Jinue

Generated by Doxygen 1.8.5

Fri Jan 4 2019 22:25:11

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:

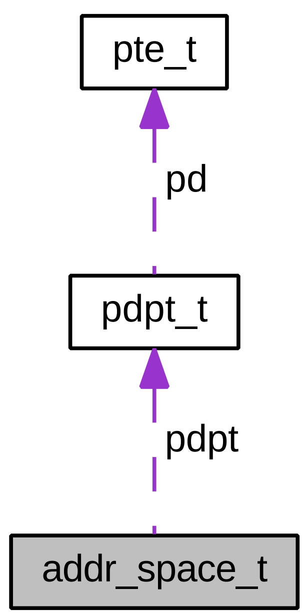
Chapter 3

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 {
    pfaddr_t pd
    pdpt_t * pdpt
  } **top_level**

3.1.1 Detailed Description

Definition at line 72 of file types.h.

3.1.2 Field Documentation

3.1.2.1 uint32_t addr_space_t::cr3

Definition at line 73 of file types.h.

Referenced by vm_switch_addr_space(), and vm_x86_create_initial_addr_space().

3.1.2.2 pfaddr_t addr_space_t::pd

Definition at line 75 of file types.h.

Referenced by vm_x86_create_initial_addr_space().

3.1.2.3 pdpt_t* addr_space_t::pdpt

Definition at line 76 of file types.h.

3.1.2.4 union { ... } addr_space_t::top_level

Referenced by vm_x86_create_initial_addr_space().

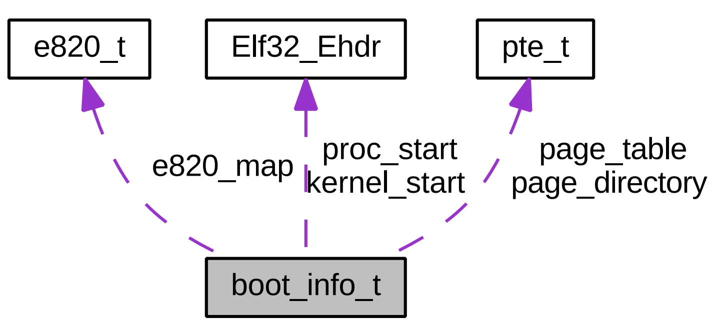
The documentation for this struct was generated from the following file:

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

3.2 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 e820_entries**
- **e820_t** ∗ **e820_map**
- void ∗ **boot_heap**
- void ∗ **boot_end**
- **pte_t** ∗ **page_table**
- **pte_t** ∗ **page_directory**
- **uint32_t setup_signature**

3.2.1 Detailed Description

Definition at line 94 of file types.h.

3.2.2 Field Documentation

3.2.2.1 void∗ boot_info_t::boot_end

Definition at line 104 of file types.h.

Referenced by boot_info_dump(), and hal_init().

3.2.2.2 void∗ boot_info_t::boot_heap

Definition at line 103 of file types.h.

Referenced by boot_info_dump(), and hal_init().

3.2.2.3 uint32_t boot_info_t::e820_entries

Definition at line 101 of file types.h.

Referenced by boot_info_dump(), bootmem_init(), and e820_dump().

3.2.2.4 e820_t∗ boot_info_t::e820_map

Definition at line 102 of file types.h.

Referenced by boot_info_dump(), bootmem_init(), and e820_dump().

3.2.2.5 void∗ boot_info_t::image_start

Definition at line 99 of file types.h.

Referenced by boot_info_dump(), bootmem_init(), hal_init(), and vm_boot_init().

3.2.2.6 void∗ boot_info_t::image_top

Definition at line 100 of file types.h.

Referenced by boot_info_dump().

3.2.2.7 uint32_t boot_info_t::kernel_size

Definition at line 96 of file types.h.

Referenced by boot_info_dump(), and hal_init().

3.2.2.8 Elf32_Ehdr∗ boot_info_t::kernel_start

Definition at line 95 of file types.h.

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

3.2.2.9 pte_t∗ boot_info_t::page_directory

Definition at line 106 of file types.h.

Referenced by boot_info_dump().

3.2.2.10 pte_t∗ boot_info_t::page_table

Definition at line 105 of file types.h.

Referenced by boot_info_dump().

3.2.2.11 uint32_t boot_info_t::proc_size

Definition at line 98 of file types.h.

Referenced by boot_info_dump().

3.2.2.12 Elf32_Ehdr∗ boot_info_t::proc_start

Definition at line 97 of file types.h.

