(fptr != NULL) {
51          addr_t return_addr = get_ret_addr(fptr);
52          if(return_addr == NULL) {
53              break;
54          }
55 
56          /* assume e8 xx xx xx xx for call instruction encoding */
57          return_addr -= 5;
58 
59          elf_symbol_t symbol;
60          int retval = elf_lookup_symbol(
61                  boot_info->kernel_start,
62                  (Elf32_Addr)return_addr,
63                  STT_FUNCTION,
64                  &symbol);
65 
66          if(retval < 0) {
67              printk("\t0x%x (unknown)\n", return_addr);
68          }
69          else {
70              const char *name = symbol.name;
71 
72              if(name == NULL) {
73                  name = "[unknown]";
74              }
75 
76              printk(
77                      "\t0x%x (%s+%u)\n",
78                      return_addr,
```

```
                    name,
                    return_addr - symbol.addr);
        }

        fptr = get_caller_fpointer(fptr);
    }
}
```

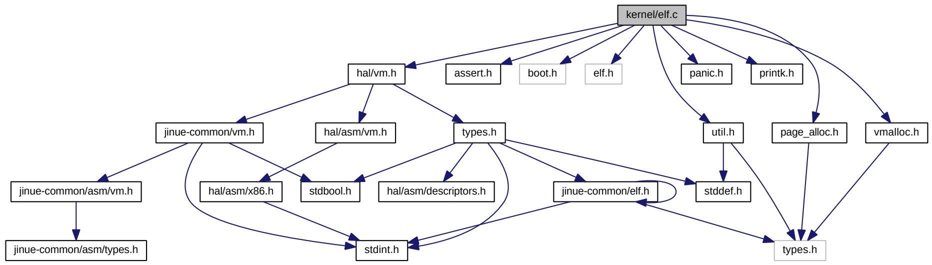
Here is the call graph for this function:



4.87 kernel/elf.c File Reference

```
#include <hal/vm.h>
#include <assert.h>
#include <boot.h>
#include <elf.h>
#include <page_alloc.h>
#include <panic.h>
#include <printk.h>
#include <vmalloc.h>
#include <util.h>
```

Include dependency graph for elf.c:



Functions

- void **elf_check** (**Elf32_Ehdr** ∗elf)
- void **elf_load** (**elf_info_t** ∗info, **Elf32_Ehdr** ∗elf, **addr_space_t** ∗addr_space, **boot_alloc_t** ∗boot_alloc)
- void **elf_setup_stack** (**elf_info_t** ∗info, **boot_alloc_t** ∗boot_alloc)
- int **elf_lookup_symbol** (const **Elf32_Ehdr** ∗elf_header, **Elf32_Addr** addr, int type, **elf_symbol_t** ∗result)

4.87.1 Function Documentation

4.87.1.1 void elf_check (Elf32_Ehdr ∗ *elf*)

Definition at line 43 of file elf.c.

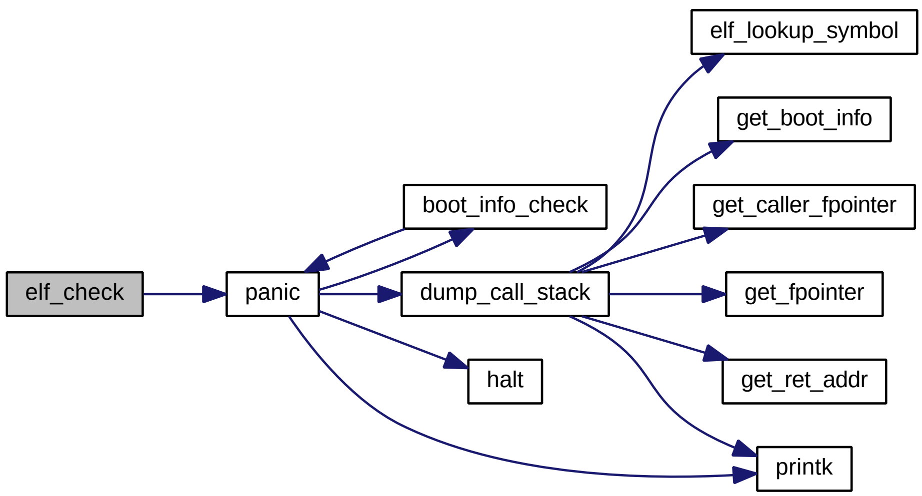
References Elf32_Ehdr::e_entry, Elf32_Ehdr::e_flags, Elf32_Ehdr::e_ident, Elf32_Ehdr::e_machine, Elf32_Ehdr::e_phentsize, Elf32_Ehdr::e_phnum, Elf32_Ehdr::e_phoff, Elf32_Ehdr::e_type, Elf32_Ehdr::e_version, EI_CLASS, EI_DATA, EI_MAG0, EI_MAG1, EI_MAG2, EI_MAG3, EI_VERSION, ELF_MAGIC0, ELF_MAGIC1, ELF_MAGIC2, ELF_MAGIC3, ELFCLASS32, ELFDATA2LSB, EM_386, ET_EXEC, and panic().

Referenced by elf_load().

```
43                                 {
44     /* check: valid ELF binary magic number */
45     if(     elf->e_ident[EI_MAG0] != ELF_MAGIC0 ||
46             elf->e_ident[EI_MAG1] != ELF_MAGIC1 ||
47             elf->e_ident[EI_MAG2] != ELF_MAGIC2 ||
48             elf->e_ident[EI_MAG3] != ELF_MAGIC3 ) {
49         panic("Not an ELF binary");
50     }
51 
52     /* check: 32-bit objects */
53     if(elf->e_ident[EI_CLASS] != ELFCLASS32) {
54         panic("Bad file class");
55     }
56 
57     /* check: endianess */
58     if(elf->e_ident[EI_DATA] != ELFDATA2LSB) {
59         panic("Bad endianess");
60     }
61 
62     /* check: version */
63     if(elf->e_version != 1 || elf->e_ident[EI_VERSION] != 1) {
64         panic("Not ELF version 1");
65     }
66 
67     /* check: machine */
68     if(elf->e_machine != EM_386) {
69         panic("This process manager binary does not target the x86 architecture");
70     }
71 
72     /* check: the 32-bit Intel architecture defines no flags */
73     if(elf->e_flags != 0) {
```

```
        panic("Invalid flags specified");
    }

    /* check: file type is executable */
    if(elf->e_type != ET_EXEC) {
        panic("process manager binary is not an an executable");
    }

    /* check: must have a program header */
    if(elf->e_phoff == 0 || elf->e_phnum == 0) {
        panic("No program headers");
    }

    /* check: must have an entry point */
    if(elf->e_entry == 0) {
        panic("No entry point for process manager");
    }

    /* check: program header entry size */
    if(elf->e_phentsize != sizeof(Elf32_Phdr)) {
        panic("Unsupported program header size");
    }
}
```

Here is the call graph for this function:



4.87.1.2 void elf_load (elf_info_t * *info,* Elf32_Ehdr * *elf,* addr_space_t * *addr_space,* boot_alloc_t * *boot_alloc*)

TODO: add exec flag once PAE is enabled

TODO add exec flag once PAE is enabled TODO lookup actual address of page frame

Definition at line 98 of file elf.c.

References elf_info_t::addr_space, ALIGN_END_PTR, ALIGN_START_PTR, elf_info_t::at_phdr, elf_info_t::at_phent, elf_info_t::at_phnum, boot_page_frame_alloc(), Elf32_Ehdr::e_entry, Elf32_Ehdr::e_phentsize, Elf32_Ehdr::e_phnum, Elf32_Ehdr::e_phoff, EARLY_PTR_TO_PHYS_ADDR, elf_check(), elf_setup_stack(), elf_info_t::entry, Elf32_Phdr::p_filesz, Elf32_Phdr::p_memsz, Elf32_Phdr::p_vaddr, PAGE_SIZE, panic(), PF_W, printk(), PT_LOAD, VM_FLAG_READ_ONLY, VM_FLAG_READ_WRITE, and vm_map_user().

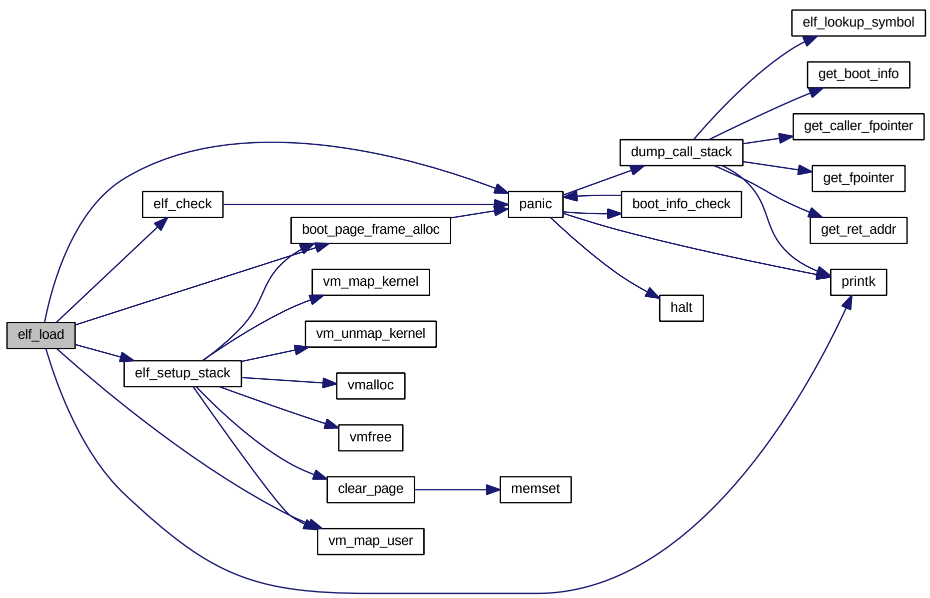
Referenced by kmain().

```
                                      {

    unsigned int idx;
```

```c

    /* check that ELF binary is valid */
    elf_check(elf);

    /* get the program header table */
    Elf32_Phdr * phdr   = (Elf32_Phdr *)((char *)elf + elf->e_phoff);

    info->at_phdr       = (addr_t)phdr;
    info->at_phnum      = elf->e_phnum;
    info->at_phent      = elf->e_phentsize;
    info->addr_space    = addr_space;
    info->entry         = (addr_t)elf->e_entry;

    for(idx = 0; idx < elf->e_phnum; ++idx) {
        if(phdr[idx].p_type != PT_LOAD) {
            continue;
        }

        /* check that the segment is not in the region reserved for kernel use */
        if(! user_buffer_check((void *)phdr[idx].p_vaddr, phdr[idx].p_memsz)) {
            panic("process manager memory layout -- address of segment too low");
        }

        /* set start and end addresses for mapping and copying */
        char *file_ptr  = (char *)elf + phdr[idx].p_offset;
        char *vptr      = (char *)phdr[idx].p_vaddr;
        char *vend      = vptr  + phdr[idx].p_memsz;  /* limit for padding */
        char *vfend     = vptr  + phdr[idx].p_filesz; /* limit for copy */

        /* align on page boundaries, be inclusive,
           note that vfend is not aligned        */
        file_ptr        = ALIGN_START_PTR(file_ptr, PAGE_SIZE);
        vptr            = ALIGN_START_PTR(vptr,     PAGE_SIZE);
        vend            = ALIGN_END_PTR(vend,       PAGE_SIZE);

        /* copy if we have to */
        if( (phdr[idx].p_flags & PF_W) || (phdr[idx].p_filesz != phdr[idx].p_memsz) ) {
            while(vptr < vend) {
                unsigned long    flags;
                char            *stop;

                /* start of this page and next page */
                char *vnext     = vptr + PAGE_SIZE;

                /* set flags */
                if(phdr[idx].p_flags & PF_W) {
                    flags = VM_FLAG_READ_WRITE;
                }
                else  {
                    flags = VM_FLAG_READ_ONLY;
                }

                /* allocate and map the new page */
                kern_paddr_t page = boot_page_frame_alloc(boot_alloc);
                vm_map_user(addr_space, (addr_t)vptr, page, flags);

                /* copy */
                if(vnext > vfend) {
                    stop = vfend;
                }
                else {
                    stop = vnext;
                }

                while(vptr < stop) {
                    *(vptr++) = *(file_ptr++);
                }

                /* pad */
                while(vptr < vnext) {
                    *(vptr++) = 0;
                }
            }
        }
        else {
            while(vptr < vend) {
                /* perform mapping */
                vm_map_user(addr_space, (addr_t)vptr, EARLY_PTR_TO_PHYS_ADDR(file_ptr),
      VM_FLAG_READ_ONLY);

                vptr     += PAGE_SIZE;
```

```
                    file_ptr += PAGE_SIZE;
                }
            }
        }

        elf_setup_stack(info, boot_alloc);

        printk("ELF binary loaded.\n");
}
```

Here is the call graph for this function:



4.87.1.3 int elf_lookup_symbol (const Elf32_Ehdr * *elf_header,* Elf32_Addr *addr,* int *type,* elf_symbol_t * *result*)

Definition at line 272 of file elf.c.

References elf_symbol_t::addr, Elf32_Ehdr::e_shnum, ELF32_ST_TYPE, elf_symbol_t::name, NULL, Elf32_Shdr::sh_entsize, Elf32_Shdr::sh_link, Elf32_Shdr::sh_offset, Elf32_Shdr::sh_size, Elf32_Shdr::sh_type, SHT_SYMTAB, Elf32_Sym::st_info, Elf32_Sym::st_name, Elf32_Sym::st_size, and Elf32_Sym::st_value.

Referenced by dump_call_stack().

```
                                        {

        int      idx;
        size_t   symbol_entry_size;
        size_t   symbol_table_size;

        const char *elf_file        = elf_file_bytes(elf_header);
        const char *symbols_table   = NULL;
        const char *string_table    = NULL;

        for(idx = 0; idx < elf_header->e_shnum; ++idx) {
            const Elf32_Shdr *section_header = elf_get_section_header(elf_header, idx);

            if(section_header->sh_type == SHT_SYMTAB) {
                symbols_table       = &elf_file[section_header->sh_offset];
                symbol_entry_size   = section_header->sh_entsize;
                symbol_table_size   = section_header->sh_size;

                const Elf32_Shdr *string_section_header = elf_get_section_header(
                        elf_header,
```

```
                    section_header->sh_link);

            string_table = &elf_file[string_section_header->sh_offset];

            break;
        }
    }

    if(symbols_table == NULL) {
        /* no symbol table */
        return -1;
    }

    const char *symbol = symbols_table;

    while(symbol < symbols_table + symbol_table_size) {
        const Elf32_Sym *symbol_header = (const Elf32_Sym *)symbol;

        if(ELF32_ST_TYPE(symbol_header->st_info) == type) {
            Elf32_Addr lookup_addr  = (Elf32_Addr)addr;
            Elf32_Addr start        = symbol_header->st_value;
            Elf32_Addr end          = start + symbol_header->st_size;

            if(lookup_addr >= start && lookup_addr < end) {
                result->addr = symbol_header->st_value;
                result->name = &string_table[symbol_header->st_name];

                return 0;
            }
        }

        symbol += symbol_entry_size;
    }

    /* not found */
    return -1;
}
```

4.87.1.4 void elf_setup_stack (elf_info_t * *info,* boot_alloc_t * *boot_alloc*)

TODO: check for overlap of stack with loaded segments

Definition at line 198 of file elf.c.

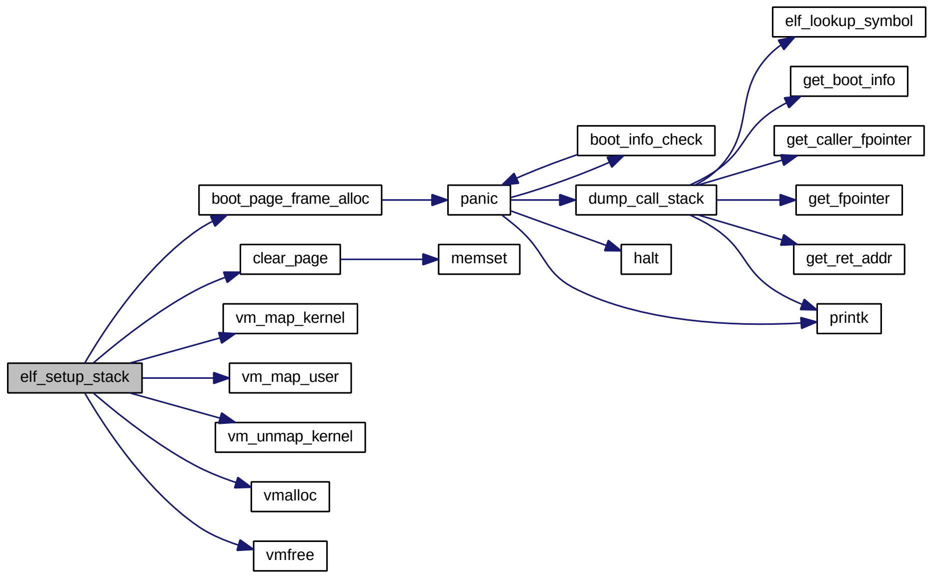
References Elf32_auxv_t::a_type, Elf32_auxv_t::a_un, Elf32_auxv_t::a_val, elf_info_t::addr_space, AT_ENTRY, AT_NULL, AT_PAGESZ, elf_info_t::at_phdr, AT_PHDR, elf_info_t::at_phent, AT_PHENT, elf_info_t::at_phnum, AT_PHNUM, AT_STACKBASE, boot_page_frame_alloc(), clear_page(), elf_info_t::entry, PAGE_SIZE, elf_info_t::stack_addr, STACK_BASE, STACK_START, VM_FLAG_READ_WRITE, vm_map_kernel(), vm_map_user(), vm_unmap_kernel(), vmalloc(), and vmfree().

Referenced by elf_load().

```
198                                                                 {
199     kern_paddr_t page;
200     addr_t vpage;
201
204     /* initial stack allocation */
205     for(vpage = (addr_t)STACK_START; vpage < (addr_t)STACK_BASE; vpage +=
      PAGE_SIZE) {
206         page  = boot_page_frame_alloc(boot_alloc);
207         vm_map_user(info->addr_space, vpage, page, VM_FLAG_READ_WRITE);
208
209         /* This newly allocated page may have data left from a previous boot which
210          * may contain sensitive information. Let's clear it. */
211         clear_page(vpage);
212     }
213
214     /* At this point, page has the address of the stack's top-most page frame,
215      * which is the one in which we are about to copy the auxiliary vectors. Map
216      * it temporarily in this address space so we can write to it. */
217     addr_t top_page = vmalloc();
218     vm_map_kernel(top_page, page, VM_FLAG_READ_WRITE);
219
```

```
    /* start at the top */
    uint32_t *sp = (uint32_t *)(top_page + PAGE_SIZE);

    /* Program name string: "proc", null-terminated */
    *(--sp) = 0;
    *(--sp) = 0x636f7270;

    char *argv0 = (char *)STACK_BASE - 2 * sizeof(uint32_t);

    /* auxiliary vectors */
    Elf32_auxv_t *auxvp = (Elf32_auxv_t *)sp - 7;

    auxvp[0].a_type     = AT_PHDR;
    auxvp[0].a_un.a_val = (int32_t)info->at_phdr;

    auxvp[1].a_type     = AT_PHENT;
    auxvp[1].a_un.a_val = (int32_t)info->at_phent;

    auxvp[2].a_type     = AT_PHNUM;
    auxvp[2].a_un.a_val = (int32_t)info->at_phnum;

    auxvp[3].a_type     = AT_PAGESZ;
    auxvp[3].a_un.a_val = PAGE_SIZE;

    auxvp[4].a_type     = AT_ENTRY;
    auxvp[4].a_un.a_val = (int32_t)info->entry;

    auxvp[5].a_type     = AT_STACKBASE;
    auxvp[5].a_un.a_val = STACK_BASE;

    auxvp[6].a_type     = AT_NULL;
    auxvp[6].a_un.a_val = 0;

    sp = (uint32_t *)auxvp;

    /* empty environment variables */
    *(--sp) = 0;

    /* argv with only program name */
    *(--sp) = 0;
    *(--sp) = (uint32_t)argv0;

    /* argc */
    *(--sp) = 1;

    info->stack_addr = (addr_t)STACK_BASE - PAGE_SIZE + ((addr_t)sp - top_page);

    /* unmap and free temporary page */
    vm_unmap_kernel(top_page);
    vmfree(top_page);
}
```

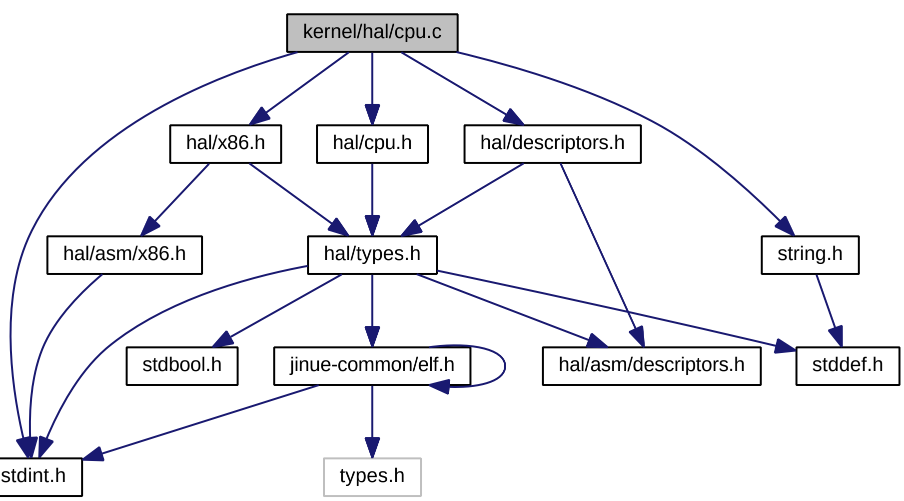
Here is the call graph for this function:



4.88 kernel/hal/cpu.c File Reference

```
#include <hal/cpu.h>
#include <hal/descriptors.h>
#include <hal/x86.h>
#include <stdint.h>
#include <string.h>
```

Include dependency graph for cpu.c:



Functions

- void **cpu_init_data** (**cpu_data_t** ∗data)

- void **cpu_detect_features** (void)

Variables

- **cpu_info_t cpu_info**

4.88.1 Function Documentation

4.88.1.1 void cpu_detect_features (void)

Definition at line 87 of file cpu.c.

References CPU_EFLAGS_ID, CPU_FEATURE_CPUID, CPU_FEATURE_LOCAL_APIC, CPU_FEATURE_PAE, CP-U_FEATURE_SYSCALL, CPU_FEATURE_SYSENTER, CPU_VENDOR_AMD, CPU_VENDOR_AMD_DW0, CPU_V-ENDOR_AMD_DW1, CPU_VENDOR_AMD_DW2, CPU_VENDOR_GENERIC, CPU_VENDOR_INTEL, CPU_VEND-OR_INTEL_DW0, CPU_VENDOR_INTEL_DW1, CPU_VENDOR_INTEL_DW2, cpuid(), CPUID_EXT_FEATURE_S-YSCALL, CPUID_FEATURE_APIC, CPUID_FEATURE_CLFLUSH, CPUID_FEATURE_PAE, CPUID_FEATURE_SE-P, cpu_info_t::dcache_alignment, x86_cpuid_regs_t::eax, x86_cpuid_regs_t::ebx, x86_cpuid_regs_t::ecx, x86_cpuid_-regs_t::edx, cpu_info_t::family, cpu_info_t::features, get_eflags(), cpu_info_t::model, set_eflags(), cpu_info_t::stepping, and cpu_info_t::vendor.

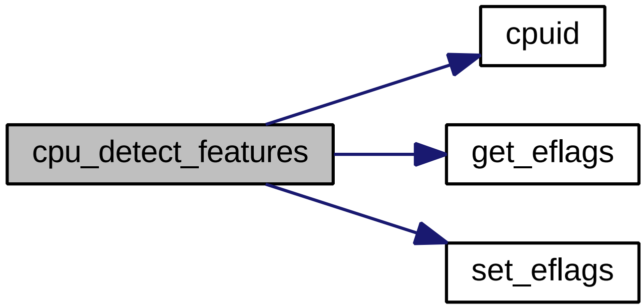
Referenced by hal_init().

```
87                                  {
88      uint32_t temp_eflags;
89
90      /* default values */
91      cpu_info.dcache_alignment   = 32;
92      cpu_info.features           = 0;
93      cpu_info.vendor             = CPU_VENDOR_GENERIC;
94      cpu_info.family             = 0;
95      cpu_info.model              = 0;
96      cpu_info.stepping           = 0;
97
98      /* The CPUID instruction is available if we can change the value of eflags
99       * bit 21 (ID) */
100     temp_eflags  = get_eflags();
101     temp_eflags ^= CPU_EFLAGS_ID;
102     set_eflags(temp_eflags);
103
104     if(temp_eflags == get_eflags()) {
105         cpu_info.features |= CPU_FEATURE_CPUID;
106     }
107
108     if(cpu_has_feature(CPU_FEATURE_CPUID)) {
109         uint32_t            signature;
110         uint32_t            flags, ext_flags;
111         uint32_t            vendor_dw0, vendor_dw1, vendor_dw2;
112         uint32_t            cpuid_max;
113         uint32_t            cpuid_ext_max;
114         x86_cpuid_regs_t    regs;
115
116         /* default values */
117         flags               = 0;
118         ext_flags           = 0;
119
120         /* function 0: vendor ID string, max value of eax when calling CPUID */
121         regs.eax = 0;
122
123         /* call CPUID instruction */
124         cpuid_max  = cpuid(&regs);
125         vendor_dw0 = regs.ebx;
126         vendor_dw1 = regs.edx;
127         vendor_dw2 = regs.ecx;
128
129         /* identify vendor */
130         if(    vendor_dw0 == CPU_VENDOR_AMD_DW0
131             && vendor_dw1 == CPU_VENDOR_AMD_DW1
```

```
            && vendor_dw2 == CPU_VENDOR_AMD_DW2) {

        cpu_info.vendor = CPU_VENDOR_AMD;
    }
    else if (vendor_dw0 == CPU_VENDOR_INTEL_DW0
        &&   vendor_dw1 == CPU_VENDOR_INTEL_DW1
        &&   vendor_dw2 == CPU_VENDOR_INTEL_DW2) {

        cpu_info.vendor = CPU_VENDOR_INTEL;
    }

    /* get processor signature (family/model/stepping) and feature flags */
    if(cpuid_max >= 1) {
        /* function 1: processor signature and feature flags */
        regs.eax = 1;

        /* call CPUID instruction */
        signature = cpuid(&regs);

        /* set processor signature */
        cpu_info.stepping  = signature       & 0xf;
        cpu_info.model     = (signature>>4)  & 0xf;
        cpu_info.family    = (signature>>8)  & 0xf;

        /* feature flags */
        flags = regs.edx;

        /* cache alignment */
        if(flags & CPUID_FEATURE_CLFLUSH) {
            cpu_info.dcache_alignment = ((regs.ebx >> 8) & 0xff) * 8;
        }
    }

    /* extended function 0: max value of eax when calling CPUID (extended function) */
    regs.eax = 0x80000000;
    cpuid_ext_max = cpuid(&regs);

    /* get extended feature flags */
    if(cpuid_ext_max >= 0x80000001) {
        /* extended function 1: extended feature flags */
        regs.eax = 0x80000001;
        (void)cpuid(&regs);

        /* extended feature flags */
        ext_flags = regs.edx;
    }

    /* support for SYSENTER/SYSEXIT instructions */
    if(flags & CPUID_FEATURE_SEP) {
        if(cpu_info.vendor == CPU_VENDOR_AMD) {
            cpu_info.features |= CPU_FEATURE_SYSENTER;
        }
        else if(cpu_info.vendor == CPU_VENDOR_INTEL) {
            if(cpu_info.family == 6 && cpu_info.model < 3 && cpu_info.stepping < 3) {
                /* not supported */
            }
            else {
                cpu_info.features |= CPU_FEATURE_SYSENTER;
            }
        }
    }

    /* support for SYSCALL/SYSRET instructions */
    if(cpu_info.vendor == CPU_VENDOR_AMD) {
        if(ext_flags & CPUID_EXT_FEATURE_SYSCALL) {
            cpu_info.features |= CPU_FEATURE_SYSCALL;
        }
    }

    /* support for local APIC */
    if(cpu_info.vendor == CPU_VENDOR_AMD || cpu_info.vendor == CPU_VENDOR_INTEL) {
        if(flags & CPUID_FEATURE_APIC) {
            cpu_info.features |= CPU_FEATURE_LOCAL_APIC;
        }
    }

    /* support for physical address extension (PAE) */
    if(cpu_info.vendor == CPU_VENDOR_AMD || cpu_info.vendor == CPU_VENDOR_INTEL) {
```

```
210             if(flags & CPUID_FEATURE_PAE) {
211                 cpu_info.features |= CPU_FEATURE_PAE;
212             }
213         }
214     }
215 }
```

Here is the call graph for this function:



4.88.1.2 void cpu_init_data (cpu_data_t * *data*)

Definition at line 42 of file cpu.c.

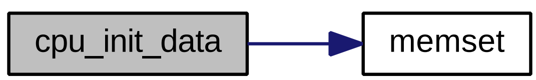
References cpu_data_t::current_addr_space, tss_t::esp0, tss_t::esp1, tss_t::esp2, cpu_data_t::gdt, GDT_KERNEL_-CODE, GDT_KERNEL_DATA, GDT_NULL, GDT_PER_CPU_DATA, GDT_TSS, GDT_USER_CODE, GDT_USER_-DATA, GDT_USER_TLS_DATA, memset(), NULL, RPL_KERNEL, SEG_DESCRIPTOR, SEG_FLAG_32BIT, SEG_-FLAG_IN_BYTES, SEG_FLAG_KERNEL, SEG_FLAG_NORMAL, SEG_FLAG_NOSYSTEM, SEG_FLAG_PRESENT, SEG_FLAG_TSS, SEG_FLAG_USER, SEG_SELECTOR, SEG_TYPE_CODE, SEG_TYPE_DATA, SEG_TYPE_TSS, cpu_data_t::self, tss_t::ss0, tss_t::ss1, tss_t::ss2, cpu_data_t::tss, and TSS_LIMIT.

Referenced by hal_init().

```
42                                     {
43     tss_t *tss;
44
45     tss = &data->tss;
46
47     /* initialize with zeroes  */
48     memset(data, '\0', sizeof(cpu_data_t));
49
50     data->self                      = data;
51     data->current_addr_space        = NULL;
52
53     /* initialize GDT */
54     data->gdt[GDT_NULL] = SEG_DESCRIPTOR(0, 0, 0);
55
56     data->gdt[GDT_KERNEL_CODE] =
57         SEG_DESCRIPTOR( 0,       0xfffff,                SEG_TYPE_CODE  |
      SEG_FLAG_KERNEL | SEG_FLAG_NORMAL);
58
59     data->gdt[GDT_KERNEL_DATA] =
60         SEG_DESCRIPTOR( 0,       0xfffff,                SEG_TYPE_DATA  |
      SEG_FLAG_KERNEL | SEG_FLAG_NORMAL);
61
62     data->gdt[GDT_USER_CODE] =
63         SEG_DESCRIPTOR( 0,       0xfffff,                SEG_TYPE_CODE  |
      SEG_FLAG_USER   | SEG_FLAG_NORMAL);
64
65     data->gdt[GDT_USER_DATA] =
66         SEG_DESCRIPTOR( 0,       0xfffff,                SEG_TYPE_DATA  |
      SEG_FLAG_USER   | SEG_FLAG_NORMAL);
67
68     data->gdt[GDT_TSS] =
```

```
        SEG_DESCRIPTOR( tss,    TSS_LIMIT-1,            SEG_TYPE_TSS   |
    SEG_FLAG_KERNEL | SEG_FLAG_TSS);

    data->gdt[GDT_PER_CPU_DATA] =
        SEG_DESCRIPTOR( data,   sizeof(cpu_data_t)-1,   SEG_TYPE_DATA  |
    SEG_FLAG_KERNEL | SEG_FLAG_32BIT | SEG_FLAG_IN_BYTES | SEG_FLAG_NOSYSTEM |
    SEG_FLAG_PRESENT);

    data->gdt[GDT_USER_TLS_DATA] = SEG_DESCRIPTOR(0, 0, 0);

    /* setup kernel stack in TSS */
    tss->ss0  = SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL);
    tss->ss1  = SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL);
    tss->ss2  = SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL);

    /* kernel stack address is updated by thread_context_switch() */
    tss->esp0 = NULL;
    tss->esp1 = NULL;
    tss->esp2 = NULL;
}
```

Here is the call graph for this function:



4.88.2 Variable Documentation

4.88.2.1 cpu_info_t cpu_info

Definition at line 39 of file cpu.c.

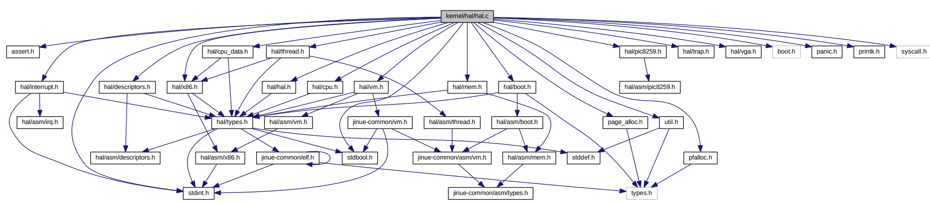
Referenced by slab_cache_init().

4.89 kernel/hal/hal.c File Reference

```
#include <assert.h>
```

```
#include <hal/boot.h>
#include <hal/cpu.h>
#include <hal/cpu_data.h>
#include <hal/descriptors.h>
#include <hal/hal.h>
#include <hal/interrupt.h>
#include <hal/mem.h>
#include <hal/pic8259.h>
#include <hal/thread.h>
#include <hal/trap.h>
#include <hal/vga.h>
#include <hal/vm.h>
#include <hal/x86.h>
#include <boot.h>
#include <panic.h>
#include <page_alloc.h>
#include <pfalloc.h>
#include <printk.h>
#include <stdbool.h>
#include <stdint.h>
#include <syscall.h>
#include <util.h>
```

Include dependency graph for hal.c:



Functions

- void **hal_init** (**boot_alloc_t** ∗boot_alloc, const **boot_info_t** ∗**boot_info**)

Variables

- int **syscall_method**

  Specifies the entry point to use for system calls.

4.89.1 Function Documentation

4.89.1.1 void hal_init (**boot_alloc_t** ∗ *boot_alloc,* const **boot_info_t** ∗ *boot_info*)

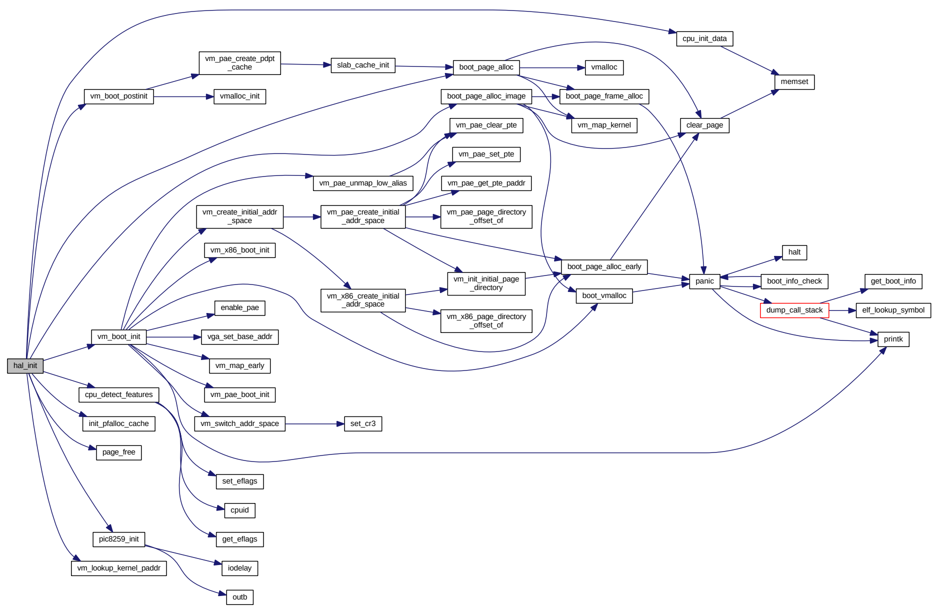
Definition at line 156 of file hal.c.

References assert, boot_heap_alloc, boot_page_alloc(), boot_page_alloc_image(), CPU_DATA_ALIGNMENT, cpu-_detect_features(), CPU_FEATURE_PAE, cpu_init_data(), global_pfalloc_cache, IDT_PIC8259_BASE, init_pfalloc_-cache(), KERNEL_PAGE_STACK_INIT, page_free(), pffree, pic8259_init(), vm_boot_init(), vm_boot_postinit(), and vm-_lookup_kernel_paddr().

Referenced by kmain().

```
{
    int idx;

    /* get cpu info */
    cpu_detect_features();

    /* allocate per-CPU data
     *
     * We need to ensure that the Task State Segment (TSS) contained in this
     * memory block does not cross a page boundary. */
    assert(sizeof(cpu_data_t) < CPU_DATA_ALIGNMENT);

    cpu_data_t *cpu_data = boot_heap_alloc(boot_alloc, cpu_data_t,
  CPU_DATA_ALIGNMENT);

    /* initialize per-CPU data */
    cpu_init_data(cpu_data);

    /* Initialize interrupt descriptor table (IDT)
     *
     * This function modifies the IDT in-place (see trap.asm). This must be
     * done before vm_boot_init() because the page protection bits set up by
     * vm_boot_init() prevent this. */
    hal_init_idt();

    /* Initialize programmable interrupt_controller. */
    pic8259_init(IDT_PIC8259_BASE);

    /* initialize virtual memory management, enable paging
     *
     * below this point, it is no longer safe to call pfalloc_early() */
    bool use_pae = cpu_has_feature(CPU_FEATURE_PAE);
    vm_boot_init(boot_info, use_pae, cpu_data, boot_alloc);

    /* Initialize GDT and TSS */
    hal_init_descriptors(cpu_data, boot_alloc);

    /* initialize the page frame allocator */
    kern_paddr_t *page_stack_buffer = (kern_paddr_t *)boot_page_alloc_image(boot_alloc);
    init_pfalloc_cache(&global_pfalloc_cache, page_stack_buffer);

    /* TODO Remove this once vm.c rework is done. */
    for(idx = 0; idx < KERNEL_PAGE_STACK_INIT; ++idx) {
        pffree(
                vm_lookup_kernel_paddr(
                        boot_page_alloc_image(boot_alloc)));
    }

    /* Initialize virtual memory allocator and VM management caches. */
    vm_boot_postinit(boot_info, boot_alloc, use_pae);

    /* TODO Remove this once add page frame system call is implemented.
     *
     * The test user space program needs one page to create a new thread. */
    page_free(boot_page_alloc(boot_alloc));

    /* choose system call method */
    hal_select_syscall_method();
}
```

Here is the call graph for this function:



4.89.2 Variable Documentation

4.89.2.1 int syscall_method

Specifies the entry point to use for system calls.

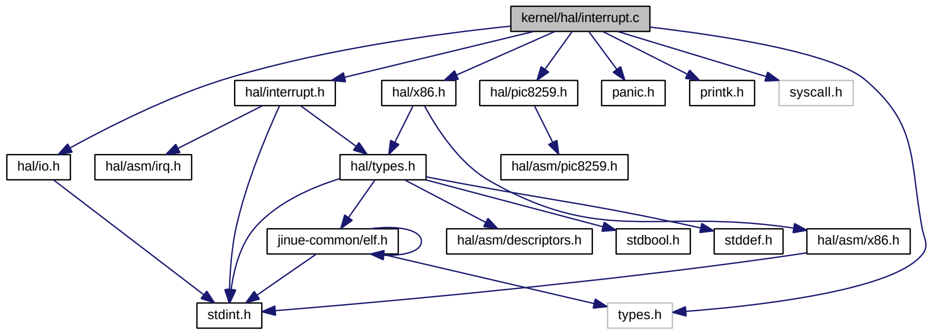
Definition at line 58 of file hal.c.

Referenced by dispatch_syscall().

4.90 kernel/hal/interrupt.c File Reference

```
#include <hal/interrupt.h>
#include <hal/io.h>
#include <hal/pic8259.h>
#include <hal/x86.h>
#include <panic.h>
#include <printk.h>
#include <syscall.h>
#include <types.h>
```

Include dependency graph for interrupt.c:



Functions

- void **dispatch_interrupt** (**trapframe_t** ∗trapframe)

4.90.1 Function Documentation

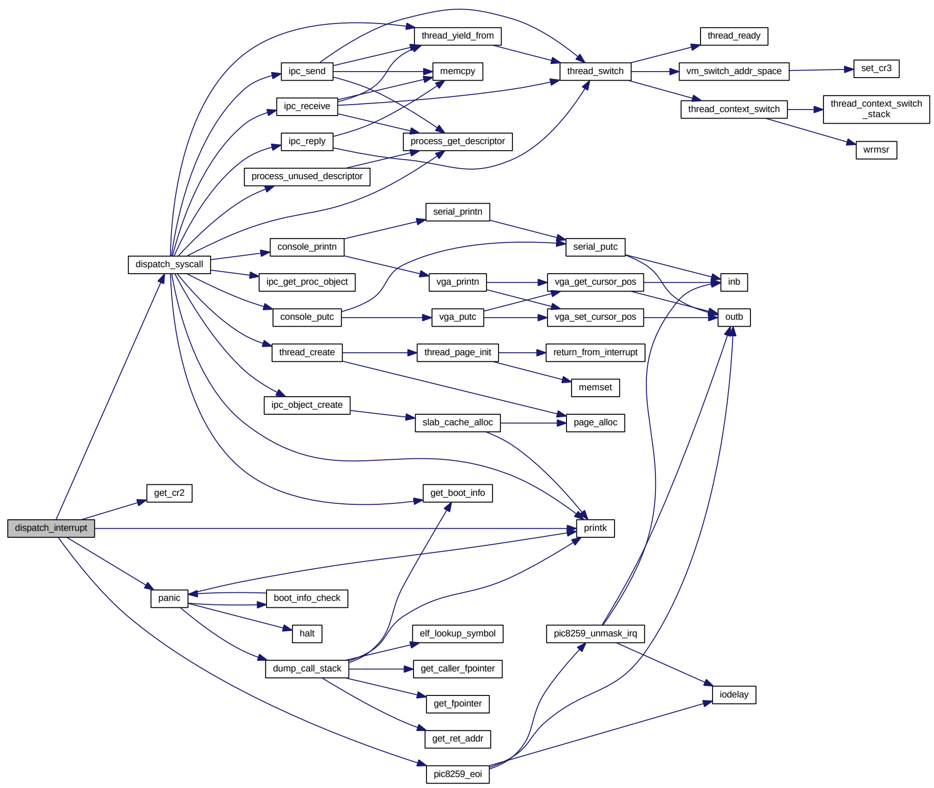
4.90.1.1 void dispatch_interrupt (trapframe_t ∗ *trapframe*)

Definition at line 42 of file interrupt.c.

References dispatch_syscall(), trapframe_t::eip, trapframe_t::errcode, get_cr2(), IDT_LAST_EXCEPTION, IDT_PIC8259_BASE, trapframe_t::ivt, panic(), pic8259_eoi(), PIC8259_IRQ_COUNT, printk(), and SYSCALL_IRQ.

```
42                                                 {
43     unsigned int    ivt         = trapframe->ivt;
44     uintptr_t       eip         = trapframe->eip;
45     uint32_t        errcode     = trapframe->errcode;
46 
47     /* exceptions */
48     if(ivt <= IDT_LAST_EXCEPTION) {
49         printk("EXCEPT: %u cr2=0x%x errcode=0x%x eip=0x%x\n", ivt, get_cr2(), errcode, eip);
50 
51         /* never returns */
52         panic("caught exception");
53     }
54 
55     if(ivt == SYSCALL_IRQ) {
56         /* slow system call method */
57         dispatch_syscall(trapframe);
58     }
59     else if(ivt >= IDT_PIC8259_BASE && ivt < IDT_PIC8259_BASE +
      PIC8259_IRQ_COUNT) {
60         int irq = ivt - IDT_PIC8259_BASE;
61         printk("IRQ: %u (vector %u)\n", irq, ivt);
62         pic8259_eoi(irq);
63     }
64     else {
65         printk("INTR: vector %u\n", ivt);
66     }
67 }
```

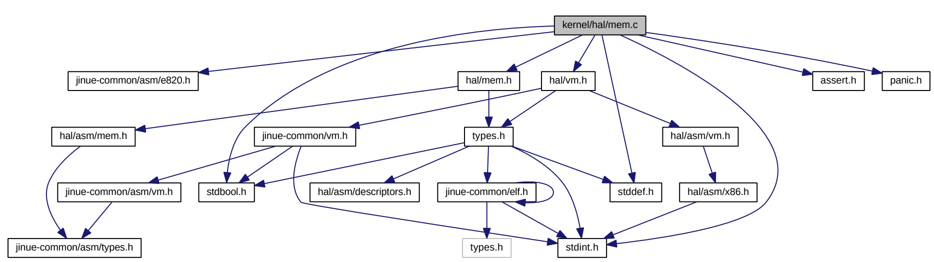
Here is the call graph for this function:



4.91 kernel/hal/mem.c File Reference

```
#include <jinue-common/asm/e820.h>
#include <hal/mem.h>
#include <hal/vm.h>
#include <assert.h>
#include <panic.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
```

Include dependency graph for mem.c:



Functions

- void **mem_check_memory** (**boot_alloc_t** ∗boot_alloc, const **boot_info_t** ∗**boot_info**)

4.91.1 Function Documentation

4.91.1.1 void mem_check_memory (boot_alloc_t ∗ *boot_alloc*, const boot_info_t ∗ *boot_info*)

ASSERTION: Any unsigned value less than MEM_ZONE_MEM32_END can be stored in 32 bits.

Definition at line 57 of file mem.c.

References e820_t::addr, assert, boot_info_t::boot_end, boot_info_t::e820_entries, boot_info_t::e820_map, E820_R-AM, EARLY_VIRT_TO_PHYS, GB, KERNEL_EARLY_LIMIT, boot_alloc_t::kernel_paddr_limit, boot_alloc_t::kernel_-paddr_top, boot_alloc_t::kernel_vm_limit, boot_alloc_t::kernel_vm_top, MEM_ZONE_DMA16_END, MEM_ZONE_DM-A16_START, MEM_ZONE_MEM32_END, MEM_ZONE_MEM32_START, panic(), boot_info_t::ramdisk_size, boot_info-_t::ramdisk_start, and e820_t::type.

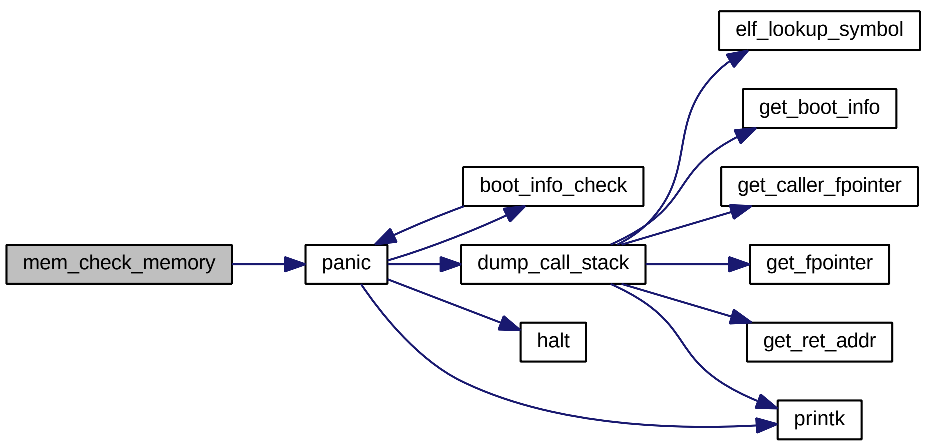
Referenced by kmain().

```
57                                                                                {
58     int idx;
59
60 #if 0
61     if((uint64_t)boot_info->ramdisk_start + boot_info->ramdisk_size >
      MEM_ZONE_MEM32_END) {
62         panic("Initial RAM disk loaded too high in memory.");
63     }
64
65     uint32_t ramdisk_end = boot_info->ramdisk_start + boot_info->ramdisk_size;
66 #endif
67
68     /* ------------------------------------------------------------------------
69      * We consult the memory map provided by the BIOS to figure out how much
70      * memory is available in both zones usable by the kernel. We also want to
71      * make sure the initial RAM disk image is in available RAM.
72      *
73      * The first step in accomplishing this is to iterate over all entries that
74      * are reported as available RAM and confirm at least one of them covers
75      * each of both zones (at least the start) and the initial RAM disk.
76      * -------------------------------------------------------------------- */
77     uint32_t zone_dma16_top = 0;
78     uint32_t zone_mem32_top = 0;
79 #if 0
80     bool ramdisk_ok         = false;
81 #endif
82
84     assert(MEM_ZONE_MEM32_END < (uint64_t)4 * GB);
85
86     for(idx = 0; idx < boot_info->e820_entries; ++idx) {
87         const e820_t *entry = &boot_info->e820_map[idx];
88
89         /* Consider only usable RAM entries. */
90         if(entry->type != E820_RAM) {
91             continue;
92         }
93
94         /* Ignore entries that start past MEM_ZONE_MEM32_END since the kernel
95          * cannot use them. Past this check, entry->addr is assumed to be
96          * representable in 32 bits.  */
97         if(entry->addr >= MEM_ZONE_MEM32_END) {
98             continue;
99         }
100
101         uint32_t entry_end = clip_e820_entry_end(entry);
102
103         /* If this entry covers the start the DMA16 zone, adjust the top pointer
104          * accordingly. Overlapping entries are resolved in favor of the largest
105          * entry. */
106         if(entry->addr <= MEM_ZONE_DMA16_START && entry_end >
      MEM_ZONE_DMA16_START) {
107             /* This condition covers the initial case where zone_dma16_top is zero. */
```

```
            if(entry_end > zone_dma16_top) {
                zone_dma16_top = entry_end;
            }
        }

        /* Do the same for the MEM32 zone. */
        if(entry->addr <= MEM_ZONE_MEM32_START && entry_end > MEM_ZONE_MEM32_START) {
            if(entry_end > zone_mem32_top) {
                zone_mem32_top = entry_end;
            }
        }

        /* If this entry covers the initial RAM disk, this is good argument in
         * favor of it being in available RAM (one more check below). Unlike the
         * above, the entry must cover the initial RAM disk image completely,
         * not just the start. */
#if 0
        if(entry->addr <= boot_info->ramdisk_start && entry_end >= ramdisk_end) {
            ramdisk_ok = true;
        }
#endif
    }

    /* ----------------------------------------------------------------------
     * Next, iterate over non-available RAM entries of the map to ensure nothing
     * is relevant there.
     * ------------------------------------------------------------------- */
    for(idx = 0; idx < boot_info->e820_entries; ++idx) {
        const e820_t *entry = &boot_info->e820_map[idx];

        /* Consider only non-usable RAM entries. */
        if(entry->type == E820_RAM) {
            continue;
        }

        if(entry->addr >= MEM_ZONE_MEM32_END) {
            continue;
        }

        uint32_t entry_end = clip_e820_entry_end(entry);

        if(ranges_overlap(MEM_ZONE_DMA16_START, MEM_ZONE_DMA16_END, entry->addr, entry_end)) {
            if(entry->addr > MEM_ZONE_DMA16_START) {
                if(entry->addr < zone_dma16_top) {
                    zone_dma16_top = entry->addr;
                }
            }
            else {
                /* This reserved entry covers the start of the zone. */
                zone_dma16_top = 0;
            }
        }

        if(ranges_overlap(MEM_ZONE_MEM32_START, MEM_ZONE_MEM32_END, entry->addr, entry_end)) {
            if(entry->addr > MEM_ZONE_MEM32_START) {
                if(entry->addr < zone_mem32_top) {
                    zone_mem32_top = entry->addr;
                }
            }
            else {
                zone_mem32_top = 0;
            }
        }

        /* Check for overlap with the initial RAM disk. */
#if 0
        if(ranges_overlap(boot_info->ramdisk_start, ramdisk_end, entry->addr, entry_end)) {
            ramdisk_ok = false;
        }
#endif
    }

    /* ----------------------------------------------------------------------
     * Now that we are done, let's look at the results.
     * ------------------------------------------------------------------- */
#if 0
    if(! ramdisk_ok) {
        panic("Initial RAM disk was loaded in reserved memory.");
    }
#endif
```

```

    /* It is early during the boot process and the page table set up by the
     * setup code is still being used. This (single) page table maps the first
     * two megabytes of RAM linearly starting at KLIMIT in the virtual address
     * space. */
    boot_alloc->kernel_vm_top      = boot_info->boot_end;
    boot_alloc->kernel_vm_limit    = (addr_t)KERNEL_EARLY_LIMIT;
    boot_alloc->kernel_paddr_top   = EARLY_VIRT_TO_PHYS(boot_alloc->kernel_vm_top);
    boot_alloc->kernel_paddr_limit = zone_dma16_top;

    if(boot_alloc->kernel_paddr_top > boot_alloc->kernel_paddr_limit) {
        panic("Kernel image was loaded in reserved memory.");
    }

    /* TODO Compute sequential allocation limit taking initrd into account */
    /* TODO Report zone limits */
}
```

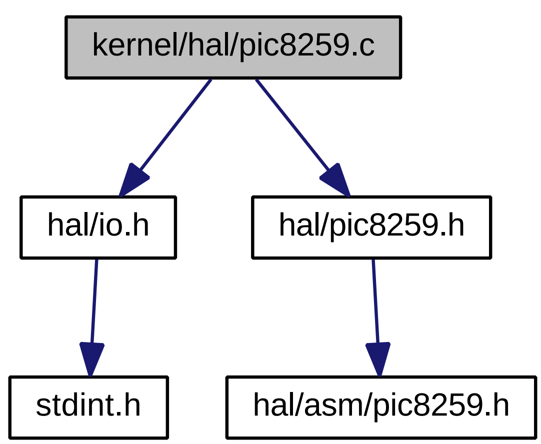
Here is the call graph for this function:



4.92 kernel/hal/pic8259.c File Reference

```
#include <hal/io.h>
#include <hal/pic8259.h>
```

Include dependency graph for pic8259.c:



Functions

- void **pic8259_init** (int intrvect_base)
- void **pic8259_mask_irq** (int irq)
- void **pic8259_unmask_irq** (int irq)
- void **pic8259_eoi** (int irq)

4.92.1 Function Documentation

4.92.1.1 void pic8259_eoi (int *irq*)

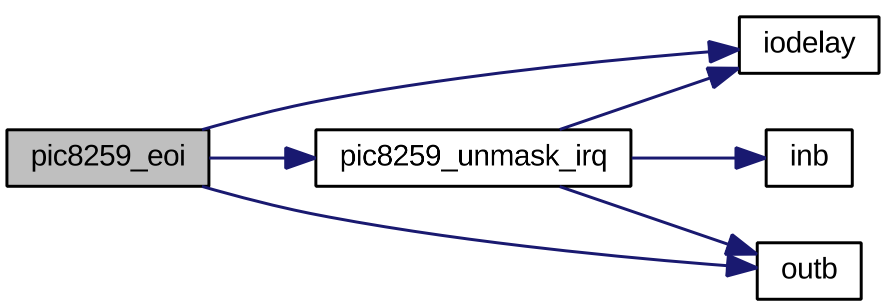
Definition at line 124 of file pic8259.c.

References iodelay(), outb(), PIC8259_EOI, PIC8259_IRQ_COUNT, PIC8259_MASTER_BASE, PIC8259_SLAVE_B-ASE, and pic8259_unmask_irq().

Referenced by dispatch_interrupt().

```
124                        {
125     if(irq < PIC8259_IRQ_COUNT) {
126         if(irq >= 8) {
127             outb(PIC8259_SLAVE_BASE+0, PIC8259_EOI);
128             iodelay();
129         }
130 
131         outb(PIC8259_MASTER_BASE+0, PIC8259_EOI);
132         iodelay();
133 
134         pic8259_unmask_irq(irq);
135     }
136 }
```

Here is the call graph for this function:



4.92.1.2 void pic8259_init (int *intrvect_base*)

Definition at line 36 of file pic8259.c.

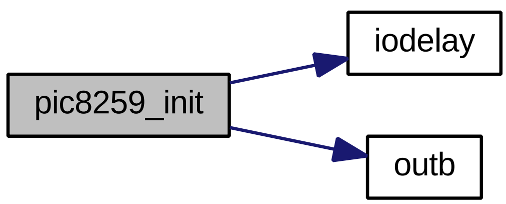
References iodelay(), outb(), PIC8259_CASCADE_INPUT, PIC8259_ICW1_1, PIC8259_ICW1_IC4, PIC8259_ICW4_-UPM, PIC8259_MASTER_BASE, and PIC8259_SLAVE_BASE.

Referenced by hal_init().

```
36                                {
37     /* Issue ICW1 to start initialization sequence of both interrupt controllers.
38      * Specify there will be an ICW4 in the sequence. Specify the interrupts are
```

```
     * edge-triggered and that the PICs are in a cascaded configuration by
     * leaving the relevant flags cleared. */
    outb(PIC8259_MASTER_BASE+0, PIC8259_ICW1_1 | PIC8259_ICW1_IC4);
    iodelay();
    outb(PIC8259_SLAVE_BASE+0, PIC8259_ICW1_1 | PIC8259_ICW1_IC4);
    iodelay();

    /* ICW2: base interrupt vector */
    outb(PIC8259_MASTER_BASE+1, intrvect_base);
    iodelay();
    outb(PIC8259_SLAVE_BASE+1, intrvect_base + 8);
    iodelay();

    /* ICW3: master-slave connections */
    outb(PIC8259_MASTER_BASE+1, (1<<PIC8259_CASCADE_INPUT));
    iodelay();
    outb(PIC8259_SLAVE_BASE+1, PIC8259_CASCADE_INPUT);
    iodelay();

    /* ICW4: Use 8088/8086 mode */
    outb(PIC8259_MASTER_BASE+1, PIC8259_ICW4_UPM);
    iodelay();
    outb(PIC8259_SLAVE_BASE+1, PIC8259_ICW4_UPM);
    iodelay();

    /* Set interrupt mask: all masked */
    outb(PIC8259_MASTER_BASE+1, 0xff & ~(1<<PIC8259_CASCADE_INPUT));
    iodelay();
    outb(PIC8259_SLAVE_BASE+1, 0xff);
    iodelay();
}
```

Here is the call graph for this function:



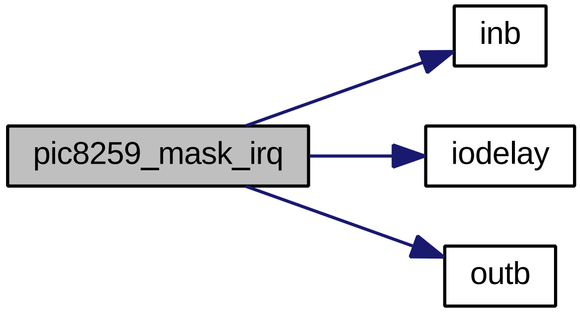
4.92.1.3 void pic8259_mask_irq (int *irq*)

Definition at line 71 of file pic8259.c.

References inb(), iodelay(), outb(), PIC8259_CASCADE_INPUT, PIC8259_IRQ_COUNT, PIC8259_MASTER_BASE, and PIC8259_SLAVE_BASE.