Referenced by boot_info_dump().

3.2.2.13 uint32_t boot_info_t::setup_signature

Definition at line 107 of file types.h.

Referenced by boot_info_check(), and boot_info_dump().

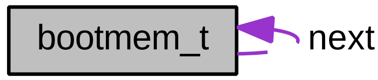
The documentation for this struct was generated from the following file:

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

3.3 bootmem_t Struct Reference

`#include <hal/bootmem.h>`

Collaboration diagram for bootmem_t:



Data Fields

- struct **bootmem_t** ∗ **next**
- **pfaddr_t addr**
- **uint32_t count**

3.3.1 Detailed Description

Definition at line 38 of file bootmem.h.

3.3.2 Field Documentation

3.3.2.1 pfaddr_t bootmem_t::addr

Definition at line 40 of file bootmem.h.

Referenced by apply_mem_hole(), bootmem_init(), dispatch_syscall(), and new_ram_map_entry().

3.3.2.2 uint32_t bootmem_t::count

Definition at line 41 of file bootmem.h.

Referenced by apply_mem_hole(), bootmem_init(), dispatch_syscall(), and new_ram_map_entry().

3.3.2.3 struct bootmem_t∗ bootmem_t::next

Definition at line 39 of file bootmem.h.

Referenced by apply_mem_hole(), bootmem_get_block(), bootmem_init(), and new_ram_map_entry().

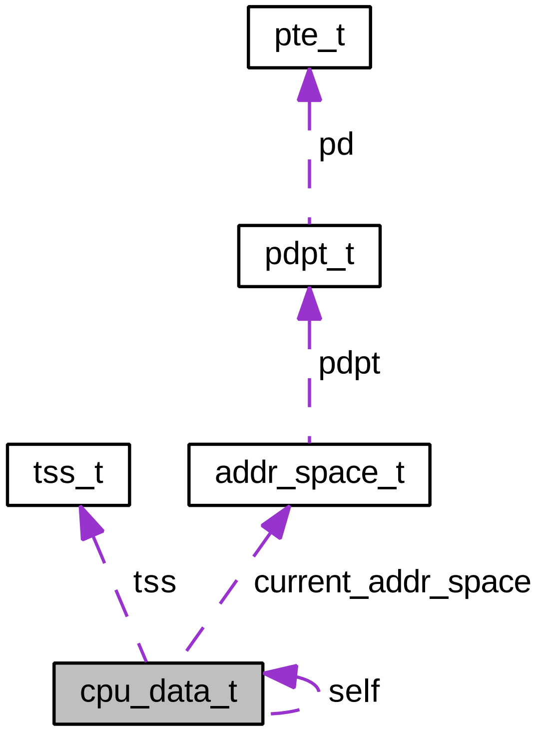
The documentation for this struct was generated from the following file:

- include/hal/**bootmem.h**

3.4 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**

3.4.1 Detailed Description

Definition at line 176 of file types.h.

3.4.2 Field Documentation

3.4.2.1 addr_space_t∗ cpu_data_t::current_addr_space

Definition at line 183 of file types.h.

Referenced by cpu_init_data().

3.4.2.2 seg_descriptor_t cpu_data_t::gdt[GDT_LENGTH]

Definition at line 177 of file types.h.

Referenced by cpu_init_data(), and hal_init().

3.4.2.3 struct cpu_data_t* cpu_data_t::self

Definition at line 182 of file types.h.

Referenced by cpu_init_data().

3.4.2.4 tss_t cpu_data_t::tss

Definition at line 181 of file types.h.

Referenced by cpu_init_data().

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

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

3.5 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**

3.5.1 Detailed Description

Definition at line 97 of file cpu.h.

3.5.2 Field Documentation

3.5.2.1 unsigned int cpu_info_t::dcache_alignment

Definition at line 98 of file cpu.h.

Referenced by cpu_detect_features(), and slab_cache_create().

3.5.2.2 int cpu_info_t::family

Definition at line 101 of file cpu.h.

Referenced by cpu_detect_features().

3.5.2.3 uint32_t cpu_info_t::features

Definition at line 99 of file cpu.h.

Referenced by cpu_detect_features().

3.5.2.4 int cpu_info_t::model

Definition at line 102 of file cpu.h.

Referenced by cpu_detect_features().

3.5.2.5 int cpu_info_t::stepping

Definition at line 103 of file cpu.h.

Referenced by cpu_detect_features().