```
71                                 {
72      if(irq < PIC8259_IRQ_COUNT) {
73          if(irq < 8 && irq != PIC8259_CASCADE_INPUT) {
74              int mask = inb(PIC8259_MASTER_BASE+1);
75              iodelay();
76 
77              mask |= (1<<irq);
78              outb(PIC8259_MASTER_BASE+1, mask);
79              iodelay();
80          }
81          else {
82              int mask = inb(PIC8259_SLAVE_BASE+1);
83              iodelay();
84 
85              mask |= (1<<(irq - 8));
86              outb(PIC8259_SLAVE_BASE+1, mask);
87              iodelay();
88          }
89      }
90 }
```

Here is the call graph for this function:



4.92.1.4 void pic8259_unmask_irq (int *irq*)

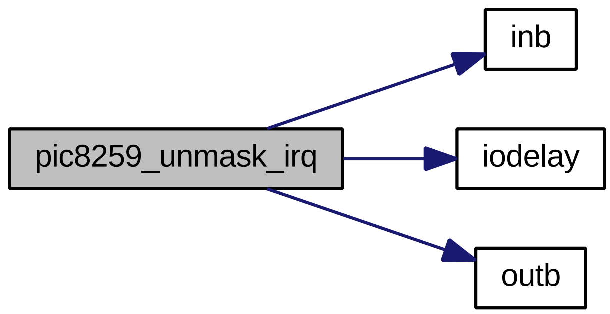
Definition at line 92 of file pic8259.c.

References inb(), iodelay(), outb(), PIC8259_CASCADE_INPUT, PIC8259_IRQ_COUNT, PIC8259_MASTER_BASE, and PIC8259_SLAVE_BASE.

Referenced by pic8259_eoi().

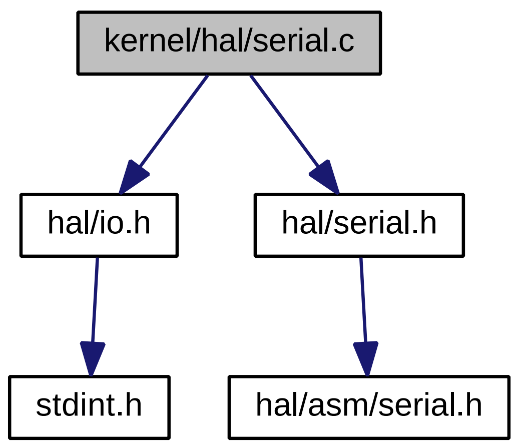
```
                                {
    if(irq < PIC8259_IRQ_COUNT) {
        int master_irq;

        if(irq < 8) {
            master_irq = irq;
        }
        else {
             /* Unmask interrupt in slave PIC. */
             int mask = inb(PIC8259_SLAVE_BASE+1);
             iodelay();

             mask &= ~(1<<(irq - 8));
             outb(PIC8259_SLAVE_BASE+1, mask);
             iodelay();

             /* We will also want to unmask the cascaded interrupt line in the
              * master PIC. */
             master_irq = PIC8259_CASCADE_INPUT;
         }

         if(irq != PIC8259_CASCADE_INPUT) {
             int mask = inb(PIC8259_MASTER_BASE+1);
             iodelay();

             mask &= ~(1<<master_irq);
             outb(PIC8259_MASTER_BASE+1, mask);
             iodelay();
         }
     }
}
```

Here is the call graph for this function:



4.93 kernel/hal/serial.c File Reference

```
#include <hal/io.h>
#include <hal/serial.h>
```
Include dependency graph for serial.c:



Functions

- void **serial_init** (int base_ioport, unsigned int baud_rate)
- void **serial_printn** (int base_ioport, const char ∗message, unsigned int n)
- void **serial_putc** (int base_ioport, char c)

4.93.1 Function Documentation

4.93.1.1 void serial_init (int *base_ioport,* unsigned int *baud_rate*)

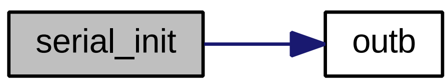
Definition at line 4 of file serial.c.

References outb(), SERIAL_REG_DIVISOR_HIGH, SERIAL_REG_DIVISOR_LOW, SERIAL_REG_FIFO_CTRL, SERIAL_REG_INTR_ENABLE, SERIAL_REG_LINE_CTRL, and SERIAL_REG_MODEM_CTRL.

Referenced by console_init().

```
{
    unsigned int divisor = 115200 / baud_rate;

    /* disable interrupts */
    outb(base_ioport + SERIAL_REG_INTR_ENABLE,  0);

    /* 8N1, enable DLAB to allow setting baud rate */
    outb(base_ioport + SERIAL_REG_LINE_CTRL,    0x83);

    /* set baud rate */
    outb(base_ioport + SERIAL_REG_DIVISOR_LOW,  (divisor & 0xff));
    outb(base_ioport + SERIAL_REG_DIVISOR_HIGH, ((divisor >> 8) & 0xff));

    /* 8N1, disable DLAB */
    outb(base_ioport + SERIAL_REG_LINE_CTRL,    0x03);

    /* enable and clear FIFO
     *
     * Receive FIFO trigger level is not relevant for us as we are only
     * transmitting. */
    outb(base_ioport + SERIAL_REG_FIFO_CTRL,    0x07);

    /* assert DTR and RTS */
    outb(base_ioport + SERIAL_REG_MODEM_CTRL,   0x03);
}
```

Here is the call graph for this function:



4.93.1.2 void serial_printn (int *base_ioport*, const char ∗ *message*, unsigned int *n*)

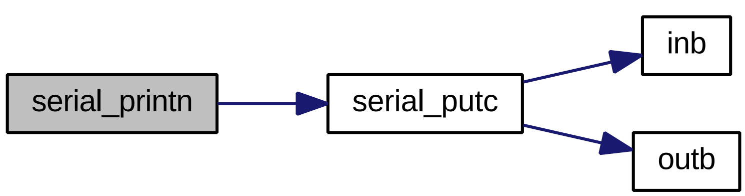
Definition at line 30 of file serial.c.

References serial_putc().

Referenced by console_printn().

```
30 {
31     int idx;
32
33     for(idx = 0; idx < n; ++idx) {
34         serial_putc(base_ioport, message[idx]);
35     }
36 }
```

Here is the call graph for this function:



4.93.1.3 void serial_putc (int *base_ioport*, char *c*)

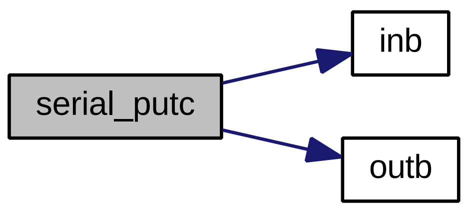
Definition at line 38 of file serial.c.

References inb(), outb(), SERIAL_REG_DATA_BUFFER, and SERIAL_REG_LINE_STATUS.

Referenced by console_putc(), and serial_printn().

```
38 {
39     /* wait for the UART to be ready to accept a new character */
40     while( (inb(base_ioport + SERIAL_REG_LINE_STATUS) & 0x20) == 0) {}
41 
42     outb(base_ioport + SERIAL_REG_DATA_BUFFER, c);
43 }
```

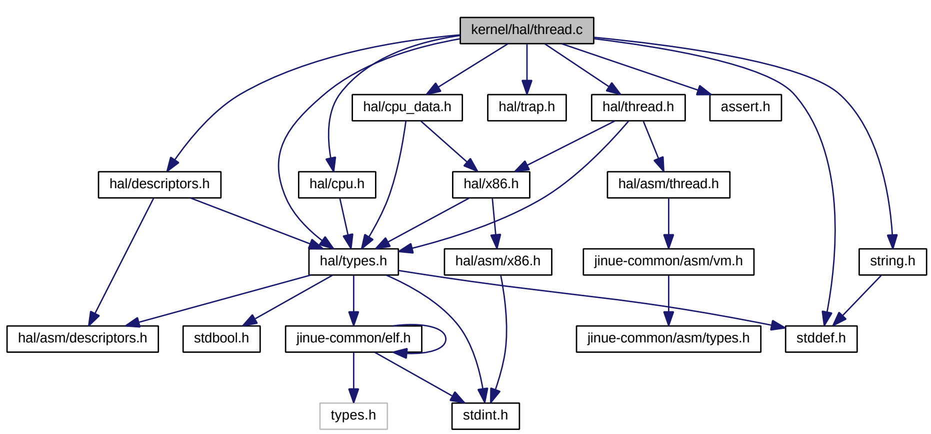
Here is the call graph for this function:



4.94 kernel/hal/thread.c File Reference

```
#include <hal/cpu.h>
#include <hal/cpu_data.h>
#include <hal/descriptors.h>
#include <hal/thread.h>
#include <hal/trap.h>
#include <hal/types.h>
#include <assert.h>
#include <stddef.h>
#include <string.h>
```

Include dependency graph for thread.c:



Functions

- void **thread_context_switch_stack** (**thread_context_t** ∗from_ctx, **thread_context_t** ∗to_ctx, **bool** destroy_from)
- **thread_t** ∗ **thread_page_init** (**addr_t** thread_page, **addr_t** entry, **addr_t** user_stack)
- void **thread_context_switch** (**thread_context_t** ∗from_ctx, **thread_context_t** ∗to_ctx, **bool** destroy_from)

4.94.1 Function Documentation

4.94.1.1 void thread_context_switch (thread_context_t ∗ *from_ctx*, thread_context_t ∗ *to_ctx*, bool *destroy_from*)

ASSERTION: to_ctx argument must not be NULL

ASSERTION: from_ctx argument must not be NULL if destroy_from is true

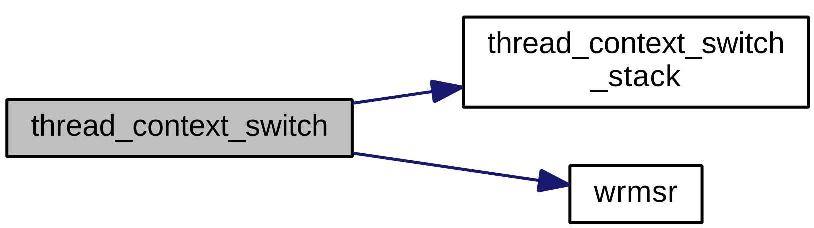
Definition at line 122 of file thread.c.

References assert, CPU_FEATURE_SYSENTER, tss_t::esp0, tss_t::esp1, tss_t::esp2, MSR_IA32_SYSENTER_ESP, NULL, thread_context_switch_stack(), and wrmsr().

Referenced by thread_switch().

```
                                           {

    assert(to_ctx != NULL);

    assert(from_ctx != NULL || ! destroy_from);

    /* nothing to do if this is already the current thread */
    if(from_ctx != to_ctx) {
        /* setup TSS with kernel stack base for this thread context */
        addr_t kernel_stack_base = get_kernel_stack_base(to_ctx);
        tss_t *tss = get_tss();

        tss->esp0 = kernel_stack_base;
        tss->esp1 = kernel_stack_base;
        tss->esp2 = kernel_stack_base;

        /* update kernel stack address for SYSENTER instruction */
        if(cpu_has_feature(CPU_FEATURE_SYSENTER)) {
            wrmsr(MSR_IA32_SYSENTER_ESP, (uint64_t)(uintptr_t)kernel_stack_base);
        }

        /* switch thread context stack */
        thread_context_switch_stack(from_ctx, to_ctx, destroy_from);
    }
}
```

Here is the call graph for this function:



4.94.1.2 void thread_context_switch_stack (thread_context_t ∗ *from_ctx*, thread_context_t ∗ *to_ctx*, bool *destroy_from*)

Referenced by thread_context_switch().

4.94.1.3 thread_t∗ thread_page_init (addr_t *thread_page,* addr_t *entry,* addr_t *user_stack*)

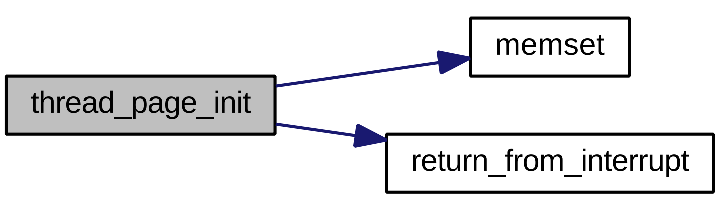
Definition at line 81 of file thread.c.

References trapframe_t::cs, trapframe_t::ds, trapframe_t::eflags, trapframe_t::eip, kernel_context_t::eip, trapframe_t::es, trapframe_t::esp, trapframe_t::fs, GDT_USER_CODE, GDT_USER_DATA, trapframe_t::gs, thread_context_t::local_storage_addr, memset(), NULL, return_from_interrupt(), RPL_USER, thread_context_t::saved_stack_pointer, SEG_SELECTOR, trapframe_t::ss, and thread_t::thread_ctx.

Referenced by thread_create(), and thread_create_boot().

```
                                   {

    /* initialize fields */
    thread_t *thread                = (thread_t *)thread_page;
    thread_context_t *thread_ctx    = &thread->thread_ctx;

    thread_ctx->local_storage_addr  = NULL;

    /* setup stack for initial return to user space */
    void *kernel_stack_base = get_kernel_stack_base(thread_ctx);

    trapframe_t *trapframe = (trapframe_t *)kernel_stack_base - 1;

    memset(trapframe, 0, sizeof(trapframe_t));

    trapframe->eip      = (uint32_t)entry;
    trapframe->esp      = (uint32_t)user_stack;
    trapframe->eflags   = 2;
    trapframe->cs       = SEG_SELECTOR(GDT_USER_CODE, RPL_USER);
    trapframe->ss       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);
    trapframe->ds       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);
    trapframe->es       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);
    trapframe->fs       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);
    trapframe->gs       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);

    kernel_context_t *kernel_context = (kernel_context_t *)trapframe - 1;

    memset(kernel_context, 0, sizeof(kernel_context_t));

    /* This is the address to which thread_context_switch_stack() will return. */
    kernel_context->eip = (uint32_t)return_from_interrupt;

    /* set thread stack pointer */
    thread_ctx->saved_stack_pointer = (addr_t)kernel_context;

    return thread;
}
```

Here is the call graph for this function:

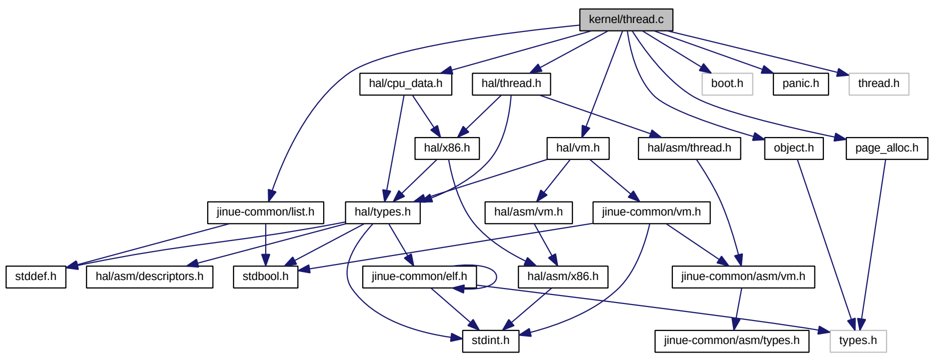


4.95 kernel/thread.c File Reference

```
#include <jinue-common/list.h>
```

```
#include <hal/cpu_data.h>
#include <hal/thread.h>
#include <hal/vm.h>
#include <boot.h>
#include <object.h>
#include <page_alloc.h>
#include <panic.h>
#include <thread.h>
```

Include dependency graph for thread.c:



Functions

- **thread_t** ∗ **thread_create** (**process_t** ∗process, **addr_t** entry, **addr_t** user_stack)
- **thread_t** ∗ **thread_create_boot** (**process_t** ∗process, **addr_t** entry, **addr_t** user_stack, **boot_alloc_t** ∗boot_alloc)
- void **thread_destroy** (**thread_t** ∗thread)
- void **thread_ready** (**thread_t** ∗thread)
- void **thread_switch** (**thread_t** ∗from_thread, **thread_t** ∗to_thread, **bool** blocked, **bool** do_destroy)
- void **thread_yield_from** (**thread_t** ∗from_thread, **bool** blocked, **bool** do_destroy)

4.95.1 Function Documentation

4.95.1.1 thread_t∗ thread_create (process_t ∗ *process*, addr_t *entry*, addr_t *user_stack*)

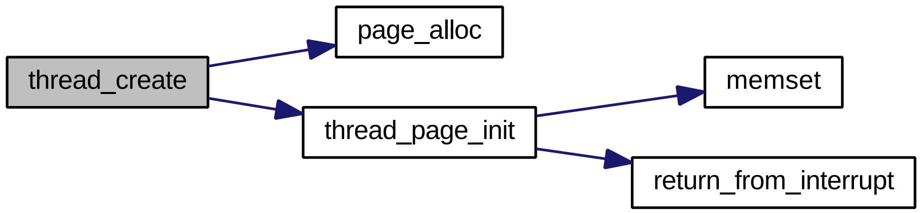
Definition at line 56 of file thread.c.

References NULL, page_alloc(), and thread_page_init().

Referenced by dispatch_syscall().

```
                                     {

    void *thread_page = page_alloc();

    if(thread_page == NULL) {
        return NULL;
    }

    thread_t *thread = thread_page_init(thread_page, entry, user_stack);
    thread_init(thread, process);

    return thread;
}
```

Here is the call graph for this function:



4.95.1.2 thread_t∗ thread_create_boot (process_t ∗ *process*, addr_t *entry*, addr_t *user_stack*, boot_alloc_t ∗ *boot_alloc*)

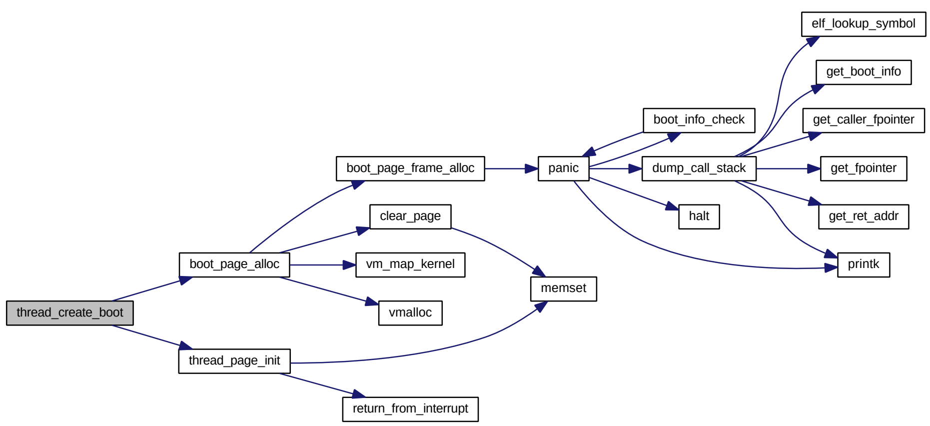
Definition at line 73 of file thread.c.

References boot_page_alloc(), and thread_page_init().

Referenced by kmain().

```
                                    {

    /* The kernel panics if this allocation fails. */
    void *thread_page    = boot_page_alloc(boot_alloc);
    thread_t *thread     = thread_page_init(thread_page, entry, user_stack);
    thread_init(thread, process);

    return thread;
}
```

Here is the call graph for this function:



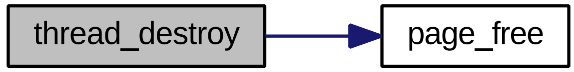
4.95.1.3 void thread_destroy (thread_t ∗ *thread*)

Definition at line 88 of file thread.c.

References page_free(), and PAGE_MASK.

```
                                    {
    void * thread_page = (void *)((uintptr_t)thread & ~PAGE_MASK);
    page_free(thread_page);
}
```

Here is the call graph for this function:



4.95.1.4 void thread_ready (thread_t ∗ *thread*)

Definition at line 93 of file thread.c.

References thread_t::thread_list.

Referenced by thread_switch().

```
93                                   {
94     /* add thread to the tail of the ready list to give other threads a chance
95      * to run */
96     jinue_list_enqueue(&ready_list, &thread->thread_list);
97 }
```

4.95.1.5 void thread_switch (thread_t ∗ *from_thread,* thread_t ∗ *to_thread,* bool *blocked,* bool *do_destroy*)

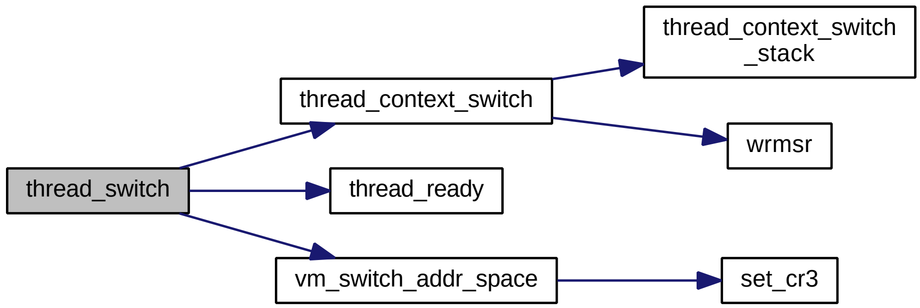
Definition at line 99 of file thread.c.

References process_t::addr_space, NULL, thread_t::process, thread_context_switch(), thread_t::thread_ctx, thread_ready(), and vm_switch_addr_space().

Referenced by ipc_receive(), ipc_reply(), ipc_send(), and thread_yield_from().

```
103                              {
104 
105     if(to_thread != from_thread) {
106         thread_context_t    *from_context;
107         process_t           *from_process;
108 
109         if(from_thread == NULL) {
110             from_context = NULL;
111             from_process = NULL;
112         }
113         else {
114             from_context = &from_thread->thread_ctx;
115             from_process = from_thread->process;
116 
117             /* Put the the thread we are switching away from (the current thread)
118              * back into the ready list, unless it just blocked or it is being
119              * destroyed. */
120             if(! (do_destroy || blocked)) {
121                 thread_ready(from_thread);
122             }
123         }
124 
125         if(from_process != to_thread->process) {
126             vm_switch_addr_space(
127                     &to_thread->process->addr_space,
128                     get_cpu_local_data()
129             );
130         }
131 
132         thread_context_switch(
133             from_context,
134             &to_thread->thread_ctx,
135             do_destroy);
136     }
137 }
```

Here is the call graph for this function:



4.95.1.6 void thread_yield_from (**thread_t** ∗ *from_thread,* **bool** *blocked,* **bool** *do_destroy*)

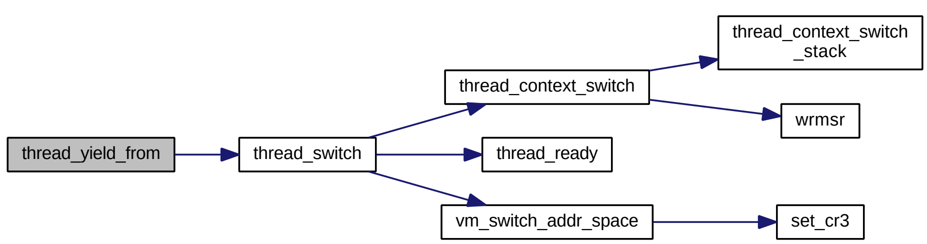
Definition at line 163 of file thread.c.

References thread_switch().

Referenced by dispatch_syscall(), ipc_receive(), ipc_send(), and kmain().

```
163                                                                              {
164     bool from_can_run = ! (blocked || do_destroy);
165 
166     thread_switch(
167             from_thread,
168             reschedule(from_thread, from_can_run),
169             blocked,
170             do_destroy);
171 }
```

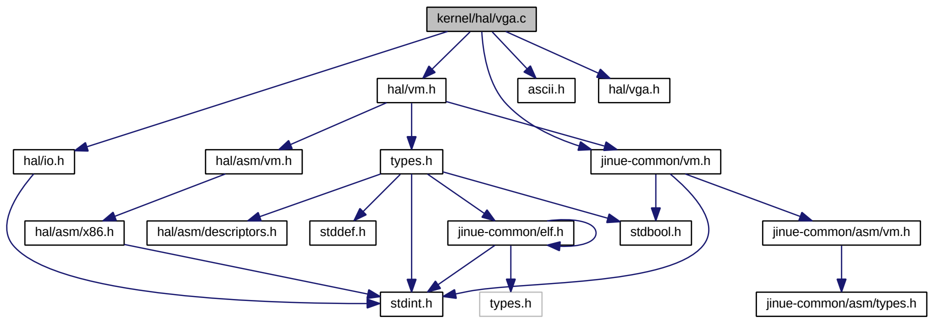
Here is the call graph for this function:



4.96 kernel/hal/vga.c File Reference

```
#include <jinue-common/vm.h>
#include <ascii.h>
#include <hal/io.h>
#include <hal/vga.h>
#include <hal/vm.h>
```

Include dependency graph for vga.c:



Functions

- void **vga_init** (void)
- void **vga_set_base_addr** (void ∗base_addr)
- 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, int colour)
- void **vga_printn** (const char ∗message, unsigned int n, int colour)
- void **vga_putc** (char c, int colour)

4.96.1 Function Documentation

4.96.1.1 void vga_clear (void)

Definition at line 65 of file vga.c.

References VGA_COLOR_ERASE, VGA_LINES, and VGA_WIDTH.

Referenced by vga_init().

```
65                      {
66     unsigned int idx = 0;
67
68     while( idx < (VGA_LINES * VGA_WIDTH * 2) ) {
69         video_base_addr[idx++] = 0x20;
70         video_base_addr[idx++] = VGA_COLOR_ERASE;
71     }
72 }
```

4.96.1.2 vga_pos_t vga_get_cursor_pos (void)

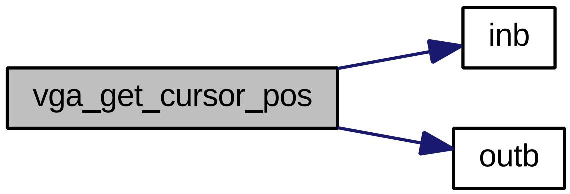
Definition at line 89 of file vga.c.

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

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

```
                                {
    unsigned char h, l;

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

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

Here is the call graph for this function:



4.96.1.3 void vga_init (void)

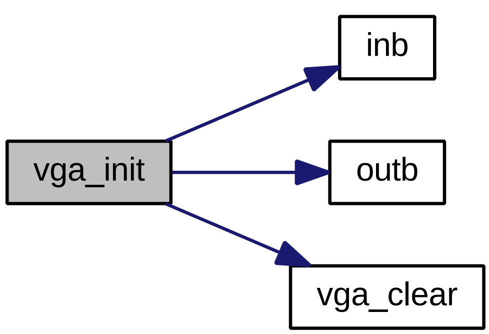
Definition at line 43 of file vga.c.

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

Referenced by console_init().

```
43                      {
44     unsigned char data;
45
46     /* Set address select bit in a known state: CRTC regs at 0x3dx */
47     data = inb(VGA_MISC_OUT_RD);
48     data |= 1;
49     outb(VGA_MISC_OUT_WR, data);
50
51     /* Move cursor to line 0 col 0 */
52     outb(VGA_CRTC_ADDR, 0x0e);
53     outb(VGA_CRTC_DATA, 0x0);
54     outb(VGA_CRTC_ADDR, 0x0f);
55     outb(VGA_CRTC_DATA, 0x0);
56
57     /* Clear the screen */
58     vga_clear();
59 }
```

Here is the call graph for this function:



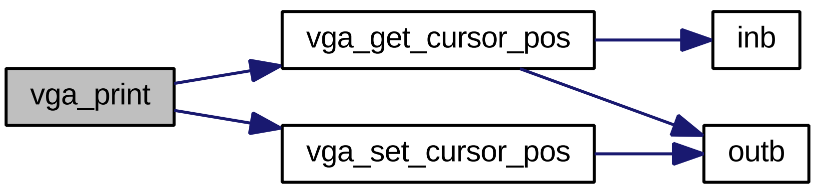
4.96.1.4 void vga_print (const char * *message*, int *colour*)

Definition at line 111 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

```
111                                             {
112     unsigned short int pos = vga_get_cursor_pos();
113     char c;
114 
115     while( (c = *(message++)) ) {
116         pos = vga_raw_putc(c, pos, colour);
117     }
118 
119     vga_set_cursor_pos(pos);
120 }
```

Here is the call graph for this function:



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

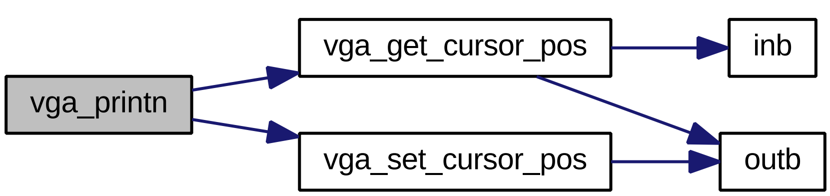
Definition at line 122 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by console_printn().

```
122                                                              {
123     vga_pos_t pos = vga_get_cursor_pos();
124     char c;
125 
126     while(n) {
127         c = *(message++);
128         pos = vga_raw_putc(c, pos, colour);
129         --n;
130     }
131 
132     vga_set_cursor_pos(pos);
133 }
```

Here is the call graph for this function:



4.96.1.6 void vga_putc (char *c*, int *colour*)

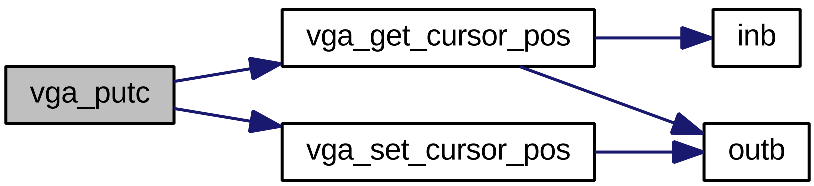
Definition at line 135 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by console_putc().

```
135                                   {
136     vga_pos_t pos = vga_get_cursor_pos();
137 
138     pos = vga_raw_putc(c, pos, colour);
139 
140     vga_set_cursor_pos(pos);
141 }
```

Here is the call graph for this function:



4.96.1.7 void vga_scroll (void)

Definition at line 74 of file vga.c.

References VGA_COLOR_ERASE, VGA_LINES, and VGA_WIDTH.

```
74                      {
75     unsigned char *di = video_base_addr;
76     unsigned char *si = video_base_addr + 2 * VGA_WIDTH;
77     unsigned int idx;
78 
79     for(idx = 0; idx < 2 * VGA_WIDTH * (VGA_LINES - 1); ++idx) {
80         *(di++) = *(si++);
81     }
82 
83     for(idx = 0; idx < VGA_WIDTH; ++idx) {
84         *(di++) = 0x20;
85         *(di++) = VGA_COLOR_ERASE;
86     }
87 }
```

4.96.1.8 void vga_set_base_addr (void * *base_addr*)

Definition at line 61 of file vga.c.

Referenced by vm_boot_init().

```
61                                     {
62     video_base_addr = base_addr;
63 }
```

4.96.1.9 void vga_set_cursor_pos (vga_pos_t *pos*)

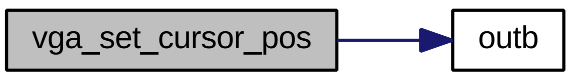
Definition at line 100 of file vga.c.

References outb(), VGA_CRTC_ADDR, and VGA_CRTC_DATA.