3.5.2.6 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**

3.6 e820_t Struct Reference

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

Data Fields

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

3.6.1 Detailed Description

Definition at line 86 of file types.h.

3.6.2 Field Documentation

3.6.2.1 e820_addr_t e820_t::addr

Definition at line 87 of file types.h.

Referenced by bootmem_init(), and e820_dump().

3.6.2.2 e820_size_t e820_t::size

Definition at line 88 of file types.h.

Referenced by bootmem_init(), e820_dump(), and e820_is_valid().

3.6.2.3 e820_type_t e820_t::type

Definition at line 89 of file types.h.

Referenced by e820_dump(), and e820_is_available().

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

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

3.7 Elf32_auxv_t Struct Reference

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

Data Fields

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

3.7.1 Detailed Description

Definition at line 308 of file elf.h.

3.7.2 Field Documentation

3.7.2.1 int Elf32_auxv_t::a_type

Definition at line 309 of file elf.h.

Referenced by elf_setup_stack().

3.7.2.2 union { ... } Elf32_auxv_t::a_un

Referenced by elf_setup_stack().

3.7.2.3 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**

3.8 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**

3.8.1 Detailed Description

Definition at line 258 of file elf.h.

3.8.2 Field Documentation

3.8.2.1 Elf32_Half Elf32_Ehdr::e_ehsize

Definition at line 267 of file elf.h.

3.8.2.2 Elf32_Addr Elf32_Ehdr::e_entry

Definition at line 263 of file elf.h.

Referenced by elf_check(), and elf_load().

3.8.2.3 Elf32_Word Elf32_Ehdr::e_flags

Definition at line 266 of file elf.h.

Referenced by elf_check().

3.8.2.4 unsigned char Elf32_Ehdr::e_ident[EI_NIDENT]

Definition at line 259 of file elf.h.

Referenced by elf_check().

3.8.2.5 Elf32_Half Elf32_Ehdr::e_machine

Definition at line 261 of file elf.h.

Referenced by elf_check().

3.8.2.6 Elf32_Half Elf32_Ehdr::e_phentsize

Definition at line 268 of file elf.h.

Referenced by elf_check(), and elf_load().

3.8.2.7 Elf32_Half Elf32_Ehdr::e_phnum

Definition at line 269 of file elf.h.

Referenced by elf_check(), and elf_load().

3.8.2.8 Elf32_Off Elf32_Ehdr::e_phoff

Definition at line 264 of file elf.h.

Referenced by elf_check(), and elf_load().

3.8.2.9 Elf32_Half Elf32_Ehdr::e_shentsize

Definition at line 270 of file elf.h.

3.8.2.10 Elf32_Half Elf32_Ehdr::e_shnum

Definition at line 271 of file elf.h.

Referenced by elf_lookup_symbol().

3.8.2.11 Elf32_Off Elf32_Ehdr::e_shoff

Definition at line 265 of file elf.h.

3.8.2.12 Elf32_Half Elf32_Ehdr::e_shstrndx

Definition at line 272 of file elf.h.

3.8.2.13 Elf32_Half Elf32_Ehdr::e_type

Definition at line 260 of file elf.h.

Referenced by elf_check().

3.8.2.14 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**

3.9 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**

3.9.1 Detailed Description

Definition at line 275 of file elf.h.

3.9.2 Field Documentation

3.9.2.1 Elf32_Word Elf32_Phdr::p_align

Definition at line 283 of file elf.h.

3.9.2.2 Elf32_Word Elf32_Phdr::p_filesz

Definition at line 280 of file elf.h.

Referenced by elf_load().

3.9.2.3 Elf32_Word Elf32_Phdr::p_flags

Definition at line 282 of file elf.h.

3.9.2.4 Elf32_Word Elf32_Phdr::p_memsz

Definition at line 281 of file elf.h.

Referenced by elf_load().

3.9.2.5 Elf32_Off Elf32_Phdr::p_offset

Definition at line 277 of file elf.h.

3.9.2.6 Elf32_Addr Elf32_Phdr::p_paddr

Definition at line 279 of file elf.h.

3.9.2.7 Elf32_Word Elf32_Phdr::p_type

Definition at line 276 of file elf.h.

3.9.2.8 Elf32_Addr Elf32_Phdr::p_vaddr

Definition at line 278 of file elf.h.

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

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

3.10 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**

3.10.1 Detailed Description

Definition at line 286 of file elf.h.

3.10.2 Field Documentation

3.10.2.1 Elf32_Addr Elf32_Shdr::sh_addr

Definition at line 290 of file elf.h.

3.10.2.2 Elf32_Word Elf32_Shdr::sh_addralign

Definition at line 295 of file elf.h.

3.10.2.3 Elf32_Word Elf32_Shdr::sh_entsize

Definition at line 296 of file elf.h.

Referenced by elf_lookup_symbol().

3.10.2.4 Elf32_Word Elf32_Shdr::sh_flags

Definition at line 289 of file elf.h.

3.10.2.5 Elf32_Word Elf32_Shdr::sh_info

Definition at line 294 of file elf.h.

3.10.2.6 Elf32_Word Elf32_Shdr::sh_link

Definition at line 293 of file elf.h.

Referenced by elf_lookup_symbol().

3.10.2.7 Elf32_Word Elf32_Shdr::sh_name

Definition at line 287 of file elf.h.

3.10.2.8 Elf32_Off Elf32_Shdr::sh_offset

Definition at line 291 of file elf.h.

Referenced by elf_lookup_symbol().

3.10.2.9 Elf32_Word Elf32_Shdr::sh_size

Definition at line 292 of file elf.h.

Referenced by elf_lookup_symbol().

3.10.2.10 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**

3.11 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**

3.11.1 Detailed Description

Definition at line 299 of file elf.h.

3.11.2 Field Documentation

3.11.2.1 unsigned char Elf32_Sym::st_info

Definition at line 303 of file elf.h.

Referenced by elf_lookup_symbol().

3.11.2.2 Elf32_Word Elf32_Sym::st_name

Definition at line 300 of file elf.h.

Referenced by elf_lookup_symbol().

3.11.2.3 unsigned char Elf32_Sym::st_other

Definition at line 304 of file elf.h.

3.11.2.4 Elf32_Half Elf32_Sym::st_shndx

Definition at line 305 of file elf.h.

3.11.2.5 Elf32_Word Elf32_Sym::st_size

Definition at line 302 of file elf.h.

Referenced by elf_lookup_symbol().

3.11.2.6 Elf32_Addr Elf32_Sym::st_value

Definition at line 301 of file elf.h.

Referenced by elf_lookup_symbol().

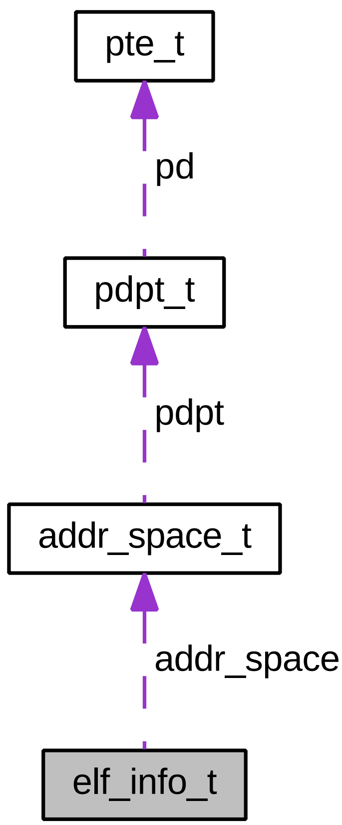
The documentation for this struct was generated from the following file:

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

3.12 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**

3.12.1 Detailed Description

Definition at line 39 of file elf.h.

3.12.2 Field Documentation

3.12.2.1 addr_space_t* elf_info_t::addr_space

Definition at line 45 of file elf.h.

Referenced by elf_load(), and elf_setup_stack().

3.12.2.2 addr_t elf_info_t::at_phdr

Definition at line 42 of file elf.h.

Referenced by elf_load(), and elf_setup_stack().

3.12.2.3 int elf_info_t::at_phent

Definition at line 43 of file elf.h.

Referenced by elf_load(), and elf_setup_stack().

3.12.2.4 int elf_info_t::at_phnum

Definition at line 44 of file elf.h.

Referenced by elf_load(), and elf_setup_stack().

3.12.2.5 addr_t elf_info_t::entry

Definition at line 40 of file elf.h.

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

3.12.2.6 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**

3.13 elf_symbol_t Struct Reference

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

Data Fields

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

3.13.1 Detailed Description

Definition at line 48 of file elf.h.

3.13.2 Field Documentation

3.13.2.1 Elf32_Addr elf_symbol_t::addr

Definition at line 49 of file elf.h.

Referenced by dump_call_stack(), and elf_lookup_symbol().