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

```
100                                     {
101     unsigned char h = pos >> 8;
102     unsigned char l = pos;
103 
104     outb(VGA_CRTC_ADDR, 0x0e);
105     outb(VGA_CRTC_DATA, h);
106     outb(VGA_CRTC_ADDR, 0x0f);
107     outb(VGA_CRTC_DATA, l);
108 }
```

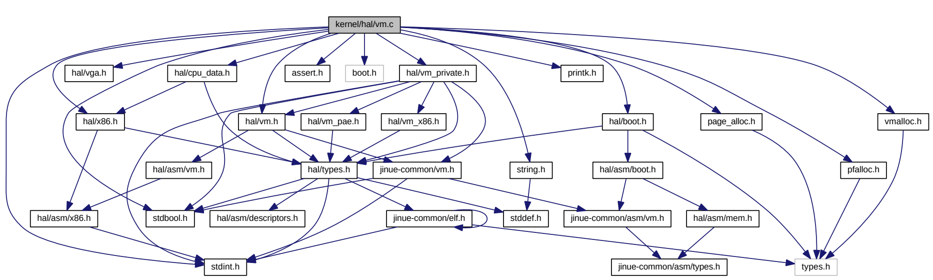
Here is the call graph for this function:



4.97 kernel/hal/vm.c File Reference

```
#include <hal/boot.h>
#include <hal/cpu_data.h>
#include <hal/vga.h>
#include <hal/vm.h>
#include <hal/vm_private.h>
#include <hal/x86.h>
#include <assert.h>
#include <boot.h>
#include <page_alloc.h>
#include <pfalloc.h>
#include <printk.h>
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include <vmalloc.h>
```

Include dependency graph for vm.c:



Functions

- void **vm_boot_init** (const **boot_info_t** ∗**boot_info**, **bool** use_pae, **cpu_data_t** ∗cpu_data, **boot_alloc_t** ∗boot-_alloc)
- void **vm_boot_postinit** (const **boot_info_t** ∗**boot_info**, **boot_alloc_t** ∗boot_alloc, **bool** use_pae)
- void **vm_map_kernel** (**addr_t** vaddr, **kern_paddr_t** paddr, int flags)
- void **vm_map_user** (**addr_space_t** ∗addr_space, **addr_t** vaddr, **user_paddr_t** paddr, int flags)
- void **vm_unmap_kernel** (**addr_t** addr)
- void **vm_unmap_user** (**addr_space_t** ∗addr_space, **addr_t** addr)
- **kern_paddr_t vm_lookup_kernel_paddr** (**addr_t** addr)
- void **vm_change_flags** (**addr_space_t** ∗addr_space, **addr_t** addr, int flags)
- void **vm_map_early** (**addr_t** vaddr, **kern_paddr_t** paddr, int flags)
- **kern_paddr_t vm_clone_page_directory** (**kern_paddr_t** template_paddr, unsigned int start_index)
- **addr_space_t** ∗ **vm_create_addr_space** (**addr_space_t** ∗addr_space)
- void **vm_init_initial_page_directory** (**pte_t** ∗page_directory, **boot_alloc_t** ∗boot_alloc, unsigned int start_index, unsigned int end_index, **bool** first_directory)
- **addr_space_t** ∗ **vm_create_initial_addr_space** (**bool** use_pae, **boot_alloc_t** ∗boot_alloc)
- void **vm_destroy_page_directory** (**kern_paddr_t** pgdir_paddr, unsigned int from_index, unsigned int to_index)
- void **vm_destroy_addr_space** (**addr_space_t** ∗addr_space)
- void **vm_switch_addr_space** (**addr_space_t** ∗addr_space, **cpu_data_t** ∗cpu_data)

Variables

- **pte_t** ∗ **global_page_tables**
- **addr_space_t initial_addr_space**
- **size_t page_table_entries**

4.97.1 Function Documentation

4.97.1.1 void vm_boot_init (const boot_info_t ∗ *boot_info*, bool *use_pae*, cpu_data_t ∗ *cpu_data*, boot_alloc_t ∗ *boot_alloc*)

below this point, it is no longer safe to call pfalloc_early()

Definition at line 57 of file vm.c.

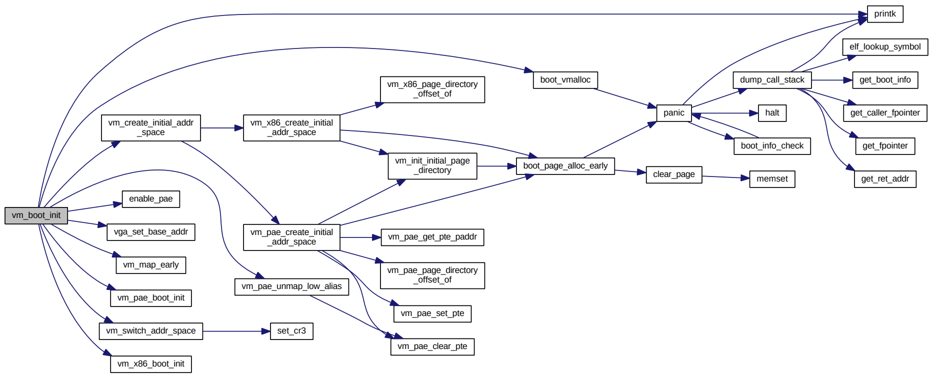
References boot_vmalloc(), addr_space_t::cr3, EARLY_PTR_TO_PHYS_ADDR, enable_pae(), boot_info_t::image_-start, boot_alloc_t::its_early, PAGE_SIZE, printk(), vga_set_base_addr(), VGA_TEXT_VID_BASE, VGA_TEXT_VID-_TOP, vm_create_initial_addr_space(), VM_FLAG_KERNEL, VM_FLAG_READ_WRITE, vm_map_early(), vm_pae_-boot_init(), vm_pae_unmap_low_alias(), vm_switch_addr_space(), and vm_x86_boot_init().

Referenced by hal_init().

```
                                          {

    addr_t addr;

    if(use_pae) {
        printk("Enabling Physical Address Extension (PAE).\n");
        vm_pae_boot_init();
    }
    else {
        vm_x86_boot_init();
    }

    pgtable_format_pae = use_pae;

```

```
    /* create initial address space */
    addr_space_t *addr_space = vm_create_initial_addr_space(use_pae, boot_alloc);

    boot_alloc->its_early = false;

    /* perform 1:1 mapping of kernel image and data */
    for(addr = (addr_t)boot_info->image_start; addr < boot_alloc->kernel_vm_top; addr +=
   PAGE_SIZE) {
        vm_map_early(addr, EARLY_PTR_TO_PHYS_ADDR(addr), VM_FLAG_KERNEL |
   VM_FLAG_READ_WRITE);
    }

    /* map VGA text buffer in the new address space
     *
     * This is a good place to do this because:
     *
     * 1) It is our last chance to allocate a continuous region of virtual memory.
     *    Once the page allocator is initialized (see call to vm_alloc_init_allocator()
     *    below) and we start using vm_alloc() to allocate memory, pages can only
     *    be allocated one at a time.
     *
     * 2) Doing this last makes things simpler because this is the only place where
     *    we have to allocate a continuous region of virtual memory but no physical
     *    memory to back it. To allocate it, we just have to increase kernel_vm_top,
     *    which represents the end of the virtual memory region that is used by the
     *    kernel. */
    addr_t vga_text_base;
    kern_paddr_t paddr = VGA_TEXT_VID_BASE;

    while(paddr < VGA_TEXT_VID_TOP) {
        /* Pages allocated by successive calls to vmalloc_boot() are guaranteed
         * to be contiguous. */
        addr = boot_vmalloc(boot_alloc);

        if(paddr == VGA_TEXT_VID_BASE) {
            /* First iteration */
            vga_text_base = addr;
        }

        vm_map_early(addr, paddr, VM_FLAG_KERNEL | VM_FLAG_READ_WRITE);
        paddr           += PAGE_SIZE;
    }

    /* remap VGA text buffer
     *
     * Note: after the call to vga_set_base_addr() below until we switch to the
     * new address space, VGA output is not possible. Calling printk() will cause
     * a kernel panic due to a page fault (and the panic handler calls printk()). */
    printk("Remapping text video memory at 0x%x\n", vga_text_base);

    vga_set_base_addr(vga_text_base);

    if(use_pae) {
        /* If we are enabling PAE, this is where the switch to the new page
         * tables actually happens instead of at the call to vm_switch_addr_space()
         * as would be expected. */
        enable_pae(addr_space->cr3);

        /* Now that PAE has been enabled, there is no need to ever disable paging
         * again, so the low alias for the first 2MB of RAM can be unmapped. This
         * is only relevant for PAE because, for the non-PAE case, this low alias
         * is just never set up in the initial address space in the first place,
         * which means there is no longer a low alias once vm_switch_addr_space()
         * is called below.
         *
         * This call to vm_pae_unmap_low_alias() does not do any TLB invalidation
         * but this is fine because the call to vm_switch_addr_space() below
         * reloads CR3.*/
        vm_pae_unmap_low_alias(addr_space);
    }

    /* switch to new address space */
    vm_switch_addr_space(addr_space, cpu_data);
}
```

Here is the call graph for this function:



4.97.1.2 void vm_boot_postinit (const boot_info_t * *boot_info,* boot_alloc_t * *boot_alloc,* bool *use_pae*)

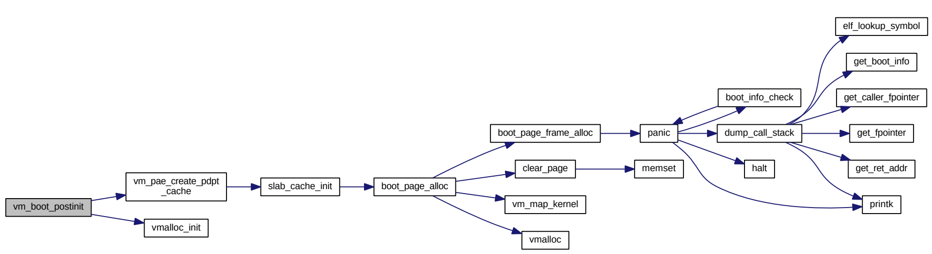
Definition at line 149 of file vm.c.

References KERNEL_IMAGE_END, KERNEL_PREALLOC_LIMIT, MB, vm_pae_create_pdpt_cache(), and vmalloc_-init().

Referenced by hal_init().

```
149                                                                                       {
150     /* initialize global page allocator (region starting at KLIMIT)
151      *
152      * TODO Some work needs to be done in the page allocator to support allocating
153      * up to the top of memory (i.e. 0x100000000, which cannot be represented on
154      * 32 bits). In the mean time, we leave a 4MB (one block) gap. */
155     vmalloc_init(
156             (addr_t)KERNEL_IMAGE_END,
157             (addr_t)0 - 4 * MB,
158             (addr_t)KERNEL_PREALLOC_LIMIT,
159             boot_alloc);
160
161     /* create slab cache to allocate PDPTs
162      *
163      * This must be done after the global page allocator has been initialized
164      * because the slab allocator needs to allocate a slab to allocate the new
165      * slab cache on the slab cache cache.
166      *
167      * This must be done before the first time vm_create_addr_space() is called. */
168     if(use_pae) {
169         vm_pae_create_pdpt_cache(boot_alloc);
170     }
171 }
```

Here is the call graph for this function:



4.97.1.3 void vm_change_flags (addr_space_t * *addr_space*, addr_t *addr*, int *flags*)

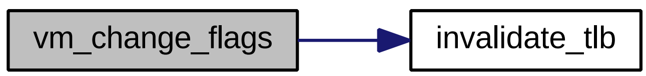
ASSERTION: there is a page table entry marked present for this address

Definition at line 463 of file vm.c.

References assert, invalidate_tlb(), and NULL.

```
463                                                                         {
464     pte_t *pte = vm_lookup_page_table_entry(addr_space, addr, false);
465
467     assert(pte != NULL && (get_pte_flags(pte) & VM_FLAG_PRESENT));
468
469     /* perform the flags change */
470     set_pte_flags(pte, flags | VM_FLAG_PRESENT);
471
472     vm_free_page_table_entry(addr, pte);
473
474     /* invalidate TLB entry for the affected page */
475     invalidate_tlb(addr);
476 }
```

Here is the call graph for this function:



4.97.1.4 kern_paddr_t vm_clone_page_directory (kern_paddr_t *template_paddr*, unsigned int *start_index*)

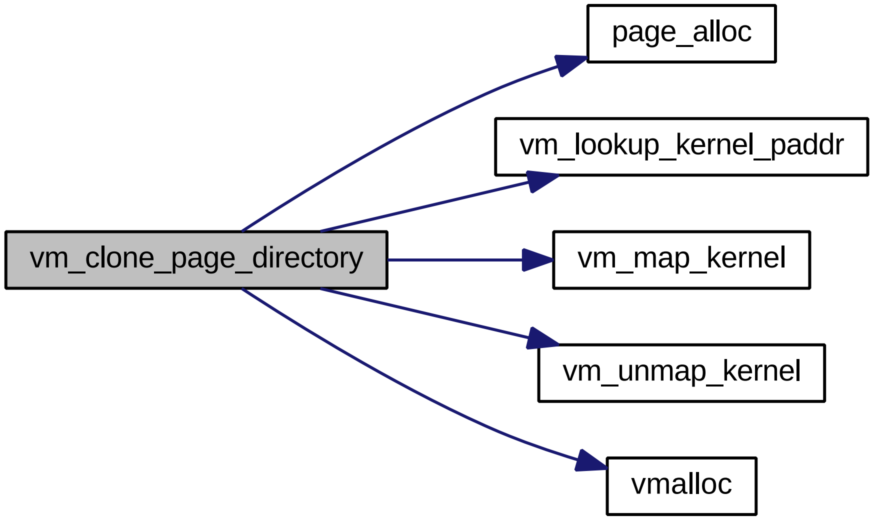
Definition at line 489 of file vm.c.

References page_alloc(), page_table_entries, VM_FLAG_READ_WRITE, vm_lookup_kernel_paddr(), vm_map_kernel(), vm_unmap_kernel(), and vmalloc().

Referenced by vm_x86_create_addr_space().

```
489                                                                                            {
490     unsigned int      idx;
491
492     /* Allocate new page directory.
493      *
494      * TODO handle allocation failure */
495     pte_t *page_directory = (pte_t *)page_alloc();
496
497     /* map page directory template */
498     pte_t *template = (pte_t *)vmalloc();
499     vm_map_kernel((addr_t)template, template_paddr, VM_FLAG_READ_WRITE);
500
501     /* clear all entries below index start_index */
502     for(idx = 0; idx < start_index; ++idx) {
503         clear_pte( get_pte_with_offset(page_directory, idx) );
504     }
505
506     /* copy entries from template for indexes start_index and above */
507     for(idx = start_index; idx < page_table_entries; ++idx) {
508         copy_pte(
509             get_pte_with_offset(page_directory, idx),
510             get_pte_with_offset(template, idx)
511         );
512     }
513
514     vm_unmap_kernel((addr_t)page_directory);
515     vm_unmap_kernel((addr_t)template);
516
517     return vm_lookup_kernel_paddr((addr_t)page_directory);
518 }
```

Here is the call graph for this function:



4.97.1.5 addr_space_t* vm_create_addr_space (addr_space_t * *addr_space*)

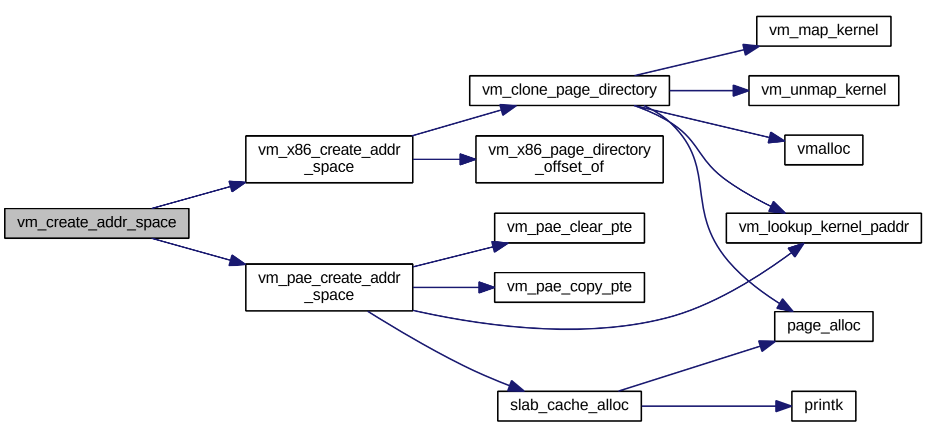
Definition at line 520 of file vm.c.

References vm_pae_create_addr_space(), and vm_x86_create_addr_space().

Referenced by process_create().

```
520                                                                    {
521     if(pgtable_format_pae) {
522         return vm_pae_create_addr_space(addr_space);
523     }
524     else {
525         return vm_x86_create_addr_space(addr_space);
526     }
527 }
```

Here is the call graph for this function:



4.97.1.6 addr_space_t∗ vm_create_initial_addr_space (bool *use_pae,* boot_alloc_t ∗ *boot_alloc*)

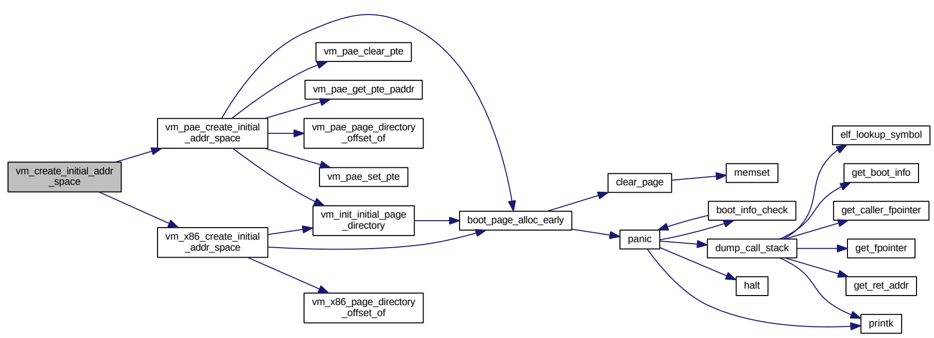
Definition at line 572 of file vm.c.

References vm_pae_create_initial_addr_space(), and vm_x86_create_initial_addr_space().

Referenced by vm_boot_init().

```
574                                       {
575
576     if(use_pae) {
577         return vm_pae_create_initial_addr_space(boot_alloc);
578     }
579     else {
580         return vm_x86_create_initial_addr_space(boot_alloc);
581     }
582 }
```

Here is the call graph for this function:



4.97.1.7 void vm_destroy_addr_space (addr_space_t ∗ *addr_space*)

ASSERTION: address space must not be NULL

ASSERTION: the initial address space should not be destroyed

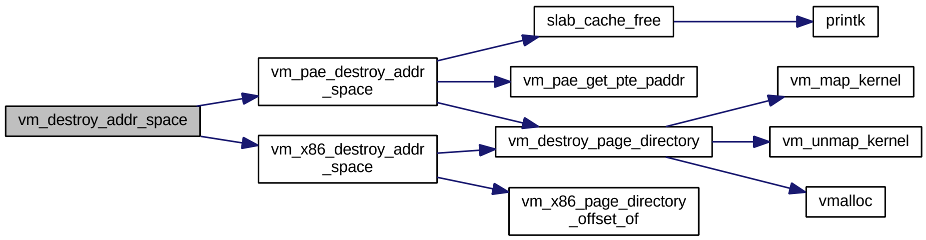
ASSERTION: the current address space should not be destroyed

Definition at line 603 of file vm.c.

References assert, NULL, vm_pae_destroy_addr_space(), and vm_x86_destroy_addr_space().

```
603                                                     {
605     assert(addr_space != NULL);
606
608     assert(addr_space != &initial_addr_space);
609
611     assert( addr_space != get_current_addr_space() );
612
613     if(pgtable_format_pae) {
614         vm_pae_destroy_addr_space(addr_space);
615     }
616     else {
617         vm_x86_destroy_addr_space(addr_space);
618     }
619 }
```

Here is the call graph for this function:



4.97.1.8 void vm_destroy_page_directory (kern_paddr_t *pgdir_paddr,* unsigned int *from_index,* unsigned int *to_index*)

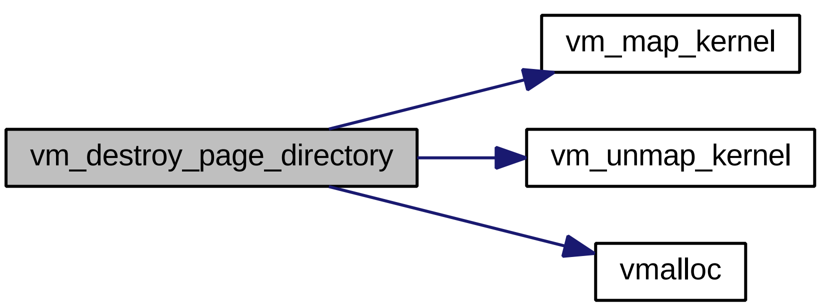
Definition at line 584 of file vm.c.

References pffree, VM_FLAG_READ_WRITE, vm_map_kernel(), vm_unmap_kernel(), and vmalloc().

Referenced by vm_pae_destroy_addr_space(), and vm_x86_destroy_addr_space().

```
584                                                                                              {
585     unsigned int idx;
586 
587     pte_t *page_directory = (pte_t *)vmalloc();
588     vm_map_kernel((addr_t)page_directory, pgdir_paddr, VM_FLAG_READ_WRITE);
589 
590     /* be careful not to free the kernel page tables */
591     for(idx = from_index; idx < to_index; ++idx) {
592         pte_t *pte = get_pte_with_offset(page_directory, idx);
593 
594         if(get_pte_flags(pte) & VM_FLAG_PRESENT) {
595             pffree( get_pte_paddr(pte) );
596         }
597     }
598 
599     vm_unmap_kernel((addr_t)page_directory);
600     pffree(pgdir_paddr);
601 }
```

Here is the call graph for this function:



4.97.1.9 void vm_init_initial_page_directory (pte_t ∗ *page_directory,* boot_alloc_t ∗ *boot_alloc,* unsigned int *start_index,* unsigned int *end_index,* bool *first_directory*)

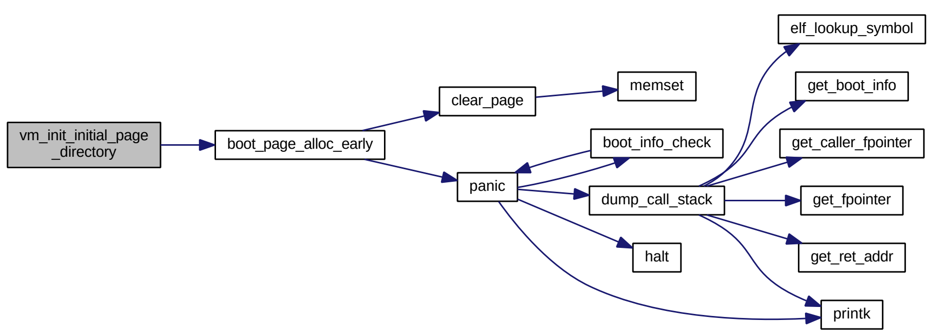
Definition at line 529 of file vm.c.

References boot_page_alloc_early(), EARLY_PTR_TO_PHYS_ADDR, page_table_entries, and VM_FLAG_READ_W-RITE.

Referenced by vm_pae_create_initial_addr_space(), and vm_x86_create_initial_addr_space().

```
                                          {

    unsigned int idx, idy;

    /* Allocate page tables and initialize page directory entries. */
    for(idx = 0; idx < page_table_entries; ++idx) {
        if(idx < start_index || idx >= end_index) {
            /* Clear page directory entries for user space and non-preallocated
             * kernel page tables. */
            clear_pte( get_pte_with_offset(page_directory, idx) );
        }
        else {
            /* Allocate page tables for kernel data/code region.
             *
             * Note that the use of pfalloc_early() here guarantees that the
             * page tables are allocated contiguously, and that they keep the
             * same address once paging is enabled. */
            pte_t *page_table = (pte_t *)boot_page_alloc_early(boot_alloc);

            if(first_directory && idx == start_index) {
                /* remember the address of the first page table for use by
                 * vm_map() later */
                global_page_tables = page_table;
            }

            set_pte(
                get_pte_with_offset(page_directory, idx),
                EARLY_PTR_TO_PHYS_ADDR(page_table),
                VM_FLAG_PRESENT | VM_FLAG_READ_WRITE);

            /* clear page table */
            for(idy = 0; idy < page_table_entries; ++idy) {
                clear_pte( get_pte_with_offset(page_table, idy) );
            }
        }
    }
}
```

Here is the call graph for this function:



4.97.1.10 kern_paddr_t vm_lookup_kernel_paddr (addr_t *addr*)

ASSERTION: there is a page table entry marked present for this address

Definition at line 450 of file vm.c.

References assert, and NULL.

Referenced by hal_init(), remove_page_frame(), vm_clone_page_directory(), and vm_pae_create_addr_space().

```
450                                                 {
451     pte_t *pte = vm_lookup_page_table_entry(NULL, addr, false);
452 
454     assert(pte != NULL && (get_pte_flags(pte) & VM_FLAG_PRESENT));
455 
456     kern_paddr_t paddr = (kern_paddr_t)get_pte_paddr(pte);
457 
458     vm_free_page_table_entry(addr, pte);
459 
460     return paddr;
461 }
```

4.97.1.11 void vm_map_early (addr_t *vaddr*, kern_paddr_t *paddr*, int *flags*)

ASSERTION: we are within the mapping set up by the setup code

ASSERTION: we assume vaddr is aligned on a page boundary

Definition at line 478 of file vm.c.

References assert, EARLY_VIRT_TO_PHYS, page_number_of, and page_offset_of.

Referenced by vm_boot_init().

```
478                                                              {
480     assert( is_early_pointer(vaddr) );
481 
483     assert( page_offset_of(vaddr) == 0 );
484 
485     pte_t *pte = get_pte_with_offset(global_page_tables, page_number_of(
      EARLY_VIRT_TO_PHYS((uintptr_t)vaddr) ));
486     set_pte(pte, paddr, flags | VM_FLAG_PRESENT);
487 }
```

4.97.1.12 void vm_map_kernel (addr_t *vaddr*, kern_paddr_t *paddr*, int *flags*)

Definition at line 434 of file vm.c.

References NULL, and VM_FLAG_KERNEL.

Referenced by add_page_frame(), boot_page_alloc(), boot_page_alloc_image(), elf_setup_stack(), vm_clone_page_-directory(), vm_destroy_page_directory(), vm_pae_lookup_page_directory(), and vm_x86_lookup_page_directory().

```
434                                                               {
435     vm_map(NULL, vaddr, paddr, flags | VM_FLAG_KERNEL);
436 }
```

4.97.1.13 void vm_map_user (addr_space_t * *addr_space*, addr_t *vaddr*, user_paddr_t *paddr*, int *flags*)

Definition at line 438 of file vm.c.

References VM_FLAG_USER.

Referenced by elf_load(), and elf_setup_stack().

```
438                                                                                        {
439     vm_map(addr_space, vaddr, paddr, flags | VM_FLAG_USER);
440 }
```

4.97.1.14 void vm_switch_addr_space (addr_space_t ∗ *addr_space*, cpu_data_t ∗ *cpu_data*)

Definition at line 621 of file vm.c.

References addr_space_t::cr3, cpu_data_t::current_addr_space, and set_cr3().

Referenced by thread_switch(), and vm_boot_init().

```
621                                                                  {
622     set_cr3(addr_space->cr3);
623 
624     cpu_data->current_addr_space = addr_space;
625 }
```

Here is the call graph for this function:

4.97.1.15 void vm_unmap_kernel (addr_t *addr*)

Definition at line 442 of file vm.c.

References NULL.

Referenced by elf_setup_stack(), remove_page_frame(), vm_clone_page_directory(), and vm_destroy_page_-directory().

```
442                              {
443     vm_unmap(NULL, addr);
444 }
```

4.97.1.16 void vm_unmap_user (addr_space_t ∗ *addr_space*, addr_t *addr*)

Definition at line 446 of file vm.c.

```
446                                                  {
447     vm_unmap(addr_space, addr);
448 }
```

4.97.2 Variable Documentation

4.97.2.1 pte_t∗ global_page_tables

Definition at line 49 of file vm.c.

4.97.2.2 addr_space_t initial_addr_space

Definition at line 51 of file vm.c.

Referenced by process_create_initial(), vm_pae_create_addr_space(), vm_pae_create_initial_addr_space(), vm_x86-_create_addr_space(), and vm_x86_create_initial_addr_space().

4.97.2.3 size_t page_table_entries

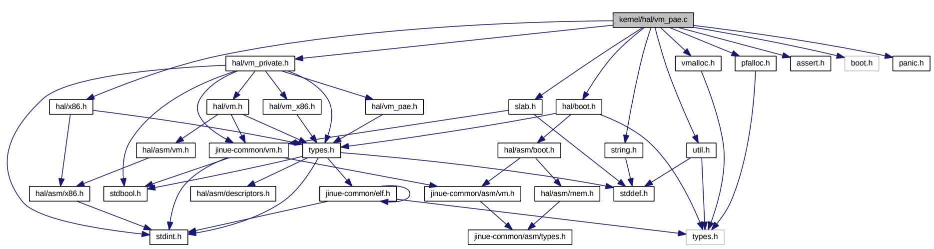
Definition at line 53 of file vm.c.

Referenced by vm_clone_page_directory(), vm_init_initial_page_directory(), vm_pae_boot_init(), vm_pae_create_initial_addr_space(), and vm_x86_boot_init().

4.98 kernel/hal/vm_pae.c File Reference

```
#include <hal/vm_private.h>
#include <hal/boot.h>
#include <hal/x86.h>
#include <assert.h>
#include <boot.h>
#include <panic.h>
#include <pfalloc.h>
#include <slab.h>
#include <string.h>
#include <util.h>
#include <vmalloc.h>
```

Include dependency graph for vm_pae.c:



Data Structures

- struct **pte_t**
- struct **pdpt_t**

Macros

- #define **PDPT_BITS** 2

  number of address bits that encode the PDPT offset
- #define **PDPT_ENTRIES** (1 << **PDPT_BITS**)

  number of entries in a Page Directory Pointer Table (PDPT)

Functions

- void **vm_pae_boot_init** (void)

  *This header file contains declarations for the PAE functions defined in **hal/vm_pae.c** (p. 342).*
- **pte_t** ∗ **vm_pae_lookup_page_directory** (**addr_space_t** ∗addr_space, void ∗addr, **bool** create_as_needed)

  Lookup and map the page directory for a specified address and address space.

- unsigned int **vm_pae_page_table_offset_of** (**addr_t** addr)
- unsigned int **vm_pae_page_directory_offset_of** (**addr_t** addr)
- **pte_t** $*$ **vm_pae_get_pte_with_offset** (**pte_t** $*$pte, unsigned int offset)
- void **vm_pae_set_pte** (**pte_t** $*$pte, **uint64_t** paddr, int flags)

  TODO handle flag bit position $>$ 31 for NX bit support.
- void **vm_pae_set_pte_flags** (**pte_t** $*$pte, int flags)

  TODO handle flag bit position $>$ 31 for NX bit support.
- int **vm_pae_get_pte_flags** (const **pte_t** $*$pte)
- **uint64_t vm_pae_get_pte_paddr** (const **pte_t** $*$pte)

  TODO mask NX bit as well, maximum 52 bits supported.
- void **vm_pae_clear_pte** (**pte_t** $*$pte)
- void **vm_pae_copy_pte** (**pte_t** $*$dest, const **pte_t** $*$src)
- void **vm_pae_create_pdpt_cache** (**boot_alloc_t** $*$boot_alloc)
- **addr_space_t** $*$ **vm_pae_create_addr_space** (**addr_space_t** $*$addr_space)
- **addr_space_t** $*$ **vm_pae_create_initial_addr_space** (**boot_alloc_t** $*$boot_alloc)
- void **vm_pae_destroy_addr_space** (**addr_space_t** $*$addr_space)
- void **vm_pae_unmap_low_alias** (**addr_space_t** $*$addr_space)

Variables

- **pdpt_t** $*$ **initial_pdpt**

4.98.1 Macro Definition Documentation

4.98.1.1 #define PDPT_BITS 2

number of address bits that encode the PDPT offset

Definition at line 46 of file vm_pae.c.