3.13.2.2 const char* elf_symbol_t::name

Definition at line 50 of file elf.h.

Referenced by dump_call_stack(), and elf_lookup_symbol().

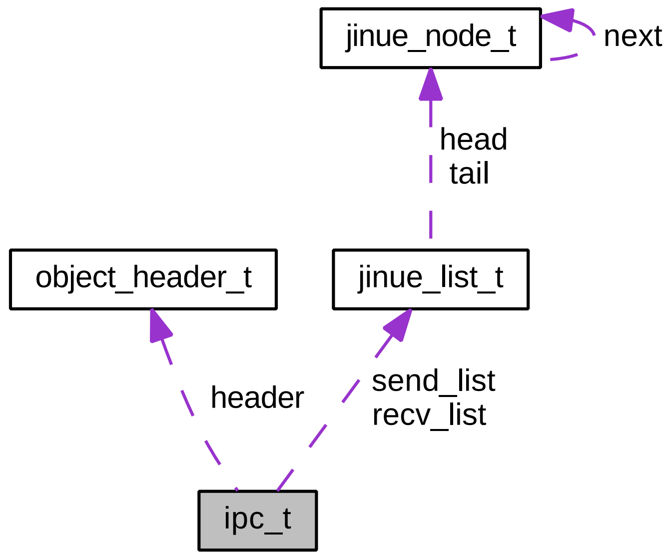
The documentation for this struct was generated from the following file:

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

3.14 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**

3.14.1 Detailed Description

Definition at line 87 of file types.h.

3.14.2 Field Documentation

3.14.2.1 object_header_t ipc_t::header

Definition at line 88 of file types.h.

Referenced by dispatch_syscall(), and ipc_object_create().

3.14.2.2 jinue_list_t ipc_t::recv_list

Definition at line 90 of file types.h.

Referenced by ipc_receive(), and ipc_send().

3.14.2.3 jinue_list_t ipc_t::send_list

Definition at line 89 of file types.h.

Referenced by ipc_receive(), and ipc_send().

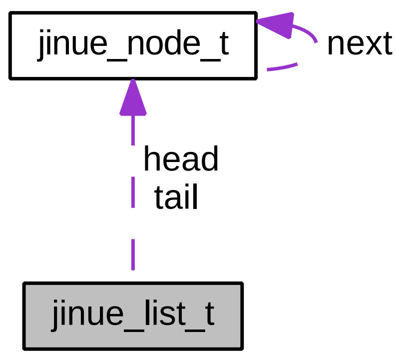
The documentation for this struct was generated from the following file:

- include/**types.h**

3.15 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**

3.15.1 Detailed Description

Definition at line 55 of file list.h.

3.15.2 Field Documentation

3.15.2.1 jinue_node_t∗ jinue_list_t::head

Definition at line 56 of file list.h.

3.15.2.2 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**

3.16 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**

3.16.1 Detailed Description

Definition at line 38 of file ipc.h.

3.16.2 Field Documentation

3.16.2.1 size_t jinue_message_t::buffer_size

Definition at line 41 of file ipc.h.

3.16.2.2 uintptr_t jinue_message_t::cookie

Definition at line 40 of file ipc.h.

3.16.2.3 size_t jinue_message_t::data_size

Definition at line 42 of file ipc.h.

3.16.2.4 size_t jinue_message_t::desc_n

Definition at line 43 of file ipc.h.

3.16.2.5 uintptr_t jinue_message_t::function

Definition at line 39 of file ipc.h.

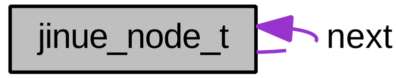
The documentation for this struct was generated from the following file:

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

3.17 jinue_node_t Struct Reference

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

Collaboration diagram for jinue_node_t:



Data Fields

- struct **jinue_node_t** ∗ **next**

3.17.1 Detailed Description

Definition at line 38 of file list.h.

3.17.2 Field Documentation

3.17.2.1 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**

3.18 jinue_reply_t Struct Reference

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

Data Fields

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

3.18.1 Detailed Description

Definition at line 46 of file ipc.h.

3.18.2 Field Documentation

3.18.2.1 size_t jinue_reply_t::data_size

Definition at line 47 of file ipc.h.

3.18.2.2 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**

3.19 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**

3.19.1 Detailed Description

Definition at line 39 of file syscall.h.

3.19.2 Field Documentation

3.19.2.1 uintptr_t jinue_syscall_args_t::arg0

Definition at line 40 of file syscall.h.