4.98.1.2 #define PDPT_ENTRIES (1 $<<$ PDPT_BITS)

number of entries in a Page Directory Pointer Table (PDPT)

Definition at line 49 of file vm_pae.c.

Referenced by vm_pae_create_addr_space(), vm_pae_create_initial_addr_space(), and vm_pae_destroy_addr_-space().

4.98.2 Function Documentation

4.98.2.1 void vm_pae_boot_init (void)

This header file contains declarations for the PAE functions defined in **hal/vm_pae.c** (p. 342).

It is intended to be included by **hal/vm.c** (p. 331) and **hal/vm_pae.c** (p. 342). There should be no reason to include it anywhere else.

Definition at line 72 of file vm_pae.c.

References PAGE_TABLE_ENTRIES, and page_table_entries.

Referenced by vm_boot_init().

```
72                          {
73     page_table_entries = (size_t)PAGE_TABLE_ENTRIES;
74 }
```

4.98.2.2 void vm_pae_clear_pte (pte_t * *pte*)

Definition at line 150 of file vm_pae.c.

References pte_t::entry.

Referenced by vm_pae_create_addr_space(), vm_pae_create_initial_addr_space(), and vm_pae_unmap_low_alias().

```
150                              {
151     pte->entry = 0;
152 }
```

4.98.2.3 void vm_pae_copy_pte (pte_t * *dest,* const pte_t * *src*)

Definition at line 154 of file vm_pae.c.

References pte_t::entry.

Referenced by vm_pae_create_addr_space().

```
154                                                   {
155     dest->entry = src->entry;
156 }
```

4.98.2.4 addr_space_t* vm_pae_create_addr_space (addr_space_t * *addr_space*)

Definition at line 170 of file vm_pae.c.

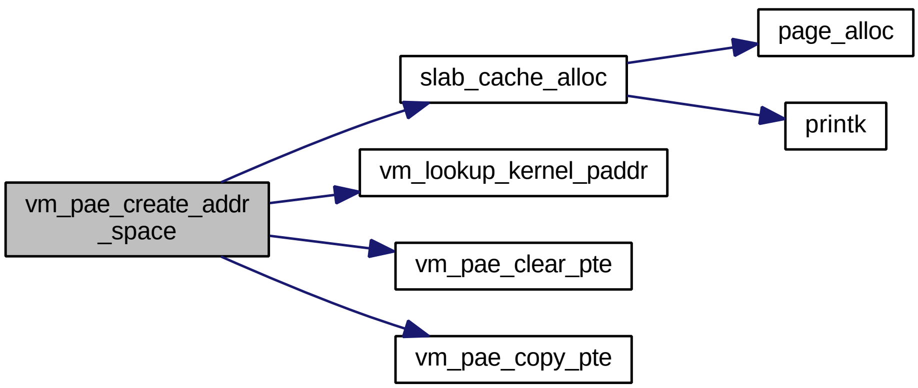
References addr_space_t::cr3, initial_addr_space, KLIMIT, NULL, page_address_of, page_offset_of, pdpt_t::pd, addr-_space_t::pdpt, PDPT_ENTRIES, slab_cache_alloc(), addr_space_t::top_level, vm_lookup_kernel_paddr(), vm_pae_-clear_pte(), and vm_pae_copy_pte().

Referenced by vm_create_addr_space().

```
170                                                                   {
171     unsigned int idx;
172     pte_t *pdpte;
173 
174     /* Create a PDPT for the new address space */
175     pdpt_t *pdpt = slab_cache_alloc(&pdpt_cache);
176 
177     if(pdpt == NULL) {
178         return NULL;
179     }
180 
181     /* Use the initial address space as a template for the kernel address range
182      * (address KLIMIT and above). The page tables for that range are shared by
183      * all address spaces. */
184     pdpt_t *template_pdpt = initial_addr_space.top_level.pdpt;
185 
186     for(idx = 0; idx < PDPT_ENTRIES; ++idx) {
187         pdpte = &pdpt->pd[idx];
188 
189         if(idx < pdpt_offset_of((addr_t)KLIMIT)) {
190             /* This PDPT entry describes an address range entirely under KLIMIT
191              * so it is all user space: do not create a page directory at this
192              * time. */
193             vm_pae_clear_pte(pdpte);
194         }
```

```
        else {
            /* This page directory describes an address range entirely above
             * KLIMIT: share the template's page directory. */
            vm_pae_copy_pte(pdpte, &template_pdpt->pd[idx]);
        }
    }

    /* Lookup the physical address of the page where the PDPT resides. */
    kern_paddr_t pdpt_page_paddr = vm_lookup_kernel_paddr((addr_t)
  page_address_of(pdpt));

    /* physical address of PDPT */
    kern_paddr_t pdpt_paddr = pdpt_page_paddr | page_offset_of(pdpt);

    addr_space->top_level.pdpt  = pdpt;
    addr_space->cr3             = pdpt_paddr;

    return addr_space;
}
```

Here is the call graph for this function:



4.98.2.5 addr_space_t∗ vm_pae_create_initial_addr_space (boot_alloc_t ∗ *boot_alloc*)

Definition at line 258 of file vm_pae.c.

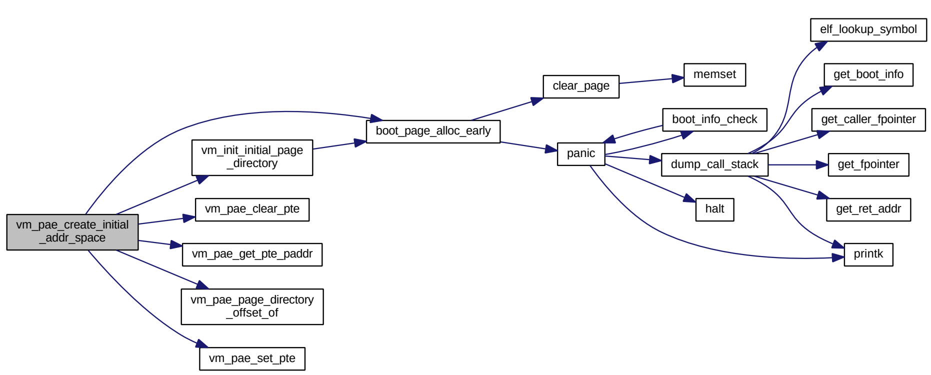
References boot_heap_alloc, boot_page_alloc_early(), addr_space_t::cr3, EARLY_PHYS_TO_VIRT, EARLY_PTR_TO_PHYS_ADDR, EARLY_VIRT_TO_PHYS, initial_addr_space, initial_pdpt, KERNEL_PREALLOC_LIMIT, KLIMIT, page_table_entries, pdpt_t::pd, addr_space_t::pdpt, PDPT_ENTRIES, addr_space_t::top_level, VM_FLAG_PRESENT, vm_init_initial_page_directory(), vm_pae_clear_pte(), vm_pae_get_pte_paddr(), vm_pae_page_directory_offset_of(), and vm_pae_set_pte().

Referenced by vm_create_initial_addr_space().

```
                                                                          {

    unsigned int idx;

    /* Allocate initial PDPT. PDPT must be 32-byte aligned. */
    initial_pdpt = boot_heap_alloc(boot_alloc, pdpt_t, 32);

    /* We want the pre-allocated kernel page tables to be contiguous. For this
     * reason, we allocate the page directories first, and then the page tables.
     *
     * This function allocates pages in this order:
     *      +---------------+-------...--------+-------...------+
     *      |    Low alias   |  pre-allocated   |  pre-allocated |
     *      | page directory |      kernel      |     kernel     |
     *      | and page table | page directories |  page tables   |
     *      +---------------+-------...--------+-------...------+
```

```
     * */

    for(idx = 0; idx < PDPT_ENTRIES; ++idx) {
        vm_pae_clear_pte(&initial_pdpt->pd[idx]);
    }

    vm_pae_init_low_alias(initial_pdpt, boot_alloc);

    const unsigned int last_idx = pdpt_offset_of((addr_t)KERNEL_PREALLOC_LIMIT - 1);

    for(idx = pdpt_offset_of((addr_t)KLIMIT); idx <= last_idx; ++idx) {
        pte_t *const pdpte       = &initial_pdpt->pd[idx];
        pte_t *page_directory    = (pte_t *)boot_page_alloc_early(boot_alloc);

        vm_pae_set_pte(
                pdpte,
                EARLY_PTR_TO_PHYS_ADDR(page_directory),
                VM_FLAG_PRESENT);
    }

    for(idx = pdpt_offset_of((addr_t)KLIMIT); idx <= last_idx; ++idx) {
        unsigned int end_index;

        pte_t *const pdpte = &initial_pdpt->pd[idx];
        pte_t *const page_directory = (pte_t *)EARLY_PHYS_TO_VIRT(
vm_pae_get_pte_paddr(pdpte));

        if(idx < pdpt_offset_of((addr_t)KERNEL_PREALLOC_LIMIT)) {
            end_index = page_table_entries;
        }
        else {
            end_index = vm_pae_page_directory_offset_of((addr_t)KERNEL_PREALLOC_LIMIT);
        }

        vm_init_initial_page_directory(
                page_directory,
                boot_alloc,
                0,
                end_index,
                idx == pdpt_offset_of((addr_t)KLIMIT));
    }

    initial_addr_space.top_level.pdpt   = initial_pdpt;
    initial_addr_space.cr3              = EARLY_VIRT_TO_PHYS(initial_pdpt);

    return &initial_addr_space;
}
```

Here is the call graph for this function:



4.98.2.6 void vm_pae_create_pdpt_cache (**boot_alloc_t** ∗ *boot_alloc*)

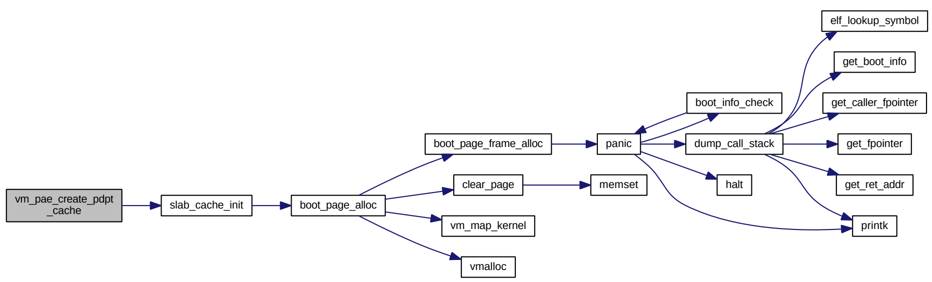
Definition at line 158 of file vm_pae.c.

References NULL, slab_cache_init(), and SLAB_DEFAULTS.

Referenced by vm_boot_postinit().

```
                                                                {
    slab_cache_init(
            &pdpt_cache,
            "vm_pae_pdpt_cache",
            sizeof(pdpt_t),
            sizeof(pdpt_t),
            NULL,
            NULL,
            SLAB_DEFAULTS,
            boot_alloc);
}
```

Here is the call graph for this function:



4.98.2.7 void vm_pae_destroy_addr_space (**addr_space_t** ∗ *addr_space*)

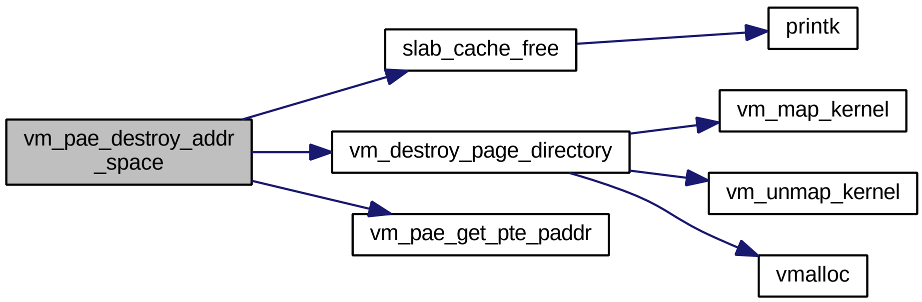
Definition at line 321 of file vm_pae.c.

References pte_t::entry, KLIMIT, pdpt_t::pd, addr_space_t::pdpt, PDPT_ENTRIES, slab_cache_free(), addr_space_t::top_level, vm_destroy_page_directory(), VM_FLAG_PRESENT, and vm_pae_get_pte_paddr().

Referenced by vm_destroy_addr_space().

```
321                                                                 {
322     unsigned int idx;
323     pte_t pdpte;
324 
325     pdpt_t *pdpt = addr_space->top_level.pdpt;
326 
327     for(idx = 0; idx < PDPT_ENTRIES; ++idx) {
328         pdpte.entry = pdpt->pd[idx].entry;
329 
330         if(idx < pdpt_offset_of((addr_t)KLIMIT)) {
331             /* This page directory describes an address range entirely under
332              * KLIMIT so it is all user space: free all page tables in this
333              * page directory as well as the page directory itself. */
334             if(pdpte.entry & VM_FLAG_PRESENT) {
335                 vm_destroy_page_directory(
336                         vm_pae_get_pte_paddr(&pdpte),
337                         0,
338                         page_table_entries);
339             }
340         }
341         else {
342             /* This page directory describes an address range entirely above
343              * KLIMIT: do nothing.
344              *
345              * The page directory must not be freed because it is shared by all
346              * address spaces. */
347         }
348     }
349 
350     slab_cache_free(pdpt);
351 }
```

Here is the call graph for this function:



4.98.2.8 int vm_pae_get_pte_flags (const pte_t * *pte*)

Definition at line 141 of file vm_pae.c.

References pte_t::entry, and PAGE_MASK.

Referenced by vm_pae_lookup_page_directory().

```
141                                        {
142     return pte->entry & PAGE_MASK;
143 }
```

4.98.2.9 uint64_t vm_pae_get_pte_paddr (const pte_t * *pte*)

TODO mask NX bit as well, maximum 52 bits supported.

Definition at line 146 of file vm_pae.c.

References pte_t::entry, and PAGE_MASK.

Referenced by vm_pae_create_initial_addr_space(), vm_pae_destroy_addr_space(), and vm_pae_lookup_page_directory().

```
146                                             {
147     return (pte->entry & ~(uint64_t)PAGE_MASK);
148 }
```

4.98.2.10 pte_t* vm_pae_get_pte_with_offset (pte_t * *pte,* unsigned int *offset*)

Definition at line 127 of file vm_pae.c.

```
127                                                                   {
128     return &pte[offset];
129 }
```

4.98.2.11 pte_t* vm_pae_lookup_page_directory (addr_space_t * *addr_space,* void * *addr,* bool *create_as_needed*)

Lookup and map the page directory for a specified address and address space.

Important note: it is the caller's responsibility to unmap and free the returned page directory when it is done with it.

Parameters

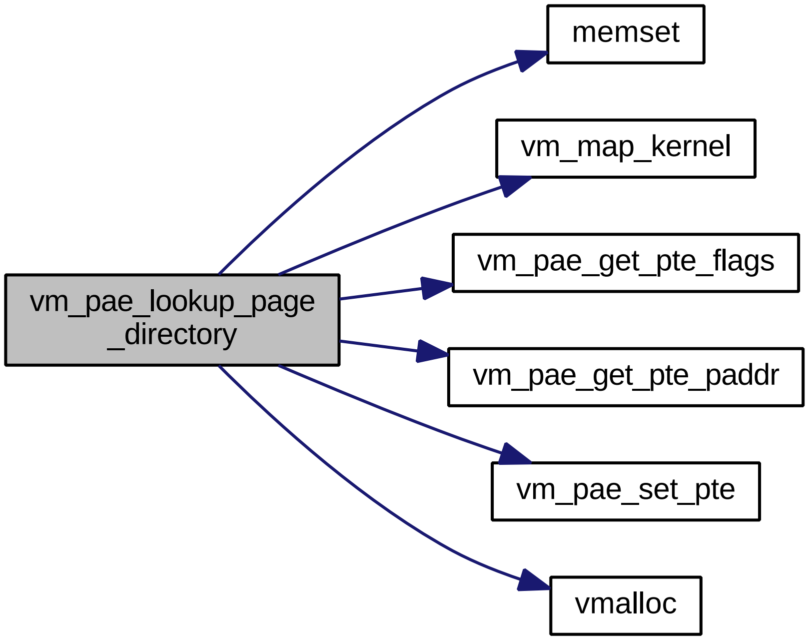
addr_space	address space in which the address is looked up.
addr	address to look up
create_as_need	Whether a page table is allocated if it does not exist

Definition at line 86 of file vm_pae.c.

References memset(), NULL, PAGE_SIZE, pdpt_t::pd, addr_space_t::pdpt, pfalloc, addr_space_t::top_level, VM_FLAG_PRESENT, VM_FLAG_READ_WRITE, vm_map_kernel(), vm_pae_get_pte_flags(), vm_pae_get_pte_paddr(), vm_pae_set_pte(), and vmalloc().

```
86                                                                                                                    {
87      pdpt_t *pdpt    = addr_space->top_level.pdpt;
88      pte_t  *pdpte   = &pdpt->pd[pdpt_offset_of(addr)];
89
90      if(vm_pae_get_pte_flags(pdpte) & VM_FLAG_PRESENT) {
91          /* map page directory */
92          pte_t *page_directory   = (pte_t *)vmalloc();
93          vm_map_kernel((addr_t)page_directory, vm_pae_get_pte_paddr(pdpte),
     VM_FLAG_READ_WRITE);
94
95          return page_directory;
96      }
97      else {
98          if(create_as_needed) {
99              /* allocate a new page directory and map it */
100             pte_t *page_directory       = (pte_t *)vmalloc();
101             kern_paddr_t pgdir_paddr    = pfalloc();
102
103             vm_map_kernel((addr_t)page_directory, pgdir_paddr,
     VM_FLAG_READ_WRITE);
104
105             /* zero content of page directory */
106             memset(page_directory, 0, PAGE_SIZE);
107
108             /* link page directory in PDPT */
109             vm_pae_set_pte(pdpte, pgdir_paddr, VM_FLAG_PRESENT);
110
111             return page_directory;
112         }
113         else {
114             return NULL;
115         }
116     }
117 }
```

Here is the call graph for this function:



4.98.2.12 unsigned int vm_pae_page_directory_offset_of (addr_t *addr*)

Definition at line 123 of file vm_pae.c.

References PAGE_DIRECTORY_OFFSET_OF.

Referenced by vm_pae_create_initial_addr_space().

```
                                                    {
    return PAGE_DIRECTORY_OFFSET_OF(addr);
}
```

4.98.2.13 unsigned int vm_pae_page_table_offset_of (addr_t *addr*)

Definition at line 119 of file vm_pae.c.

References PAGE_TABLE_OFFSET_OF.

```
                                                {
    return PAGE_TABLE_OFFSET_OF(addr);
}
```

4.98.2.14 void vm_pae_set_pte (pte_t $*$ *pte,* uint64_t *paddr,* int *flags*)

TODO handle flag bit position $>$ 31 for NX bit support.

Definition at line 132 of file vm_pae.c.

References pte_t::entry.

Referenced by vm_pae_create_initial_addr_space(), and vm_pae_lookup_page_directory().

```
132                                                         {
133     pte->entry = paddr | flags;
134 }
```

4.98.2.15 void vm_pae_set_pte_flags (pte_t * *pte,* int *flags*)

TODO handle flag bit position > 31 for NX bit support.

Definition at line 137 of file vm_pae.c.

References pte_t::entry, and PAGE_MASK.

```
137                                               {
138     pte->entry = (pte->entry & ~(uint64_t)PAGE_MASK) | flags;
139 }
```

4.98.2.16 void vm_pae_unmap_low_alias (addr_space_t * *addr_space*)

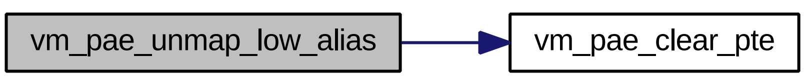
Definition at line 353 of file vm_pae.c.

References pdpt_t::pd, addr_space_t::pdpt, addr_space_t::top_level, and vm_pae_clear_pte().

Referenced by vm_boot_init().

```
353                                                    {
354     /* Enabling PAE requires disabling paging temporarily, which in turn requires
355      * an alias of the kernel image region at address 0 to match its physical
356      * address. This function gets rid of this alias once PAE is enabled.
357      *
358      * There is no need for TLB invalidation because the caller reloads CR3 just
359      * after calling this function. */
360     vm_pae_clear_pte(&addr_space->top_level.pdpt->pd[0]);
361 }
```

Here is the call graph for this function:



4.98.3 Variable Documentation

4.98.3.1 pdpt_t* initial_pdpt

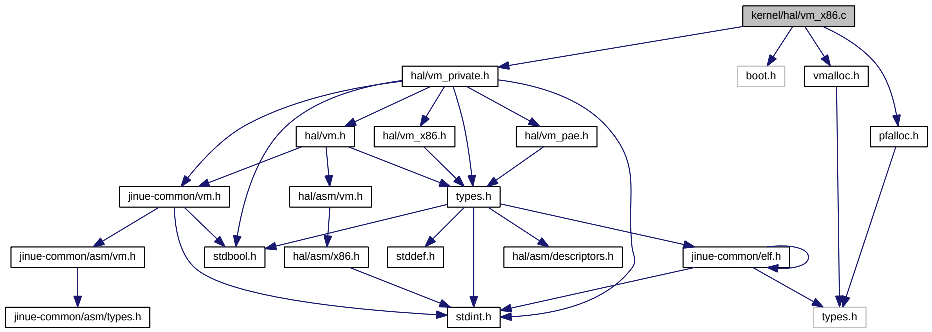
Definition at line 63 of file vm_pae.c.

Referenced by vm_pae_create_initial_addr_space().

4.99 kernel/hal/vm_x86.c File Reference

```
#include <hal/vm_private.h>
#include <boot.h>
#include <pfalloc.h>
#include <vmalloc.h>
```

Include dependency graph for vm_x86.c:



Data Structures

- struct **pte_t**

Functions

- void **vm_x86_boot_init** (void)

    This header file contains declarations for the non-PAE functions defined in ***hal/vm_x86.c*** *(p. 351).*
- **addr_space_t** ∗ **vm_x86_create_addr_space** (**addr_space_t** ∗addr_space)
- **addr_space_t** ∗ **vm_x86_create_initial_addr_space** (**boot_alloc_t** ∗boot_alloc)
- void **vm_x86_destroy_addr_space** (**addr_space_t** ∗addr_space)
- unsigned int **vm_x86_page_table_offset_of** (**addr_t** addr)
- unsigned int **vm_x86_page_directory_offset_of** (**addr_t** addr)
- **pte_t** ∗ **vm_x86_lookup_page_directory** (**addr_space_t** ∗addr_space)

    Lookup and map the page directory for a specified address and address space.
- **pte_t** ∗ **vm_x86_get_pte_with_offset** (**pte_t** ∗pte, unsigned int offset)
- void **vm_x86_set_pte** (**pte_t** ∗pte, **uint32_t** paddr, int flags)
- void **vm_x86_set_pte_flags** (**pte_t** ∗pte, int flags)
- int **vm_x86_get_pte_flags** (const **pte_t** ∗pte)
- **uint32_t vm_x86_get_pte_paddr** (const **pte_t** ∗pte)
- void **vm_x86_clear_pte** (**pte_t** ∗pte)
- void **vm_x86_copy_pte** (**pte_t** ∗dest, const **pte_t** ∗src)

4.99.1 Function Documentation

4.99.1.1 void vm_x86_boot_init (void)

This header file contains declarations for the non-PAE functions defined in **hal/vm_x86.c** (p. 351).

It is intended to be included by **hal/vm.c** (p. 331) and **hal/vm_x86.c** (p. 351). There should be no reason to include it anywhere else.

Definition at line 41 of file vm_x86.c.

References PAGE_TABLE_ENTRIES, and page_table_entries.

Referenced by vm_boot_init().

```
41                          {
42     page_table_entries = (size_t)PAGE_TABLE_ENTRIES;
43 }
```

4.99.1.2 void vm_x86_clear_pte (pte_t * *pte*)

Definition at line 133 of file vm_x86.c.

References pte_t::entry.

```
133                                  {
134     pte->entry = 0;
135 }
```

4.99.1.3 void vm_x86_copy_pte (pte_t * *dest,* const pte_t * *src*)

Definition at line 137 of file vm_x86.c.

References pte_t::entry.

```
137                                                     {
138     dest->entry = src->entry;
139 }
```

4.99.1.4 addr_space_t* vm_x86_create_addr_space (addr_space_t * *addr_space*)

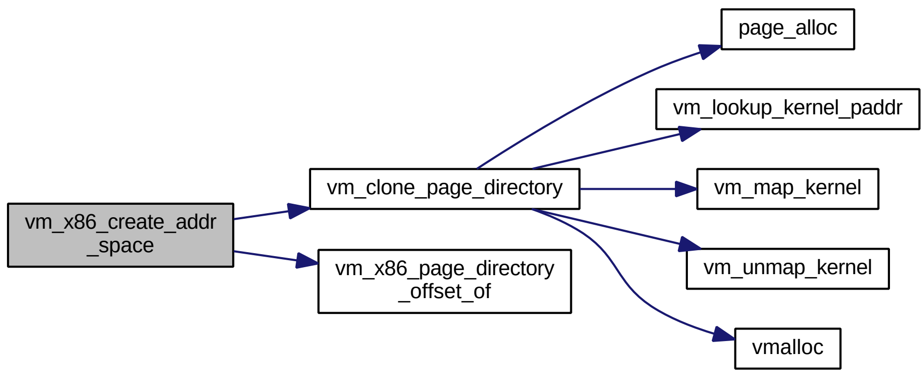
Definition at line 45 of file vm_x86.c.

References addr_space_t::cr3, initial_addr_space, KLIMIT, addr_space_t::pd, addr_space_t::top_level, vm_clone_-page_directory(), and vm_x86_page_directory_offset_of().

Referenced by vm_create_addr_space().

```
45                                                                  {
46     /* Create a new page directory where entries for the address range starting
47      * at KLIMIT are copied from the initial address space. The mappings starting
48      * at KLIMIT belong to the kernel and are identical in all address spaces. */
49     kern_paddr_t paddr = vm_clone_page_directory(
50             initial_addr_space.top_level.pd,
51             vm_x86_page_directory_offset_of((addr_t)KLIMIT));
52 
53     addr_space->top_level.pd = paddr;
54     addr_space->cr3          = paddr;
55 
56     return addr_space;
57 }
```

Here is the call graph for this function:



4.99.1.5 addr_space_t∗ vm_x86_create_initial_addr_space (boot_alloc_t ∗ *boot_alloc*)

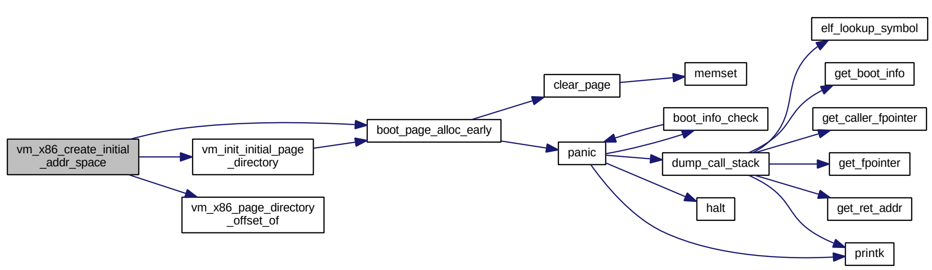
Definition at line 59 of file vm_x86.c.

References boot_page_alloc_early(), addr_space_t::cr3, EARLY_PTR_TO_PHYS_ADDR, EARLY_VIRT_TO_PHYS, initial_addr_space, KERNEL_PREALLOC_LIMIT, KLIMIT, addr_space_t::pd, addr_space_t::top_level, vm_init_initial_-page_directory(), and vm_x86_page_directory_offset_of().

Referenced by vm_create_initial_addr_space().

```
59                                                                      {
60     pte_t *page_directory = (pte_t *)boot_page_alloc_early(boot_alloc);
61 
62     vm_init_initial_page_directory(
63             page_directory,
64             boot_alloc,
65             vm_x86_page_directory_offset_of((addr_t)KLIMIT),
66             vm_x86_page_directory_offset_of((addr_t)KERNEL_PREALLOC_LIMIT),
67             true);
68 
69     initial_addr_space.top_level.pd = EARLY_PTR_TO_PHYS_ADDR(page_directory);
70     initial_addr_space.cr3          = EARLY_VIRT_TO_PHYS((uintptr_t)page_directory);
71 
72     return &initial_addr_space;
73 }
```

Here is the call graph for this function:



4.99.1.6 void vm_x86_destroy_addr_space (addr_space_t ∗ *addr_space*)

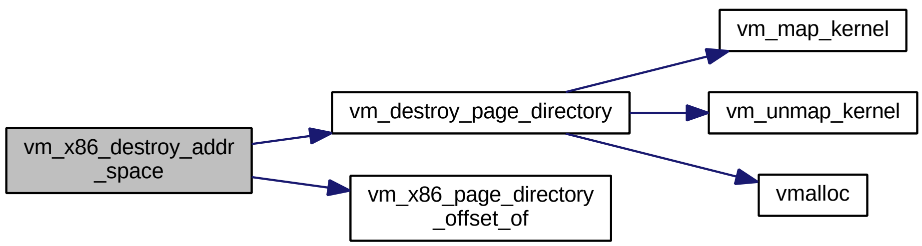
Definition at line 75 of file vm_x86.c.

References KLIMIT, addr_space_t::pd, addr_space_t::top_level, vm_destroy_page_directory(), and vm_x86_page_-directory_offset_of().

Referenced by vm_destroy_addr_space().

```
75                                                         {
76     vm_destroy_page_directory(
77             addr_space->top_level.pd,
78             /* Free page tables for addresses 0..KLIMIT, be careful not to free
79              * the kernel page tables starting at KLIMIT. */
80             0,
81             vm_x86_page_directory_offset_of((addr_t)KLIMIT));
82 }
```

Here is the call graph for this function:



4.99.1.7 int vm_x86_get_pte_flags (const pte_t ∗ *pte*)

Definition at line 125 of file vm_x86.c.

References pte_t::entry, and PAGE_MASK.

```
125                                         {
126     return pte->entry & PAGE_MASK;
127 }
```

4.99.1.8 uint32_t vm_x86_get_pte_paddr (const pte_t ∗ *pte*)

Definition at line 129 of file vm_x86.c.

References pte_t::entry, and PAGE_MASK.

```
129                                              {
130     return pte->entry & ~PAGE_MASK;
131 }
```

4.99.1.9 pte_t∗ vm_x86_get_pte_with_offset (pte_t ∗ *pte,* unsigned int *offset*)

Definition at line 113 of file vm_x86.c.

```
113                                                                    {
114     return &pte[offset];
115 }
```

4.99.1.10 pte_t∗ vm_x86_lookup_page_directory (addr_space_t ∗ *addr_space*)

Lookup and map the page directory for a specified address and address space.

This is the implementation for standard 32-bit (i.e. non-PAE) paging. This means that there is only one preallocated page directory, so the addr and create_as_needed arguments are both irrelevant.

Important note: it is the caller's responsibility to unmap and free the returned page directory when it is done with it.

Parameters

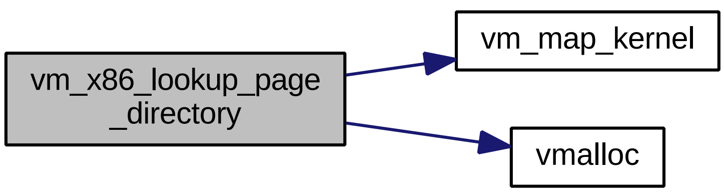
addr_space	address space in which the address is looked up.
addr	address to look up
create_as_need	Whether a page table is allocated if it does not exist

Definition at line 106 of file vm_x86.c.

References addr_space_t::pd, addr_space_t::top_level, VM_FLAG_READ_WRITE, vm_map_kernel(), and vmalloc().

```
106                                                               {
107     pte_t *page_directory = (pte_t *)vmalloc();
108     vm_map_kernel((addr_t)page_directory, addr_space->top_level.pd,
      VM_FLAG_READ_WRITE);
109
110     return page_directory;
111 }
```

Here is the call graph for this function:



4.99.1.11 unsigned int vm_x86_page_directory_offset_of (addr_t *addr*)

Definition at line 88 of file vm_x86.c.

References PAGE_DIRECTORY_OFFSET_OF.

Referenced by vm_x86_create_addr_space(), vm_x86_create_initial_addr_space(), and vm_x86_destroy_addr_-space().

```
88                                                           {
89     return PAGE_DIRECTORY_OFFSET_OF(addr);
90 }
```

4.99.1.12 unsigned int vm_x86_page_table_offset_of (addr_t *addr*)

Definition at line 84 of file vm_x86.c.

References PAGE_TABLE_OFFSET_OF.

```
84                                                       {
85     return PAGE_TABLE_OFFSET_OF(addr);
86 }
```

4.99.1.13 void vm_x86_set_pte (pte_t ∗ *pte,* uint32_t *paddr,* int *flags*)

Definition at line 117 of file vm_x86.c.

References pte_t::entry.

```
117                                                              {
118     pte->entry = paddr | flags;
119 }
```

4.99.1.14 void vm_x86_set_pte_flags (pte_t ∗ *pte,* int *flags*)

Definition at line 121 of file vm_x86.c.

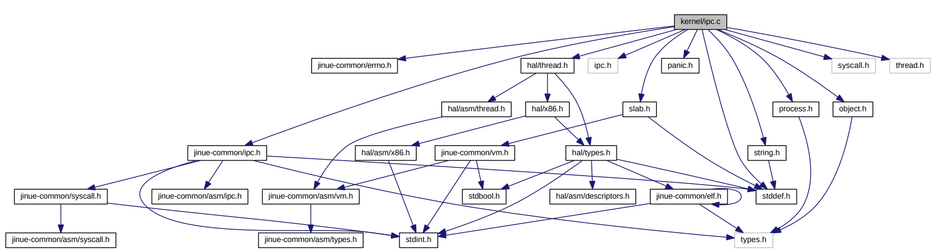
References pte_t::entry, and PAGE_MASK.

```
121                                                    {
122     pte->entry = (pte->entry & ~PAGE_MASK) | flags;
123 }
```

4.100 kernel/ipc.c File Reference

```
#include <jinue-common/errno.h>
#include <jinue-common/ipc.h>
#include <hal/thread.h>
#include <ipc.h>
#include <object.h>
#include <panic.h>
#include <process.h>
#include <slab.h>
#include <stddef.h>
#include <string.h>
#include <syscall.h>
#include <thread.h>
```

Include dependency graph for ipc.c:



Functions

- void **ipc_boot_init** (**boot_alloc_t** ∗boot_alloc)
- **ipc_t** ∗ **ipc_object_create** (int flags)
- **ipc_t** ∗ **ipc_get_proc_object** (void)
- void **ipc_send** (**jinue_syscall_args_t** ∗args)
- void **ipc_receive** (**jinue_syscall_args_t** ∗args)
- void **ipc_reply** (**jinue_syscall_args_t** ∗args)

4.100.1 Function Documentation

4.100.1.1 void ipc_boot_init (**boot_alloc_t** ∗ *boot_alloc*)

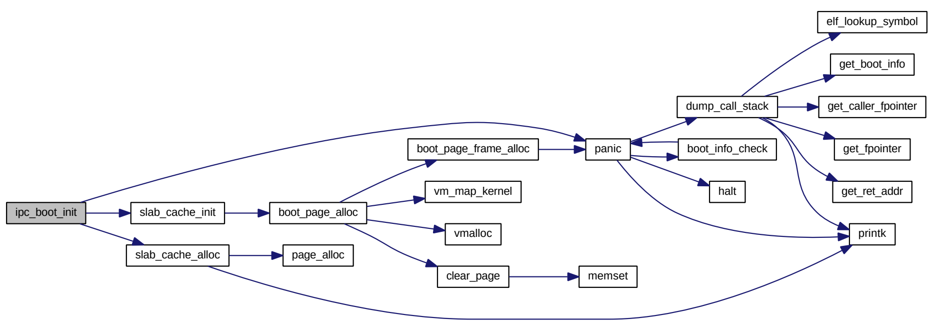
Definition at line 58 of file ipc.c.

References NULL, panic(), slab_cache_alloc(), slab_cache_init(), and SLAB_DEFAULTS.

Referenced by kmain().

```
58                                        {
59     slab_cache_init(
60             &ipc_object_cache,
61             "ipc_object_cache",
62             sizeof(ipc_t),
63             0,
64             ipc_object_ctor,
65             NULL,
66             SLAB_DEFAULTS,
67             boot_alloc);
68
69     proc_ipc = slab_cache_alloc(&ipc_object_cache);
70
71     if(proc_ipc == NULL) {
72         panic("Cannot create process manager IPC object.");
73     }
74 }
```

Here is the call graph for this function:



4.100.1.2 ipc_t∗ ipc_get_proc_object (void)

Definition at line 86 of file ipc.c.

Referenced by dispatch_syscall().

```
86                              {
87     return proc_ipc;
88 }
```

4.100.1.3 ipc_t∗ ipc_object_create (int *flags*)

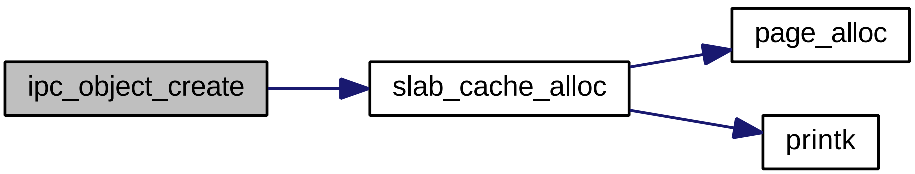
Definition at line 76 of file ipc.c.

References object_header_t::flags, ipc_t::header, NULL, and slab_cache_alloc().

Referenced by dispatch_syscall().

```
{
    ipc_t *ipc = slab_cache_alloc(&ipc_object_cache);

    if(ipc != NULL) {
        ipc->header.flags = flags;
    }

    return ipc;
}
```

Here is the call graph for this function:



4.100.1.4 void ipc_receive (jinue_syscall_args_t * *args*)

Definition at line 205 of file ipc.c.

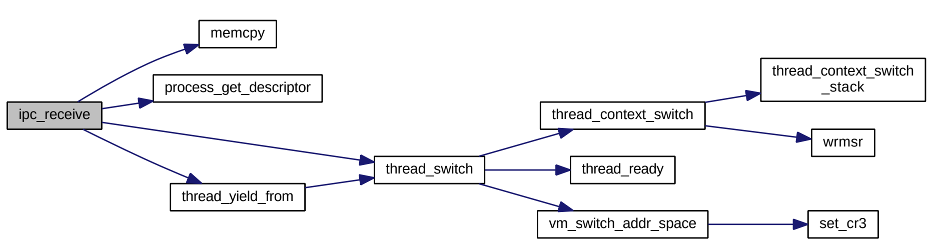
References jinue_syscall_args_t::arg0, jinue_syscall_args_t::arg1, jinue_syscall_args_t::arg2, jinue_syscall_args_t::arg3, message_info_t::data_size, object_header_t::flags, thread_t::header, JINUE_E2BIG, JINUE_EBADF, JINUE_EINVAL, JINUE_EIO, JINUE_EPERM, jinue_node_entry, memcpy(), thread_t::message_args, thread_t::message_buffer, thread_t::message_info, NULL, OBJECT_REF_FLAG_CLOSED, OBJECT_TYPE_IPC, thread_t::process, process_get_descriptor(), ipc_t::recv_list, ipc_t::send_list, thread_t::sender, thread_t::thread_list, thread_switch(), thread_yield_from(), message_info_t::total_size, and object_header_t::type.

Referenced by dispatch_syscall().

```
205                                 {
206     thread_t *thread = get_current_thread();
207 
208     int fd = (int)args->arg1;
209 
210     object_ref_t *ref = process_get_descriptor(thread->process, fd);
211 
212     if(! object_ref_is_valid(ref)) {
213         syscall_args_set_error(args, JINUE_EBADF);
214         return;
215     }
216 
217     if(object_ref_is_closed(ref)) {
218         syscall_args_set_error(args, JINUE_EIO);
219         return;
220     }
221 
222     if(! object_ref_is_owner(ref)) {
223         syscall_args_set_error(args, JINUE_EPERM);
224         return;
225     }
226 
227     object_header_t *header = ref->object;
228 
229     if(object_is_destroyed(header)) {
230         ref->flags |= OBJECT_REF_FLAG_CLOSED;
231         object_subref(header);
232 
233         syscall_args_set_error(args, JINUE_EIO);
234         return;
235     }
236 
237     if(header->type != OBJECT_TYPE_IPC) {
238         syscall_args_set_error(args, JINUE_EBADF);
```

```
        return;
    }

    ipc_t *ipc = (ipc_t *)header;

    char *user_ptr = (char *)args->arg2;
    size_t buffer_size = jinue_args_get_buffer_size(args);

    if(! user_buffer_check(user_ptr, buffer_size)) {
        syscall_args_set_error(args, JINUE_EINVAL);
        return;
    }

    thread_t *send_thread = jinue_node_entry(
        jinue_list_dequeue(&ipc->send_list),
        thread_t,
        thread_list);

    if(send_thread == NULL) {
        /* No thread is waiting to send a message, so we must wait on the receive
         * list. */
        jinue_list_enqueue(&ipc->recv_list, &thread->thread_list);

        thread_yield_from(
                thread,
                true,       /* make thread block */
                false);     /* don't destroy */

        /* set by sending thread */
        send_thread = thread->sender;
    }
    else {
        object_addref(&send_thread->header);
        thread->sender = send_thread;
    }

    if(send_thread->message_info.total_size > buffer_size) {
        /* message is too big for receive buffer */
        object_subref(&send_thread->header);
        thread->sender = NULL;

        syscall_args_set_error(send_thread->message_args, JINUE_E2BIG);
        syscall_args_set_error(args, JINUE_E2BIG);

        /* switch back to sender thread to return from call immediately */
        thread_switch(
                thread,
                send_thread,
                false,      /* don't block (put this thread back in ready queue) */
                false);     /* don't destroy */

        return;
    }

    memcpy(
        user_ptr,
        send_thread->message_buffer,
        send_thread->message_info.data_size);

    args->arg0 = send_thread->message_args->arg0;
    args->arg1 = ref->cookie;
    /* argument 2 is left intact (buffer pointer) */
    args->arg3 = send_thread->message_args->arg3;
}
```

Here is the call graph for this function:



4.100.1.5 void ipc_reply (jinue_syscall_args_t * args)

TODO is there a better error number for this situation?

TODO remove this check when descriptor passing is implemented

TODO copy descriptors

TODO set return value and error number

Definition at line 304 of file ipc.c.

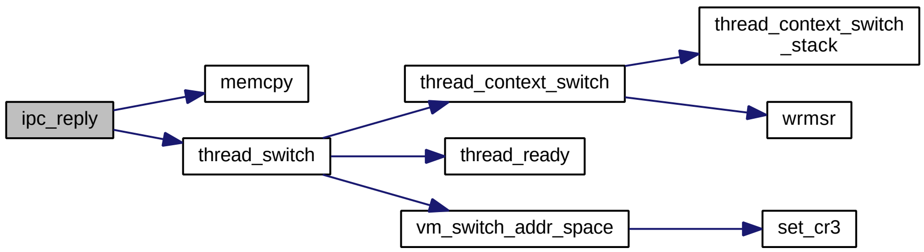
References jinue_syscall_args_t::arg2, jinue_syscall_args_t::arg3, message_info_t::buffer_size, message_info_t::data_size, message_info_t::desc_n, thread_t::header, JINUE_EINVAL, JINUE_ENOSYS, JINUE_SEND_BUFFER_SIZE_OFFSET, JINUE_SEND_MAX_N_DESC, JINUE_SEND_MAX_SIZE, JINUE_SEND_SIZE_MASK, memcpy(), thread_t::message_args, thread_t::message_buffer, thread_t::message_info, NULL, thread_t::sender, and thread_switch().

Referenced by dispatch_syscall().

```
304                                             {
305     thread_t *thread        = get_current_thread();
306     thread_t *send_thread   = thread->sender;
307
308     if(send_thread == NULL) {
310         syscall_args_set_error(args, JINUE_EINVAL);
311         return;
312     }
313
314     size_t buffer_size   = jinue_args_get_buffer_size(args);
315     size_t data_size     = jinue_args_get_data_size(args);
316     size_t desc_n        = jinue_args_get_n_desc(args);
317     size_t total_size    =
318             data_size +
319             desc_n * sizeof(jinue_ipc_descriptor_t);
320
321     if(buffer_size > JINUE_SEND_MAX_SIZE) {
322         syscall_args_set_error(args, JINUE_EINVAL);
323         return;
324     }
325
326     if(total_size > buffer_size) {
327         syscall_args_set_error(args, JINUE_EINVAL);
328         return;
329     }
330
331     if(desc_n > JINUE_SEND_MAX_N_DESC) {
332         syscall_args_set_error(args, JINUE_EINVAL);
333         return;
334     }
335
336     /* the reply must fit in the sender's buffer */
337     if(total_size > send_thread->message_info.buffer_size) {
338         syscall_args_set_error(args, JINUE_EINVAL);
339         return;
340     }
341
343     if(desc_n > 0) {
344         syscall_args_set_error(args, JINUE_ENOSYS);
345         return;
346     }
347
348     const char *user_ptr = (const char *)args->arg2;
349
350     if(! user_buffer_check(user_ptr, buffer_size)) {
351         syscall_args_set_error(args, JINUE_EINVAL);
352         return;
353     }
354
355     memcpy(&send_thread->message_buffer, user_ptr, data_size);
356
360     syscall_args_set_return(send_thread->message_args, 0);
361     send_thread->message_args->arg3 =
362             args->arg3 & ~(JINUE_SEND_SIZE_MASK << JINUE_SEND_BUFFER_SIZE_OFFSET);
363
364     send_thread->message_info.data_size = data_size;
365     send_thread->message_info.desc_n    = desc_n;
366
```

```
    object_subref(&send_thread->header);
    thread->sender = NULL;

    syscall_args_set_return(args, 0);

    /* switch back to sender thread to return from call immediately */
    thread_switch(
            thread,
            send_thread,
            false,      /* don't block (put this thread back in ready queue) */
            false);     /* don't destroy */
}
```

Here is the call graph for this function:



4.100.1.6 void ipc_send (jinue_syscall_args_t * *args*)

TODO remove this check when descriptor passing is implemented

TODO copy descriptors

TODO copy descriptors

Definition at line 90 of file ipc.c.

References jinue_syscall_args_t::arg0, jinue_syscall_args_t::arg1, jinue_syscall_args_t::arg2, message_info_t::buffer_size, message_info_t::cookie, message_info_t::data_size, message_info_t::desc_n, object_header_t::flags, message_info_t::function, thread_t::header, JINUE_EBADF, JINUE_EINVAL, JINUE_EIO, JINUE_ENOSYS, jinue_node_entry, JINUE_SEND_MAX_N_DESC, JINUE_SEND_MAX_SIZE, memcpy(), thread_t::message_args, thread_t::message_buffer, thread_t::message_info, NULL, OBJECT_REF_FLAG_CLOSED, OBJECT_TYPE_IPC, thread_t::process, process_get_descriptor(), ipc_t::recv_list, ipc_t::send_list, thread_t::sender, thread_t::thread_list, thread_switch(), thread_yield_from(), message_info_t::total_size, and object_header_t::type.

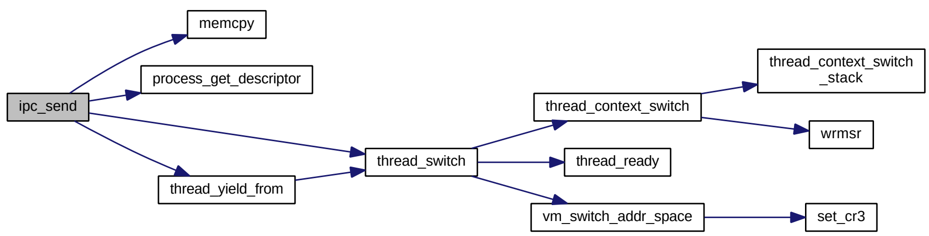
Referenced by dispatch_syscall().

```
90                                      {
91     thread_t *thread = get_current_thread();
92
93     message_info_t *message_info = &thread->message_info;
94
95     message_info->function      = args->arg0;
96     message_info->buffer_size   = jinue_args_get_buffer_size(args);
97     message_info->data_size     = jinue_args_get_data_size(args);
98     message_info->desc_n        = jinue_args_get_n_desc(args);
99     message_info->total_size    =
100             message_info->data_size +
101             message_info->desc_n * sizeof(jinue_ipc_descriptor_t);
102
103     if(message_info->buffer_size > JINUE_SEND_MAX_SIZE) {
104         syscall_args_set_error(args, JINUE_EINVAL);
105         return;
106     }
107
108     if(message_info->total_size > message_info->buffer_size) {
109         syscall_args_set_error(args, JINUE_EINVAL);
```

```
        return;
    }

    if(message_info->desc_n > JINUE_SEND_MAX_N_DESC) {
        syscall_args_set_error(args, JINUE_EINVAL);
        return;
    }

    if(message_info->desc_n > 0) {
        syscall_args_set_error(args, JINUE_ENOSYS);
        return;
    }

    int fd = (int)args->arg1;

    object_ref_t *ref = process_get_descriptor(thread->process, fd);

    if(! object_ref_is_valid(ref)) {
        syscall_args_set_error(args, JINUE_EBADF);
        return;
    }

    if(object_ref_is_closed(ref)) {
        syscall_args_set_error(args, JINUE_EIO);
        return;
    }

    message_info->cookie = ref->cookie;

    object_header_t *header = ref->object;

    if(object_is_destroyed(header)) {
        ref->flags |= OBJECT_REF_FLAG_CLOSED;
        object_subref(header);

        syscall_args_set_error(args, JINUE_EIO);
        return;
    }

    if(header->type != OBJECT_TYPE_IPC) {
        syscall_args_set_error(args, JINUE_EBADF);
        return;
    }

    ipc_t *ipc = (ipc_t *)header;

    char *user_ptr = (char *)args->arg2;

    if(! user_buffer_check(user_ptr, message_info->buffer_size)) {
        syscall_args_set_error(args, JINUE_EINVAL);
        return;
    }

    memcpy(&thread->message_buffer, user_ptr, message_info->data_size);

    /* return values are set by ipc_reply() (or by ipc_receive() if the call
     * fails because the message is too big for the receiver's buffer) */
    thread->message_args = args;

    thread_t *recv_thread = jinue_node_entry(
            jinue_list_dequeue(&ipc->recv_list),
            thread_t,
            thread_list);

    if(recv_thread == NULL) {
        /* No thread is waiting to receive this message, so we must wait on the
         * sender list. */
        jinue_list_enqueue(&ipc->send_list, &thread->thread_list);

        thread_yield_from(
                thread,
                true,       /* make thread block */
                false);     /* don't destroy */
    }
    else {
        object_addref(&thread->header);
        recv_thread->sender = thread;

        /* switch to receiver thread, which will resume inside syscall_receive() */
        thread_switch(
                thread,
```

```
                recv_thread,
                true,       /* block sender thread */
                false);     /* don't destroy sender */
    }

    /* copy reply to user space buffer */
    memcpy(user_ptr, &thread->message_buffer, message_info->data_size);

}
```

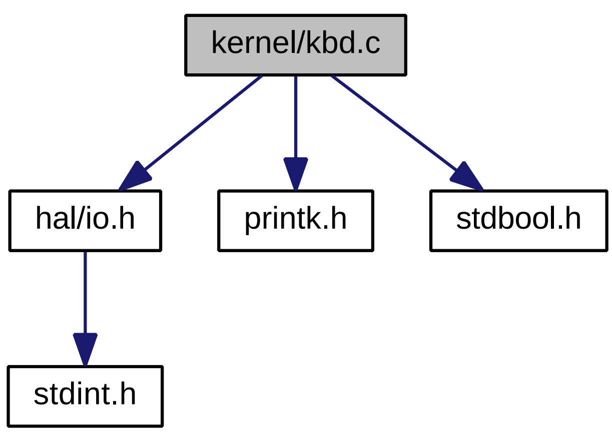
Here is the call graph for this function:



4.101 kernel/kbd.c File Reference

```
#include <hal/io.h>
#include <printk.h>
#include <stdbool.h>
```

Include dependency graph for kbd.c:



Functions

- void **any_key** (void)

4.101.1 Function Documentation

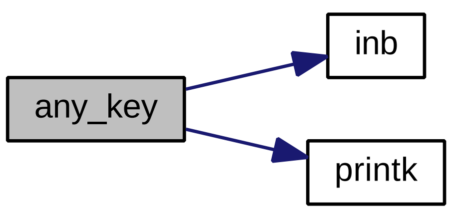
4.101.1.1 void any_key (void)

Definition at line 36 of file kbd.c.

References inb(), and printk().

```
                    {
    unsigned char buffer;
    bool ignore;

    /* prompt */
    printk("(press enter)");

    /* wait for key, ignore break codes */
    ignore = false;
    while(1) {
        do {
            buffer = inb(0x64);
        } while( (buffer & 1) == 0 );

        buffer = inb(0x60);

        if(buffer == 0x0e || buffer == 0x0f) {
            ignore = true;
            continue;
        }

        if(ignore) {
            ignore = false;
            continue;
        }

        if(buffer == 0x1c || buffer == 0x5a) {
            break;
        }
    }

    /* advance cursor */
    printk("\n");
}
```

Here is the call graph for this function:

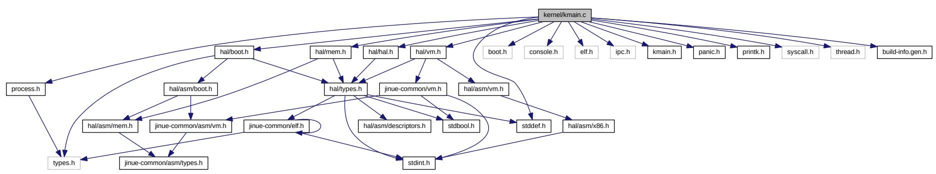


4.102 kernel/kmain.c File Reference

`#include <hal/boot.h>`

```
#include <hal/hal.h>
#include <hal/mem.h>
#include <hal/vm.h>
#include <boot.h>
#include <console.h>
#include <elf.h>
#include <ipc.h>
#include <kmain.h>
#include <panic.h>
#include <printk.h>
#include <process.h>
#include <stddef.h>
#include <syscall.h>
#include <thread.h>
#include "build-info.gen.h"
```

Include dependency graph for kmain.c:



Functions

- void **kmain** (void)

4.102.1 Function Documentation

4.102.1.1 void kmain (void)

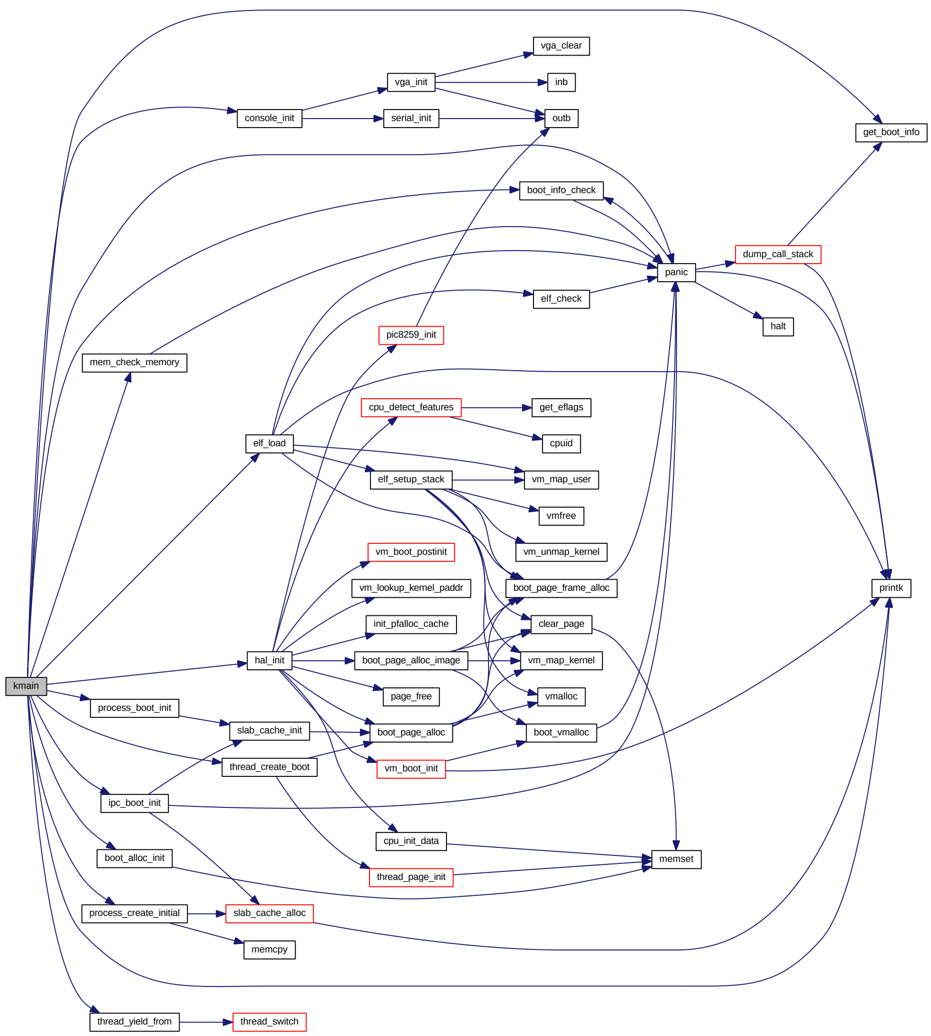
Definition at line 67 of file kmain.c.

References process_t::addr_space, boot_alloc_init(), boot_info_t::boot_heap, boot_info_check(), BUILD_HOST, BUILD_TIME, boot_info_t::cmdline, console_init(), elf_load(), elf_info_t::entry, get_boot_info(), GIT_REVISION, hal_init(), ipc_boot_init(), boot_info_t::kernel_size, mem_check_memory(), NULL, panic(), printk(), process_boot_init(), process_create_initial(), boot_info_t::ramdisk_size, boot_info_t::ramdisk_start, elf_info_t::stack_addr, thread_create_boot(), thread_yield_from(), and VGA_COLOR_YELLOW.

```
67                  {
68      elf_info_t elf_info;
69
70      /* initialize console and say hello */
71      console_init();
72
73      /* Say hello. */
74      printk("Kernel revision " GIT_REVISION " built " BUILD_TIME " on "
     BUILD_HOST "\n");
75
76      const boot_info_t *boot_info = get_boot_info();
77      (void)boot_info_check(true);
78
79      printk("Kernel size is %u bytes.\n", boot_info->kernel_size);
80
81      if(boot_info->ramdisk_start == 0 || boot_info->ramdisk_size == 0) {
82          printk("%kWarning: no initial RAM disk loaded.\n", VGA_COLOR_YELLOW);
83      }
84      else {
```

```
        printk("RAM disk with size %u bytes loaded at address %x.\n", boot_info->ramdisk_size, boot_info->ramdisk_start);
    }

    printk("Kernel command line:\n", boot_info->kernel_size);
    printk("    %s\n", boot_info->cmdline);

    /* Initialize the boot allocator. */
    boot_alloc_t boot_alloc;
    boot_alloc_init(&boot_alloc, boot_info->boot_heap);
    mem_check_memory(&boot_alloc, boot_info);

    /* initialize hardware abstraction layer */
    hal_init(&boot_alloc, boot_info);

    /* initialize caches */
    ipc_boot_init(&boot_alloc);
    process_boot_init(&boot_alloc);

    /* create process for process manager */
    process_t *process = process_create_initial();

    if(process == NULL) {
        panic("Could not create initial process.");
    }

    /* load process manager binary */
    Elf32_Ehdr *elf = find_process_manager();
    elf_load(&elf_info, elf, &process->addr_space, &boot_alloc);

    /* create initial thread */
    thread_t *thread = thread_create_boot(
            process,
            elf_info.entry,
            elf_info.stack_addr,
            &boot_alloc);

    if(thread == NULL) {
        panic("Could not create initial thread.");
    }

    /* start process manager
     *
     * We switch from NULL since this is the first thread. */
    thread_yield_from(
            NULL,
            false,      /* don't block */
            false);     /* don't destroy */
                        /* just be nice */

    /* should never happen */
    panic("thread_yield_from() returned in kmain()");
}
```

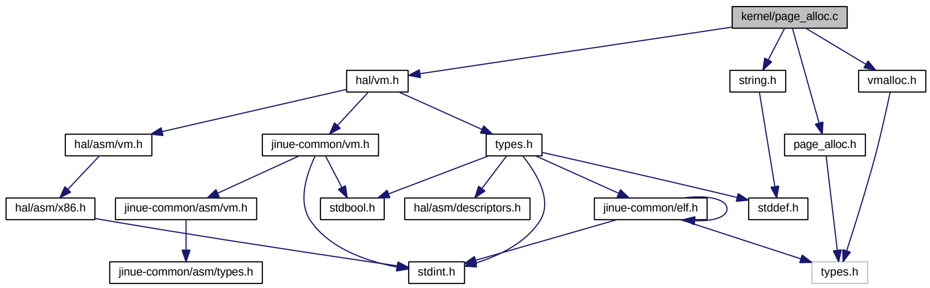
Here is the call graph for this function:



4.103 kernel/page_alloc.c File Reference

```
#include <hal/vm.h>
#include <page_alloc.h>
#include <string.h>
#include <vmalloc.h>
```

Include dependency graph for page_alloc.c:



Data Structures

- struct **alloc_page**

Functions

- void ∗ **page_alloc** (void)

    Allocate a page of kernel memory.
- void **page_free** (void ∗page)

    Free a page of kernel memory.
- **bool page_alloc_is_empty** (void)

    Check that pages are available to be allocated.
- **bool add_page_frame** (**kern_paddr_t** paddr)

    Map a page frame and add it to the page allocator.
- **kern_paddr_t remove_page_frame** (void)

    Remove a page frame from the allocator.
- void **clear_page** (void ∗page)

    Clear a page by writing all bytes to zero.

4.103.1 Function Documentation

4.103.1.1 bool add_page_frame (kern_paddr_t *paddr*)

Map a page frame and add it to the page allocator.

This function is used to implement a system call that allows userspace to provide additional page frames to the kernel. This function fails when no more pages of kernel address space can be allocated with **vmalloc()** (p. 275) to map the provided page frame.

Parameters

paddr	physical address of the provided page frame

Returns

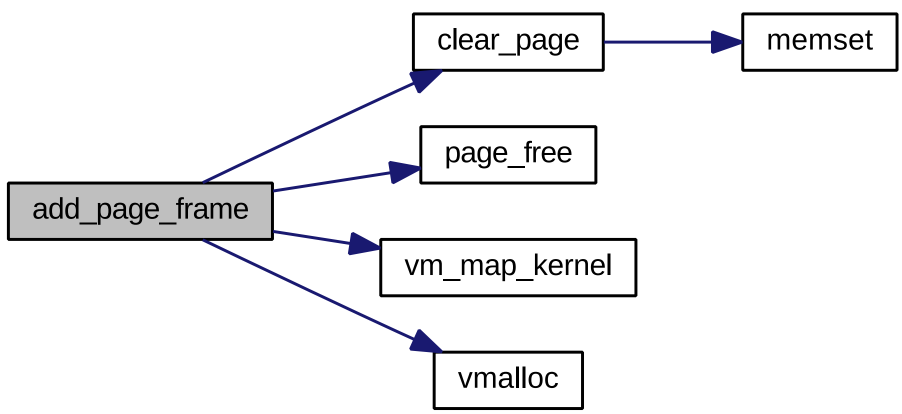
true if the function succeeded

Definition at line 111 of file page_alloc.c.

References clear_page(), NULL, page_free(), VM_FLAG_READ_WRITE, vm_map_kernel(), and vmalloc().

```
111                                     {
112     void *page = vmalloc();
113
114     if(page == NULL) {
115         return false;
116     }
117
118     vm_map_kernel(page, paddr, VM_FLAG_READ_WRITE);
119
120     /* Since this page is coming from userspace, is is important to clear it:
121      * 1) The page may contain sensitive information, which we don't want to
122      *    leak through Meltdown-like vulnerabilities; and
123      * 2) Since the content is userspace-chosen, it could be used for kernel
124      *    vulnerability exploits. */
125     clear_page(page);
126     page_free(page);
127
128     return true;
129 }
```

Here is the call graph for this function:



4.103.1.2 void clear_page (void ∗ *page*)

Clear a page by writing all bytes to zero.

Parameters

page	the page to clear

Definition at line 173 of file page_alloc.c.

References memset(), and PAGE_SIZE.

Referenced by add_page_frame(), boot_page_alloc(), boot_page_alloc_early(), boot_page_alloc_image(), elf_setup_-stack(), and remove_page_frame().

```
173                         {
174     memset(page, 0, PAGE_SIZE);
175 }
```

Here is the call graph for this function:

4.103.1.3 void∗ page_alloc (void)

Allocate a page of kernel memory.

Pages allocated by this function can be used for any purpose in the kernel, e.g. as slabs for the slab allocator or as page tables.

Pages allocated by this function are not guaranteed to be mapped in the allocations region of the kernel address space (that is, the region managed by **vmalloc()** (p. 275)). While most will be, pages originally allocated in the image region during initialization by calling **boot_page_alloc_image()** (p. 72) can be reclaimed with **page_free()** (p. 252) and then re-allocated by this function.

Returns

allocated page

Definition at line 59 of file page_alloc.c.

References alloc_page::next, and NULL.

Referenced by remove_page_frame(), slab_cache_alloc(), thread_create(), and vm_clone_page_directory().

```
59                      {
60     struct alloc_page *alloc_page = head_page;
61
62     if(alloc_page != NULL) {
63         head_page = alloc_page->next;
64     }
65
66     return alloc_page;
67 }
```

4.103.1.4 bool page_alloc_is_empty (void)

Check that pages are available to be allocated.

Page availability can be checked with this function before calling either **page_alloc()** (p. 251) or **remove_page_frame()** (p. 253).

Returns

true if pages are available (one or more)

Definition at line 95 of file page_alloc.c.

References NULL.

```
95                              {
96     return head_page == NULL;
97 }
```

4.103.1.5 void page_free (void ∗ *page*)

Free a page of kernel memory.

Pages freed by calling this function are available to be re-allocated by the **page_alloc()** (p. 251) function. This function can be used to free pages allocated by **page_alloc()** (p. 251) or to reclaim pages allocated during kernel initialization by **boot_page_alloc()** (p. 70) or **boot_page_alloc_image()** (p. 72).

Parameters

page	the page to free

Definition at line 80 of file page_alloc.c.

References alloc_page::next.

Referenced by add_page_frame(), hal_init(), and thread_destroy().

```
80                          {
81     struct alloc_page *alloc_page = page;
82     alloc_page->next    = head_page;
83     head_page           = alloc_page;
84 }
```

4.103.1.6 kern_paddr_t remove_page_frame (void)

Remove a page frame from the allocator.

This function is used implement a system call that allows userspace to reclaim free kernel memory for its own use. The address space page is freed with **vmfree()** (p. 276) and the physical address of the underlying page frame is returned.

Returns

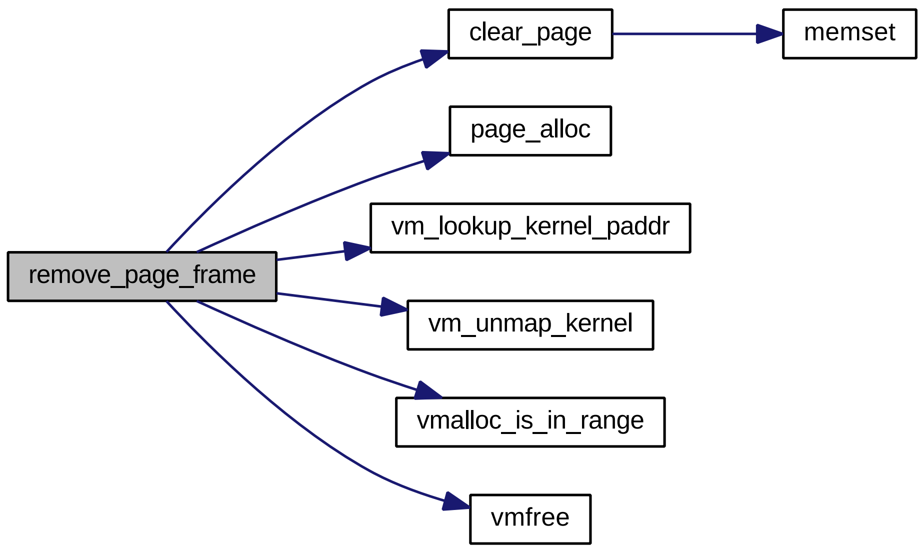
physical address of the freed page frame, or PFNULL if none is available

Definition at line 142 of file page_alloc.c.

References clear_page(), NULL, page_alloc(), PFNULL, vm_lookup_kernel_paddr(), vm_unmap_kernel(), vmalloc_is_-in_range(), and vmfree().

```
142                                  {
143     void *page = page_alloc();
144 
145     if(page == NULL) {
146         return PFNULL;
147     }
148 
149     /* This page is going to userspace. Let's clear its content so we don't
150      * leak information about the kernel's internal state that could be useful
151      * for exploiting vulnerabilities. */
152     clear_page(page);
153 
154     kern_paddr_t paddr = vm_lookup_kernel_paddr(page);
155 
156     vm_unmap_kernel(page);
157 
158     /* The page may be in the image region instead of the allocations region if
159      * it was allocated during kernel initialization. */
160     if(vmalloc_is_in_range(page)) {
161         vmfree(page);
162     }
163 
164     return paddr;
165 }
```

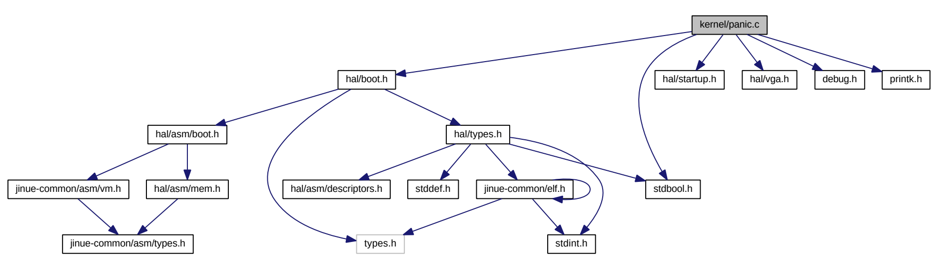
Here is the call graph for this function:



4.104 kernel/panic.c File Reference

```
#include <hal/boot.h>
#include <hal/startup.h>
#include <hal/vga.h>
#include <debug.h>
#include <printk.h>
#include <stdbool.h>
```

Include dependency graph for panic.c:



Functions

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

4.104.1 Function Documentation

4.104.1.1 void panic (const char ∗ *message*)

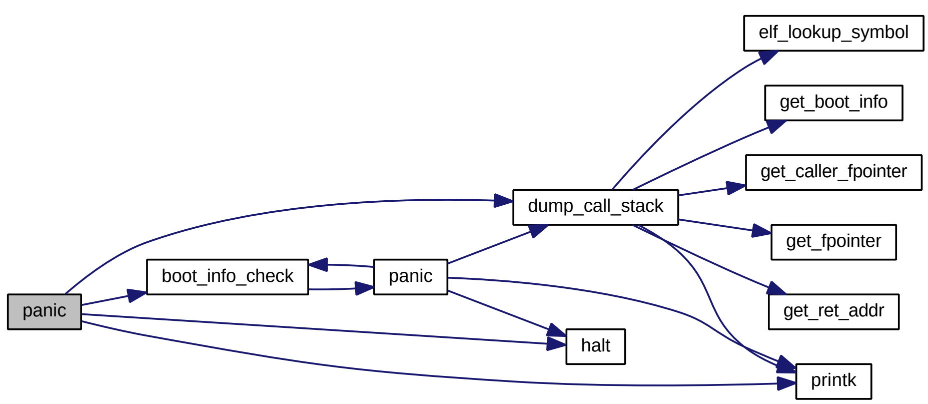
Definition at line 40 of file panic.c.

References boot_info_check(), dump_call_stack(), halt(), printk(), and VGA_COLOR_RED.

Referenced by __assert_failed(), boot_heap_pop(), boot_info_check(), boot_page_alloc_early(), boot_page_frame_-alloc(), boot_vmalloc(), dispatch_interrupt(), elf_check(), elf_load(), ipc_boot_init(), kmain(), mem_check_memory(), and pfalloc_from().

```
{
    static int enter_count = 0;

    ++enter_count;

    /* When things go seriously wrong, things that panic does itself can create
     * a further panic, for example by triggering a hardware exception. The
     * enter_count static variable keeps count of the number of times panic()
     * is entered. */
    switch(enter_count) {
    case 1:
    case 2:
        /* The first two times panic() is entered, a panic message is displayed
         * along with a full call sack dump. */
        printk( "%kKERNEL PANIC%s: %s\n",
                VGA_COLOR_RED,
                enter_count==1?"":" (recursive)",
                message);

        if( boot_info_check(false) ) {
            dump_call_stack();
        }
        else {
            printk("Cannot dump call stack because boot information structure is invalid.\n");
        }
        break;
    case 3:
        /* The third time, a "recursive count exceeded" message is displayed. We
         * try to limit the number of actions we take to limit the chances of a
         * further panic. */
        printk("%kKERNEL PANIC (recursive count exceeded)\n", VGA_COLOR_RED);
        break;
    default:
        /* The fourth time, we do nothing but halt the CPU. */
        break;
    }

    halt();
}
```

Here is the call graph for this function:

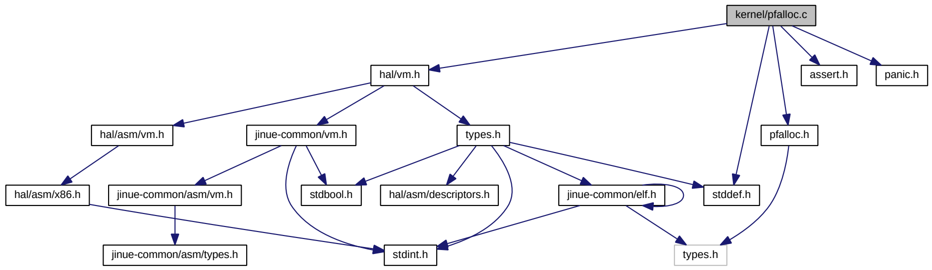


4.105 kernel/pfalloc.c File Reference

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

```
#include <assert.h>
#include <panic.h>
#include <pfalloc.h>
#include <stddef.h>
```

Include dependency graph for pfalloc.c:



Functions

- void **init_pfalloc_cache** (**pfalloc_cache_t** ∗pfcache, **kern_paddr_t** ∗stack_page)
- **kern_paddr_t pfalloc_from** (**pfalloc_cache_t** ∗pfcache)
- void **pffree_to** (**pfalloc_cache_t** ∗pfcache, **kern_paddr_t** paddr)

Variables

- **pfalloc_cache_t global_pfalloc_cache**

4.105.1 Function Documentation

4.105.1.1 void init_pfalloc_cache (pfalloc_cache_t ∗ *pfcache*, kern_paddr_t ∗ *stack_page*)

Definition at line 40 of file pfalloc.c.

References pfalloc_cache_t::count, KERNEL_PAGE_STACK_SIZE, PFNULL, and pfalloc_cache_t::ptr.

Referenced by hal_init().

```
40                                                                      {
41     kern_paddr_t    *ptr;
42     unsigned int     idx;
43 
44     ptr = stack_page;
45 
46     for(idx = 0;idx < KERNEL_PAGE_STACK_SIZE; ++idx) {
47         ptr[idx] = PFNULL;
48     }
49 
50     pfcache->ptr   = stack_page;
51     pfcache->count = 0;
52 }
```

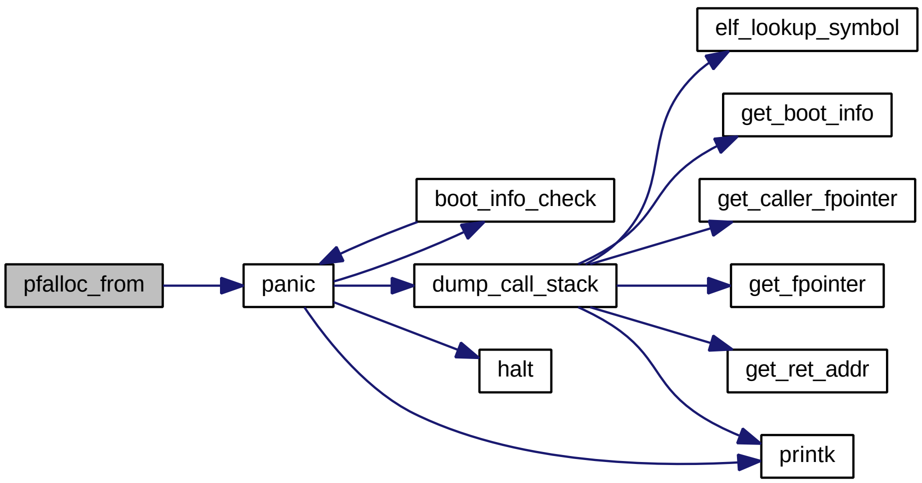
4.105.1.2 kern_paddr_t pfalloc_from (pfalloc_cache_t ∗ *pfcache*)

Definition at line 54 of file pfalloc.c.

References pfalloc_cache_t::count, panic(), and pfalloc_cache_t::ptr.

```
                                                  {
    if(pfcache->count == 0) {
        panic("pfalloc_from(): no more pages to allocate");
    }

    --pfcache->count;

    return *(--pfcache->ptr);
}
```

Here is the call graph for this function:



4.105.1.3 void pffree_to (pfalloc_cache_t * *pfcache,* kern_paddr_t *paddr*)

We are leaking memory here. Should we panic instead?

Definition at line 64 of file pfalloc.c.

References pfalloc_cache_t::count, KERNEL_PAGE_STACK_SIZE, and pfalloc_cache_t::ptr.

```
64                                                          {
65      if(pfcache->count >= KERNEL_PAGE_STACK_SIZE) {
67          return;
68      }
69
70      ++pfcache->count;
71
72      (pfcache->ptr++)[0] = paddr;
73  }
```

4.105.2 Variable Documentation

4.105.2.1 pfalloc_cache_t global_pfalloc_cache

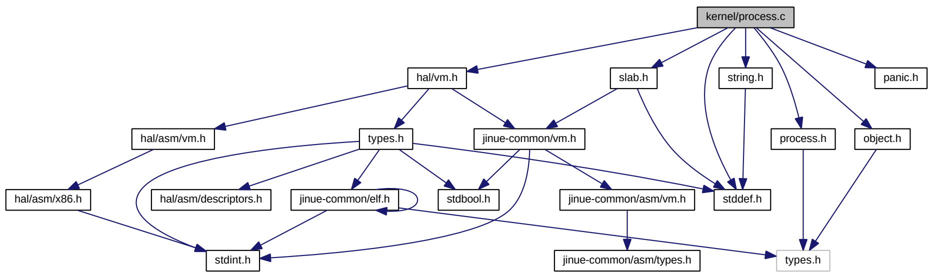
Definition at line 38 of file pfalloc.c.

Referenced by hal_init().

4.106 kernel/process.c File Reference

```
#include <hal/vm.h>
#include <panic.h>
#include <process.h>
#include <object.h>
#include <slab.h>
#include <stddef.h>
#include <string.h>
```

Include dependency graph for process.c:



Functions

- void **process_boot_init** (**boot_alloc_t** ∗boot_alloc)
- **process_t** ∗ **process_create** (void)
- **process_t** ∗ **process_create_initial** (void)
- **object_ref_t** ∗ **process_get_descriptor** (**process_t** ∗process, int fd)
- int **process_unused_descriptor** (**process_t** ∗process)

4.106.1 Function Documentation

4.106.1.1 void process_boot_init (boot_alloc_t ∗ *boot_alloc*)

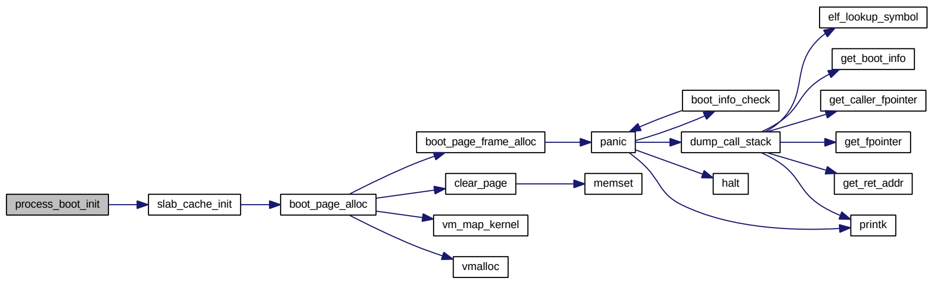
Definition at line 49 of file process.c.

References NULL, slab_cache_init(), and SLAB_DEFAULTS.

Referenced by kmain().

```
49                                                {
50     slab_cache_init(
51             &process_cache,
52             "process_cache",
53             sizeof(process_t),
54             0,
55             process_ctor,
56             NULL,
57             SLAB_DEFAULTS,
58             boot_alloc);
59 }
```

Here is the call graph for this function:



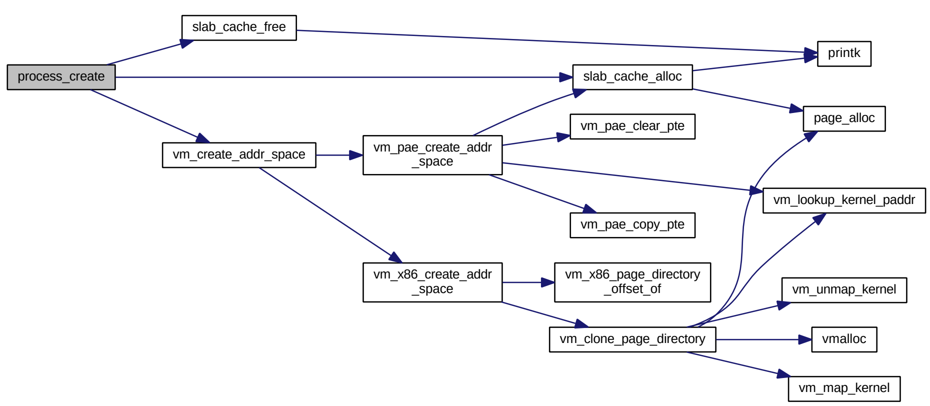
4.106.1.2 process_t∗ process_create (void)

Definition at line 65 of file process.c.

References process_t::addr_space, NULL, slab_cache_alloc(), slab_cache_free(), and vm_create_addr_space().

```
65                              {
66     process_t *process = slab_cache_alloc(&process_cache);
67
68     if(process != NULL) {
69         addr_space_t *addr_space = vm_create_addr_space(&process->addr_space);
70
71         /* The address space object is located inside the process object but the
72          * call to vm_create_addr_space() above can still fail if we cannot
73          * allocate the initial page directory/tables or, when PAE is enabled,
74          * if we cannot allocate a PDPT. */
75         if(addr_space == NULL) {
76             slab_cache_free(process);
77             return NULL;
78         }
79
80         process_init(process);
81     }
82
83     return process;
84 }
```

Here is the call graph for this function:



4.106.1.3 process_t∗ process_create_initial (void)

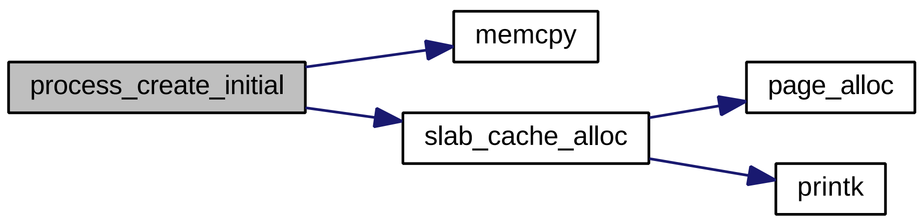
Definition at line 86 of file process.c.

References process_t::addr_space, initial_addr_space, memcpy(), NULL, and slab_cache_alloc().

Referenced by kmain().

```
86                                           {
87     process_t *process = slab_cache_alloc(&process_cache);
88
89     if(process != NULL) {
90         memcpy(&process->addr_space, &initial_addr_space, sizeof(addr_space_t));
91         process_init(process);
92     }
93
94     return process;
95 }
```

Here is the call graph for this function:



4.106.1.4 object_ref_t∗ process_get_descriptor (process_t ∗ *process,* int *fd*)

Definition at line 97 of file process.c.

References process_t::descriptors, NULL, and PROCESS_MAX_DESCRIPTORS.

Referenced by dispatch_syscall(), ipc_receive(), ipc_send(), and process_unused_descriptor().

```
97                                                                {
98     if(fd < 0 || fd > PROCESS_MAX_DESCRIPTORS) {
99         return NULL;
100     }
101
102     return &process->descriptors[fd];
103 }
```

4.106.1.5 int process_unused_descriptor (process_t ∗ *process*)

Definition at line 105 of file process.c.

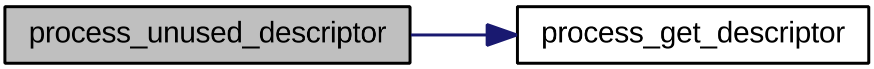
References process_get_descriptor(), and PROCESS_MAX_DESCRIPTORS.

Referenced by dispatch_syscall().

```
105                                                   {
106     int idx;
107
108     for(idx = 0; idx < PROCESS_MAX_DESCRIPTORS; ++idx) {
109         object_ref_t *ref = process_get_descriptor(process, idx);
110
111         if(! object_ref_is_valid(ref)) {
```

```
                return idx;
            }
        }

        return -1;
}
```

Here is the call graph for this function:

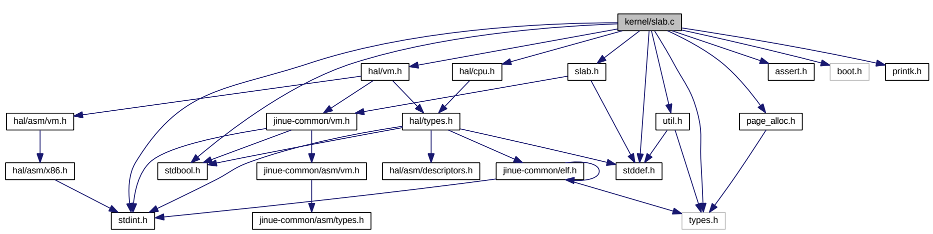


4.107 kernel/slab.c File Reference

Kernel object allocator.

```
#include <hal/cpu.h>
#include <hal/vm.h>
#include <assert.h>
#include <boot.h>
#include <page_alloc.h>
#include <printk.h>
#include <slab.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <types.h>
#include <util.h>
```

Include dependency graph for slab.c:



Functions

- void **slab_cache_init** (**slab_cache_t** ∗cache, char ∗name, **size_t** size, **size_t** alignment, **slab_ctor_t** ctor, **slab-_ctor_t** dtor, int flags, **boot_alloc_t** ∗boot_alloc)

    Initialize an object cache.
- void ∗ **slab_cache_alloc** (**slab_cache_t** ∗cache)

    Allocate an object from the specified cache.
- void **slab_cache_free** (void ∗buffer)

    Free an object.
- void **slab_cache_reap** (**slab_cache_t** ∗cache)

    Return memory to the page allocator.
- void **slab_cache_set_working_set** (**slab_cache_t** ∗cache, unsigned int n)

    Set a cache's working set.

4.107.1 Detailed Description

Kernel object allocator. This file implements a slab allocator as described in Jeff Bonwick's paper "The Slab Allocator: An Object-Caching Kernel Memory Allocator":

`https://www.usenix.org/publications/library/proceedings/bos94/full_papers/bonwick.-ps`

This is the main object allocator for the kernel. (Some early allocations performed during kernel initialization use the boot heap instead - see boot.c.)

Definition in file **slab.c**.

4.107.2 Function Documentation

4.107.2.1 void∗ slab_cache_alloc (slab_cache_t ∗ *cache*)

Allocate an object from the specified cache.

The cache must have been initialized with **slab_cache_init()** (p. 270). If no more space is available on existing slabs, this function tries to allocate a new slab using the kernel's page allocator (i.e. **page_alloc()** (p. 251)). It page allocation fails, this function fails by returning NULL.

Parameters

cache	the cache from which to allocate an object

Returns

the address of the allocated object, or NULL if allocation failed

ASSERTION: now that slab_cache_grow() has run, we should have found at least one empty slab

Important note regarding the slab lists: The empty, partial and full slab lists are doubly-linked lists. This is done to allow the deletion of an arbitrary link given a pointer to it. We do not allow reverse traversal: we do not maintain a tail pointer and, more importantly: we do *NOT* maintain the previous pointer of the first link in the list (i.e. it is garbage data, not NULL).

ASSERTION: there is at least one buffer on the free list

ASSERT: the slab is the head of the partial list

Definition at line 232 of file slab.c.

References assert, slab_cache_t::bufctl_offset, slab_cache_t::ctor, slab_cache_t::empty_count, slab_cache_t::flags, slab_t::free_list, slab_cache_t::name, slab_bufctl_t::next, slab_t::next, NULL, slab_t::obj_count, slab_cache_t::obj_size, page_alloc(), slab_t::prev, printk(), SLAB_POISON, SLAB_POISON_ALIVE_VALUE, SLAB_POISON_DEAD_VALU-E, SLAB_RED_ZONE, SLAB_RED_ZONE_VALUE, slab_cache_t::slabs_empty, slab_cache_t::slabs_full, and slab_-cache_t::slabs_partial.

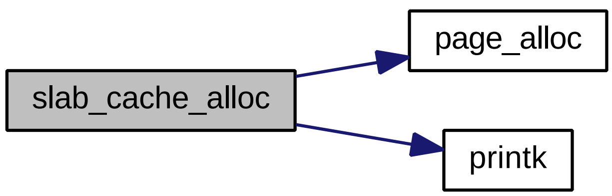
Referenced by ipc_boot_init(), ipc_object_create(), process_create(), process_create_initial(), and vm_pae_create_-addr_space().

```
                                            {
    slab_t          *slab;

    if(cache->slabs_partial != NULL) {
        slab = cache->slabs_partial;
    }
    else {
        if(cache->slabs_empty == NULL) {
            void *slab_addr = page_alloc();
```

```c

            if(slab_addr == NULL) {
                return NULL;
            }

            init_and_add_slab(cache, slab_addr);
        }

        slab = cache->slabs_empty;

        assert(slab != NULL);

        /* We are about to allocate one object from this slab, so it will
         * not be empty anymore...*/
        cache->slabs_empty      = slab->next;

        --(cache->empty_count);

        slab->next              = cache->slabs_partial;
        if(slab->next != NULL) {
            slab->next->prev = slab;
        }
        cache->slabs_partial    = slab;
    }

    slab_bufctl_t *bufctl = slab->free_list;

    assert(bufctl != NULL);

    slab->free_list  = bufctl->next;
    slab->obj_count += 1;

    /* If we just allocated the last buffer, move the slab to the full
     * list */
    if(slab->free_list == NULL) {
        /* remove from the partial slabs list */

        assert(cache->slabs_partial == slab);

        cache->slabs_partial = slab->next;

        if(slab->next != NULL) {
            slab->next->prev = slab->prev;
        }

        /* add to the full slabs list */
        slab->next        = cache->slabs_full;
        cache->slabs_full = slab;

        if(slab->next != NULL) {
            slab->next->prev = slab;
        }
    }

    uint32_t *buffer = (uint32_t *)( (char *)bufctl - cache->bufctl_offset );

    if(cache->flags & SLAB_POISON) {
        unsigned int idx;
        unsigned int dump_lines = 0;

        for(idx = 0; idx < cache->obj_size / sizeof(uint32_t); ++idx) {
            if(buffer[idx] != SLAB_POISON_DEAD_VALUE) {
                if(dump_lines == 0) {
                    printk("detected write to freed object, cache: %s buffer: 0x%x:\n",
                        cache->name,
                        (unsigned int)buffer
                    );
                }

                if(dump_lines < 4) {
                    printk(" value 0x%x at byte offset %u\n", buffer[idx], idx * sizeof(
  uint32_t));
                }

                ++dump_lines;
            }

            buffer[idx] = SLAB_POISON_ALIVE_VALUE;
        }

        /* If both SLAB_POISON and SLAB_RED_ZONE are enabled, we perform
```

```
333          * redzone checking even on freed objects. */
334         if(cache->flags & SLAB_RED_ZONE) {
335             if(buffer[idx] != SLAB_RED_ZONE_VALUE) {
336                 printk("detected write past the end of freed object, cache: %s buffer: 0x%x value: 0x%x\n",
337                     cache->name,
338                     (unsigned int)buffer,
339                     buffer[idx]
340                 );
341             }
342 
343             buffer[idx] = SLAB_RED_ZONE_VALUE;
344         }
345 
346         if(cache->ctor != NULL) {
347             cache->ctor((void *)buffer, cache->obj_size);
348         }
349     }
350     else if(cache->flags & SLAB_RED_ZONE) {
351         buffer[cache->obj_size / sizeof(uint32_t)] = SLAB_RED_ZONE_VALUE;
352     }
353 
354     return (void *)buffer;
355 }
```

Here is the call graph for this function:



4.107.2.2 void slab_cache_free (void ∗ *buffer*)

Free an object.

Parameters

buffer	the object to free

Definition at line 363 of file slab.c.

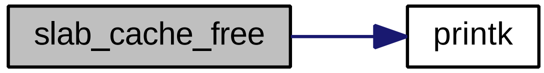
References ALIGN_START_PTR, slab_cache_t::bufctl_offset, slab_t::cache, slab_cache_t::dtor, slab_cache_t::empty-_count, slab_cache_t::flags, slab_t::free_list, slab_cache_t::name, slab_bufctl_t::next, slab_t::next, NULL, slab_t::obj_-count, slab_cache_t::obj_size, slab_t::prev, printk(), SLAB_POISON, SLAB_POISON_DEAD_VALUE, SLAB_RED_Z-ONE, SLAB_RED_ZONE_VALUE, SLAB_SIZE, slab_cache_t::slabs_empty, slab_cache_t::slabs_full, and slab_cache-_t::slabs_partial.

Referenced by process_create(), and vm_pae_destroy_addr_space().

```
363                                 {
364     /* compute address of slab data structure */
365     addr_t slab_start       = ALIGN_START_PTR(buffer, SLAB_SIZE);
366     slab_t *slab            = (slab_t *)(slab_start + SLAB_SIZE - sizeof(slab_t) );
367 
368     /* obtain address of cache and bufctl */
369     slab_cache_t *cache     = slab->cache;
370     slab_bufctl_t *bufctl   = (slab_bufctl_t *)((char *)buffer + cache->
      bufctl_offset);
371 
372     /* If slab is on the full slabs list, move it to the partial list
373      * since we are about to return a buffer to it. */
374     if(slab->free_list == NULL) {
375         /* remove from full slabs list */
```

```
        if(cache->slabs_full == slab) {
            cache->slabs_full = slab->next;
        }
        else {
            slab->prev->next = slab->next;
        }

        if(slab->next != NULL) {
            slab->next->prev = slab->prev;
        }

        /* add to partial slabs list */
        slab->next           = cache->slabs_partial;
        cache->slabs_partial = slab;

        if(slab->next != NULL) {
            slab->next->prev = slab;
        }
    }

    if(cache->flags & SLAB_RED_ZONE) {
        uint32_t *rz_word = (uint32_t *)((char *)buffer + cache->obj_size);

        if(*rz_word != SLAB_RED_ZONE_VALUE) {
            printk("detected write past the end of object, cache: %s buffer: 0x%x value: 0x%x\n",
                cache->name,
                (unsigned int)buffer,
                *rz_word
            );
        }

        *rz_word = SLAB_RED_ZONE_VALUE;
    }

    if(cache->flags & SLAB_POISON) {
        unsigned int idx;

        if(cache->dtor != NULL) {
            cache->dtor(buffer, cache->obj_size);
        }

        uint32_t *buffer32 = (uint32_t *)buffer;

        for(idx = 0; idx < cache->obj_size / sizeof(uint32_t); ++idx) {
            buffer32[idx] = SLAB_POISON_DEAD_VALUE;
        }
    }

    /* link buffer into slab free list */
    bufctl->next     = slab->free_list;
    slab->free_list  = bufctl;
    slab->obj_count -= 1;

    /* If we just returned the last object to the slab, move the slab to
     * the empty list. */
    if(slab->obj_count == 0) {
        /* remove from partial slabs list */
        if(cache->slabs_partial == slab) {
            cache->slabs_partial = slab->next;
        }
        else {
            slab->prev->next = slab->next;
        }

        if(slab->next != NULL) {
            slab->next->prev = slab->prev;
        }

        /* add to empty slabs list */
        slab->next         = cache->slabs_empty;
        cache->slabs_empty = slab;

        if(slab->next != NULL) {
            slab->next->prev = slab;
        }

        ++(cache->empty_count);
    }
}
```

Here is the call graph for this function:



4.107.2.3 void slab_cache_init (slab_cache_t * *cache*, char * *name*, size_t *size*, size_t *alignment*, slab_ctor_t *ctor*, slab_ctor_t *dtor*, int *flags*, boot_alloc_t * *boot_alloc*)

Initialize an object cache.

The following flags are supported:

- SLAB_HWCACHE_ALIGN Align objects on at least the line size of the CPU's data cache.
- SLAB_COMPACT the bufctl can safely be put inside the object without destroying the constructed state. If not set, additional space is reserved specifically for the bufctl to prevent corruption of the constructed state.
- SLAB_RED_ZONE (redzone checking - debugging) Add a guard word at the end of each object and use this to detect writes past the end of the object.
- SLAB_POISON (debugging) Fill uninitialized objects with a recognizable pattern before calling the constructor function to help identify members that do not get initialized. Do the same when freeing objects and use this to detect writes to freed objects.

This function uses the kernel's boot-time page allocator to allocate an initial slab. This helps with bootstrapping because it allows a few objects (up to s slab's worth) to be allocated before the main page allocator has been initialized and then replenished by user space. It also means this function can only be called during kernel initialization (it would not make sense to call it later).

Parameters

cache	the cache to initialize
name	a human-readable name for the cache, used in debugging messages
size	the size of objects allocated on this cache
alignment	the minimum object alignment, or zero for no constraint
ctor	the object constructor function
dtor	the object destructor function
flags	see description
boot_alloc	the kernel boot-time page allocator structure

ASSERTION: buffer size is at least the size of a pointer

ASSERTION: name is not NULL string

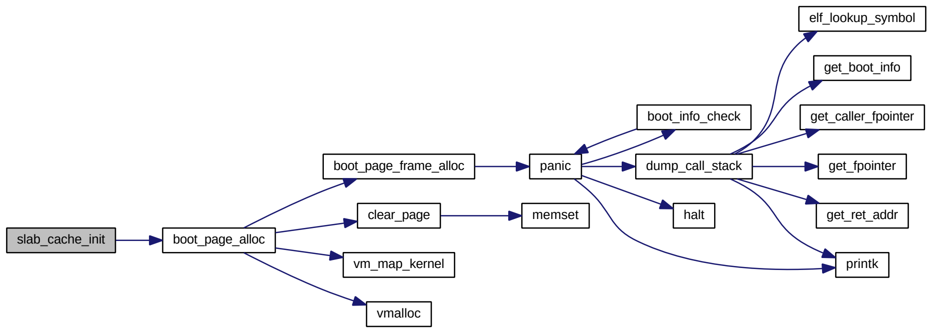
Definition at line 137 of file slab.c.

References ALIGN_END, slab_cache_t::alignment, slab_cache_t::alloc_size, assert, boot_page_alloc(), slab_cache_t::bufctl_offset, cpu_info, slab_cache_t::ctor, cpu_info_t::dcache_alignment, slab_cache_t::dtor, slab_cache_t::empty_count, slab_cache_t::flags, slab_cache_t::max_colour, slab_cache_t::name, slab_cache_t::next_colour, NULL, slab_cache_t::obj_size, SLAB_COMPACT, SLAB_DEFAULT_WORKING_SET, SLAB_HWCACHE_ALIGN, SLAB_POISON, SLAB_RED_ZONE, SLAB_SIZE, slab_cache_t::slabs_empty, slab_cache_t::slabs_full, slab_cache_t::slabs_partial, and slab_cache_t::working_set.

Referenced by ipc_boot_init(), process_boot_init(), and vm_pae_create_pdpt_cache().

```
                                      {

    assert(size >= sizeof(void *));

    assert(name != NULL);

    cache->name             = name;
    cache->ctor             = ctor;
    cache->dtor             = dtor;
    cache->slabs_empty      = NULL;
    cache->slabs_partial    = NULL;
    cache->slabs_full       = NULL;
    cache->empty_count      = 0;
    cache->flags            = flags;
    cache->next_colour      = 0;
    cache->working_set      = SLAB_DEFAULT_WORKING_SET;

    /* Compute actual alignment. */
    if(alignment == 0) {
        cache->alignment = sizeof(uint32_t);
    }
    else {
        cache->alignment = alignment;
    }

    if((flags & SLAB_HWCACHE_ALIGN) && cache->alignment < cpu_info.dcache_alignment) {
        cache->alignment = cpu_info.dcache_alignment;
    }

    cache->alignment = ALIGN_END(cache->alignment, sizeof(uint32_t));

    /* Reserve space for bufctl and/or redzone word. */
    cache->obj_size = ALIGN_END(size, sizeof(uint32_t));

    if((flags & SLAB_POISON) && (flags & SLAB_RED_ZONE)) {
        /* bufctl and redzone word appended to buffer */
        cache->alloc_size = cache->obj_size + sizeof(uint32_t) + sizeof(slab_bufctl_t);
    }
    else if((flags & SLAB_POISON) || (flags & SLAB_RED_ZONE)) {
        /* bufctl or redzone word appended to buffer (can be shared) */
        cache->alloc_size = cache->obj_size + sizeof(uint32_t);
    }
    else if(ctor != NULL && ! (flags & SLAB_COMPACT)) {
        /* If a constructor is defined, we cannot put the bufctl inside
         * the object because that could overwrite constructed state,
         * unless client explicitly says it's ok (SLAB_COMPACT flag). */
        cache->alloc_size = cache->obj_size + sizeof(slab_bufctl_t);
    }
    else {
        cache->alloc_size = cache->obj_size;
    }

    if(cache->alloc_size % cache->alignment != 0) {
        cache->alloc_size += cache->alignment - cache->alloc_size % cache->alignment;
    }

    size_t avail_space = SLAB_SIZE - sizeof(slab_t);

    unsigned int buffers_per_slab = avail_space / cache->alloc_size;

    size_t wasted_space = avail_space - buffers_per_slab * cache->alloc_size;

    cache->max_colour = (wasted_space / cache->alignment) * cache->alignment;

    cache->bufctl_offset = cache->alloc_size - sizeof(slab_bufctl_t);

    /* Allocate first slab.
     *
     * This is needed to allow a few objects to be allocated during kernel
     * initialization. */
    init_and_add_slab(cache, boot_page_alloc(boot_alloc));
}
```

Here is the call graph for this function:



4.107.2.4 void slab_cache_reap (slab_cache_t ∗ *cache*)

Return memory to the page allocator.

Free slabs in excess to the cache's working set are finalized and freed.

Parameters

cache	the cache from which to reclaim memory

Definition at line 544 of file slab.c.

References slab_cache_t::empty_count, slab_t::next, slab_cache_t::slabs_empty, and slab_cache_t::working_set.

```
544                                       {
545     while(cache->empty_count > cache->working_set) {
546         /* select the first empty slab */
547         slab_t *slab = cache->slabs_empty;
548 
549         /* unlink it and update count */
550         cache->slabs_empty  = slab->next;
551         cache->empty_count -= 1;
552 
553         /* destroy slab */
554         destroy_slab(cache, slab);
555     }
556 }
```

4.107.2.5 void slab_cache_set_working_set (slab_cache_t ∗ *cache,* unsigned int *n*)

Set a cache's working set.

The working set is defined as the number of free slabs the cache keeps for itself when pages are reclaimed from it. (This is terminology used in the Bonwick paper.) This provides some hysteresis to prevent slabs from being continuously created and destroyed, which requires calling the constructor and destructor functions on individual objects on the slabs.

Parameters

cache	the cache for which to set the working set
n	the size of the working set (number of pages)

Definition at line 571 of file slab.c.

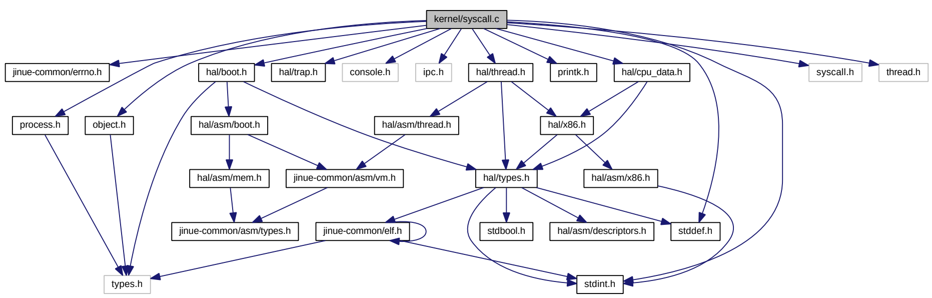
References slab_cache_t::working_set.

```
                                                                    {
    cache->working_set = n;
}
```

4.108 kernel/syscall.c File Reference

```
#include <jinue-common/errno.h>
#include <hal/boot.h>
#include <hal/cpu_data.h>
#include <hal/thread.h>
#include <hal/trap.h>
#include <console.h>
#include <ipc.h>
#include <object.h>
#include <printk.h>
#include <process.h>
#include <stddef.h>
#include <stdint.h>
#include <syscall.h>
#include <thread.h>
```

Include dependency graph for syscall.c:



Functions

- void **dispatch_syscall** (**trapframe_t** ∗trapframe)

4.108.1 Function Documentation

4.108.1.1 void dispatch_syscall (trapframe_t ∗ *trapframe*)

TODO for check negative values (especially -1)

TODO: permission check

TODO: permission check, sanity check (data size vs buffer size)

TODO: check user pointer

Definition at line 48 of file syscall.c.

References jinue_mem_entry_t::addr, e820_t::addr, jinue_syscall_args_t::arg0, jinue_syscall_args_t::arg1, jinue_-syscall_args_t::arg2, jinue_syscall_args_t::arg3, boot_info, CONSOLE_DEFAULT_COLOR, console_printn(), console-_putc(), object_ref_t::cookie, boot_info_t::e820_entries, boot_info_t::e820_map, jinue_mem_map_t::entry, object_ref-

_t::flags, get_boot_info(), ipc_t::header, IPC_FLAG_NONE, IPC_FLAG_SYSTEM, ipc_get_proc_object(), ipc_object_-create(), ipc_receive(), ipc_reply(), ipc_send(), JINUE_EAGAIN, JINUE_EINVAL, JINUE_ENOSYS, JINUE_IPC_PRO-C, JINUE_IPC_SYSTEM, NULL, jinue_mem_map_t::num_entries, object_ref_t::object, OBJECT_REF_FLAG_OWNE-R, OBJECT_REF_FLAG_VALID, printk(), thread_t::process, process_get_descriptor(), process_unused_descriptor(), jinue_mem_entry_t::size, e820_t::size, SYSCALL_FUNCT_CONSOLE_PUTC, SYSCALL_FUNCT_CONSOLE_PUTS, SYSCALL_FUNCT_CREATE_IPC, SYSCALL_FUNCT_GET_PHYS_MEMORY, SYSCALL_FUNCT_GET_THREA-D_LOCAL_ADDR, SYSCALL_FUNCT_PROC_BASE, SYSCALL_FUNCT_RECEIVE, SYSCALL_FUNCT_REPLY, SYSCALL_FUNCT_SET_THREAD_LOCAL_ADDR, SYSCALL_FUNCT_SYSCALL_METHOD, SYSCALL_FUNCT-_SYSTEM_BASE, SYSCALL_FUNCT_THREAD_CREATE, SYSCALL_FUNCT_THREAD_YIELD, syscall_method, thread_create(), thread_yield_from(), jinue_mem_entry_t::type, and e820_t::type.

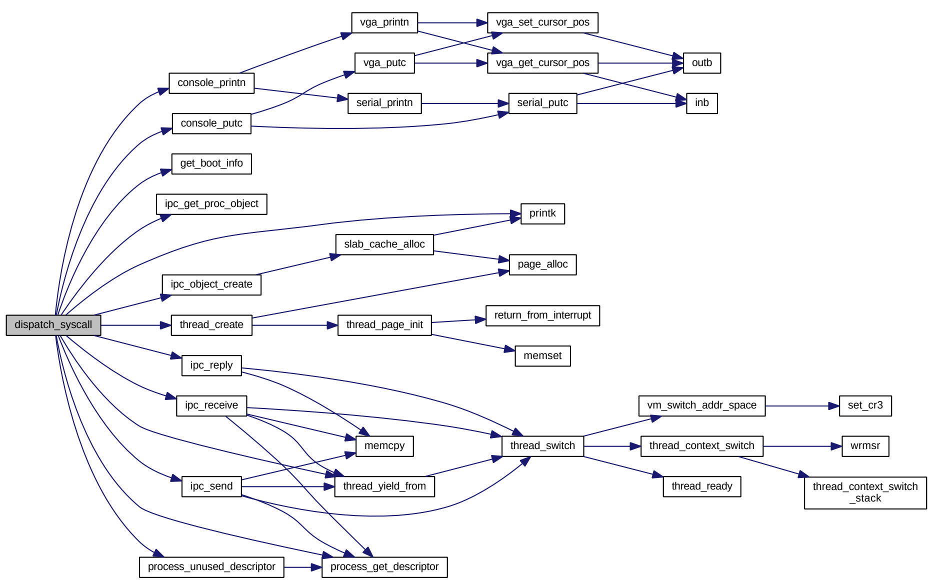
Referenced by dispatch_interrupt().

```
                                            {
    jinue_syscall_args_t *args = (jinue_syscall_args_t *)&trapframe->msg_arg0;

    uintptr_t function_number = args->arg0;

    if(function_number < SYSCALL_FUNCT_PROC_BASE) {
        /* microkernel system calls */
        switch(function_number) {

        case SYSCALL_FUNCT_SYSCALL_METHOD:
            syscall_args_set_return(args, syscall_method);
            break;

        case SYSCALL_FUNCT_CONSOLE_PUTC:
            console_putc(
                    (char)args->arg1,
                    CONSOLE_DEFAULT_COLOR);
            syscall_args_set_return(args, 0);
            break;

        case SYSCALL_FUNCT_CONSOLE_PUTS:
            console_printn(
                    (char *)args->arg2,
                    jinue_args_get_data_size(args),
                    CONSOLE_DEFAULT_COLOR);
            syscall_args_set_return(args, 0);
            break;

        case SYSCALL_FUNCT_THREAD_CREATE:
        {
            thread_t *thread = thread_create(
                    /* TODO use arg1 as an address space reference if specified */
                    get_current_thread()->process,
                    (addr_t)args->arg2,
                    (addr_t)args->arg3);

            if(thread == NULL) {
                syscall_args_set_error(args, JINUE_EAGAIN);
            }
            else {
                syscall_args_set_return(args, 0);
            }
        }
            break;

        case SYSCALL_FUNCT_THREAD_YIELD:
            thread_yield_from(
                    get_current_thread(),
                    false,          /* don't block */
                    args->arg1);    /* destroy (aka. exit) thread if true */
            syscall_args_set_return(args, 0);
            break;

        case SYSCALL_FUNCT_SET_THREAD_LOCAL_ADDR:
            thread_context_set_local_storage(
                    &get_current_thread()->thread_ctx,
                    (addr_t)args->arg1,
                    (size_t)args->arg2);
            syscall_args_set_return(args, 0);
            break;

        case SYSCALL_FUNCT_GET_THREAD_LOCAL_ADDR:
            syscall_args_set_return_ptr(
```

```
                    args,
                    thread_context_get_local_storage(
                            &get_current_thread()->thread_ctx));
            break;

        case SYSCALL_FUNCT_GET_PHYS_MEMORY:
        {
            unsigned int idx;

            size_t buffer_size = jinue_args_get_buffer_size(args);
            jinue_mem_map_t *map = (jinue_mem_map_t *)jinue_args_get_buffer_ptr(args);
            const boot_info_t *boot_info = get_boot_info();

            if(buffer_size < sizeof(jinue_mem_map_t) + boot_info->e820_entries * sizeof(jinue_mem_entry_t) ) {
                syscall_args_set_error(args, JINUE_EINVAL);
            }
            else {
                map->num_entries = boot_info->e820_entries;

                for(idx = 0; idx < map->num_entries; ++idx) {
                    map->entry[idx].addr = boot_info->e820_map[idx].addr;
                    map->entry[idx].size = boot_info->e820_map[idx].size;
                    map->entry[idx].type = boot_info->e820_map[idx].type;
                }

                syscall_args_set_return(args, 0);
            }
        }
            break;

        case SYSCALL_FUNCT_CREATE_IPC:
        {
            ipc_t *ipc;

            thread_t *thread = get_current_thread();

            int fd = process_unused_descriptor(thread->process);

            if(fd < 0) {
                syscall_args_set_error(args, JINUE_EAGAIN);
                break;
            }

            if(args->arg1 & JINUE_IPC_PROC) {
                ipc = ipc_get_proc_object();
            }
            else {
                int flags = IPC_FLAG_NONE;

                if(args->arg1 & JINUE_IPC_SYSTEM) {
                    flags |= IPC_FLAG_SYSTEM;
                }

                ipc = ipc_object_create(flags);

                if(ipc == NULL) {
                    syscall_args_set_error(args, JINUE_EAGAIN);
                    break;
                }
            }

            object_ref_t *ref = process_get_descriptor(thread->process, fd);

            object_addref(&ipc->header);

            ref->object = &ipc->header;
            ref->flags  = OBJECT_REF_FLAG_VALID | OBJECT_REF_FLAG_OWNER;
            ref->cookie = 0;

            syscall_args_set_return(args, fd);

        }
            break;
        case SYSCALL_FUNCT_RECEIVE:
            ipc_receive(args);
            break;

        case SYSCALL_FUNCT_REPLY:
            ipc_reply(args);
            break;
```

```

        default:
            printk("SYSCALL: function %u arg1=%u(0x%x) arg2=%u(0x%x) arg3=%u(0x%x)\n",
                function_number,
                args->arg1, args->arg1,
                args->arg2, args->arg2,
                args->arg3, args->arg3 );

            syscall_args_set_error(args, JINUE_ENOSYS);
        }
    }
    else if(function_number < SYSCALL_FUNCT_SYSTEM_BASE) {
        /* process manager system calls */
        printk("PROC SYSCALL: function %u arg1=%u(0x%x) arg2=%u(0x%x) arg3=%u(0x%x)\n",
                function_number,
                args->arg1, args->arg1,
                args->arg2, args->arg2,
                args->arg3, args->arg3 );

        syscall_args_set_error(args, JINUE_ENOSYS);
    }
    else {
        /* inter-process message */
        ipc_send(args);
    }
}
```

Here is the call graph for this function:

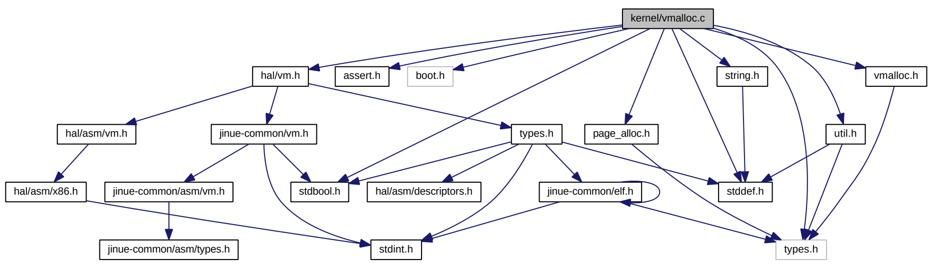


4.109 kernel/vmalloc.c File Reference

Virtual address space allocator.

```
#include <hal/vm.h>
#include <assert.h>
#include <boot.h>
#include <page_alloc.h>
#include <stdbool.h>
#include <stddef.h>
#include <string.h>
#include <types.h>
#include <util.h>
#include <vmalloc.h>
```

Include dependency graph for vmalloc.c:



Data Structures

- struct **vmalloc_t**
- struct **vmalloc_block_t**

Macros

- #define **VMALLOC_STACK_ENTRIES** (**PAGE_SIZE** / sizeof(**addr_t**))
- #define **VMALLOC_BLOCK_SIZE** (**VMALLOC_STACK_ENTRIES** ∗ **PAGE_SIZE**)

Typedefs

- typedef struct **vmalloc_block_t** **vmalloc_block_t**

Functions

- **addr_t vmalloc** (void)

  Allocate a page of virtual address space.
- void **vmfree** (**addr_t** page)

  Free a page of virtual address space.
- void **vmalloc_init** (**addr_t** start_addr, **addr_t** end_addr, **addr_t** preinit_limit, **boot_alloc_t** ∗boot_alloc)

  Basic initialization of the virtual memory allocator.
- **bool vmalloc_is_in_range** (**addr_t** page)

  Check whether the specified page is in the region managed by the allocator.

4.109.1 Detailed Description

Virtual address space allocator. Functions in this file are used to manage the allocation of pages in a virtual address space region.

The kernel address space has only one region where allocations are managed by this allocator, the so-called allocations region. A different allocator is used during initialization - see boot.c. That other allocator manages a different, smaller region of the address space, the so-called image region.

While this allocation manages a single region of the address space, it is structured in such a way that it can easily be used to manage multiple address space regions independently, with each region being represented by a **vmalloc_t** (p. 56) structure.

This allocator allocates pages one at a time. There is no way to allocate multiple contiguous pages after kernel initialization. (However, this is something the initialization-time allocator can do.

The allocator manages its address space region in identically-sized blocks of a few megabytes aligned on their size. Each block is represented by a **vmalloc_block_t** (p. 55) structure and has a stack to record available pages. The size of this stack is exactly one page, which is what determines the block size.

vm_block_t structures for an allocator are in an array that ensures the right vm_block_t structure can be found quickly during de-allocation. Non-depleted blocks are linked to a free list (a circular, doubly-linked list) that allows the allocator to quickly find a block with free pages during allocations.

Warning: the prev member (back pointer) of **vmalloc_block_t** (p. 55) is only meaningful while the block is linked to the free list. If there is a need to check whether the block is linked or not and vmalloc_stack_is_empty() isn't appropriate, check if the next member (not prev) is NULL.

Definition in file **vmalloc.c**.

4.109.2 Macro Definition Documentation

4.109.2.1 #define VMALLOC_BLOCK_SIZE (VMALLOC_STACK_ENTRIES ∗ PAGE_SIZE)

Definition at line 84 of file vmalloc.c.

4.109.2.2 #define VMALLOC_STACK_ENTRIES (PAGE_SIZE / sizeof(addr_t))

Definition at line 82 of file vmalloc.c.

4.109.3 Typedef Documentation

4.109.3.1 typedef struct vmalloc_block_t vmalloc_block_t

Definition at line 124 of file vmalloc.c.

4.109.4 Function Documentation

4.109.4.1 addr_t vmalloc (void)

Allocate a page of virtual address space.

Returns

address of allocated page or NULL if allocation failed

Definition at line 150 of file vmalloc.c.

Referenced by add_page_frame(), boot_page_alloc(), elf_setup_stack(), vm_clone_page_directory(), vm_destroy_-page_directory(), vm_pae_lookup_page_directory(), and vm_x86_lookup_page_directory().

```
150                     {
151     return vmalloc_from(&kernel_vmallocator);
152 }
```

4.109.4.2 void vmalloc_init (addr_t *start_addr*, addr_t *end_addr*, addr_t *preinit_limit*, boot_alloc_t ∗ *boot_alloc*)

Basic initialization of the virtual memory allocator.

This function initializes the allocator structure, and then initializes the first few blocks up to the limit set by the preinit_limit argument (more precisely, up to and including the block that contains preinit_limit - 1). TODO mention how to initialize the rest once this is implemented.

Parameters

start_addr	the start address of the region managed by the allocator
end_addr	the end address of the region managed by the allocator
preinit_limit	the limit address for preinitialized blocks
boot_alloc	the initialization-time page allocator structure

Definition at line 178 of file vmalloc.c.

Referenced by vm_boot_postinit().

```
182                                     {
183
184     init_allocator(
185             &kernel_vmallocator,
186             start_addr,
187             end_addr,
188             preinit_limit,
189             boot_alloc);
190 }
```

4.109.4.3 bool vmalloc_is_in_range (addr_t *page*)

Check whether the specified page is in the region managed by the allocator.

Parameters

page	the address of the page

Returns

true if it is in the region, false otherwise

Definition at line 199 of file vmalloc.c.

Referenced by remove_page_frame().

```
199                                      {
200     return addr_is_in_initialized_range(&kernel_vmallocator, page);
201 }
```

4.109.4.4 void vmfree (addr_t *page*)

Free a page of virtual address space.

Parameters

page	the address of the page to free

Definition at line 160 of file vmalloc.c.

Referenced by elf_setup_stack(), and remove_page_frame().

```
160                          {
161     vmfree_to(&kernel_vmallocator, page);
162 }
```

Index