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

3.19.2.2 uintptr_t jinue_syscall_args_t::arg1

Definition at line 41 of file syscall.h.

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

3.19.2.3 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().

3.19.2.4 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**

3.20 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**

3.20.1 Detailed Description

Definition at line 214 of file types.h.

3.20.2 Field Documentation

3.20.2.1 uint32_t kernel_context_t::ebp

Definition at line 218 of file types.h.

3.20.2.2 uint32_t kernel_context_t::ebx

Definition at line 217 of file types.h.

3.20.2.3 uint32_t kernel_context_t::edi

Definition at line 215 of file types.h.

3.20.2.4 uint32_t kernel_context_t::eip

Definition at line 219 of file types.h.

Referenced by thread_page_create().

3.20.2.5 uint32_t kernel_context_t::esi

Definition at line 216 of file types.h.

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

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

3.21 memory_block_t Struct Reference

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

Data Fields

- **pfaddr_t addr**
- **uint32_t count**

3.21.1 Detailed Description

Definition at line 42 of file pfalloc.h.

3.21.2 Field Documentation

3.21.2.1 pfaddr_t memory_block_t::addr

Definition at line 43 of file pfalloc.h.

Referenced by dispatch_syscall().

3.21.2.2 uint32_t memory_block_t::count

Definition at line 44 of file pfalloc.h.

Referenced by dispatch_syscall().

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

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

3.22 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**

3.22.1 Detailed Description

Definition at line 65 of file types.h.

3.22.2 Field Documentation

3.22.2.1 size_t message_info_t::buffer_size

Definition at line 68 of file types.h.

Referenced by ipc_reply(), and ipc_send().

3.22.2.2 uintptr_t message_info_t::cookie

Definition at line 67 of file types.h.

Referenced by ipc_send().

3.22.2.3 size_t message_info_t::data_size

Definition at line 69 of file types.h.

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

3.22.2.4 size_t message_info_t::desc_n

Definition at line 70 of file types.h.

Referenced by ipc_reply(), and ipc_send().

3.22.2.5 uintptr_t message_info_t::function

Definition at line 66 of file types.h.

Referenced by ipc_send().

3.22.2.6 size_t message_info_t::total_size

Definition at line 71 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**

3.23 object_header_t Struct Reference

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

Data Fields

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

3.23.1 Detailed Description

Definition at line 45 of file types.h.

3.23.2 Field Documentation

3.23.2.1 int object_header_t::flags

Definition at line 48 of file types.h.

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

3.23.2.2 int object_header_t::ref_count

Definition at line 47 of file types.h.

3.23.2.3 int object_header_t::type

Definition at line 46 of file types.h.

Referenced by ipc_receive(), and ipc_send().

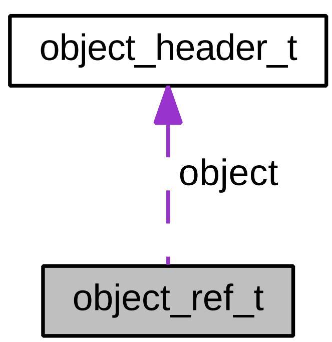
The documentation for this struct was generated from the following file:

- include/**types.h**

3.24 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**

3.24.1 Detailed Description

Definition at line 51 of file types.h.

3.24.2 Field Documentation

3.24.2.1 uintptr_t object_ref_t::cookie

Definition at line 54 of file types.h.

Referenced by dispatch_syscall().

3.24.2.2 uintptr_t object_ref_t::flags

Definition at line 53 of file types.h.

Referenced by dispatch_syscall().

3.24.2.3 object_header_t* object_ref_t::object

Definition at line 52 of file types.h.

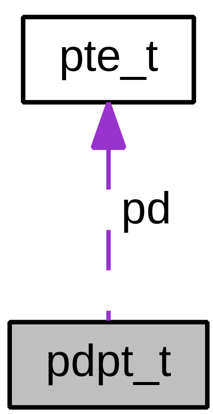
Referenced by dispatch_syscall().

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

- include/**types.h**

3.25 pdpt_t Struct Reference

Collaboration diagram for pdpt_t:



Data Fields

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

3.25.1 Detailed Description

Definition at line 57 of file vm_pae.c.

3.25.2 Field Documentation

3.25.2.1 pte_t pdpt_t::pd[PDPT_ENTRIES]

Definition at line 58 of file vm_pae.c.

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

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

3.26 pfcache_t Struct Reference

`#include <pfalloc.h>`

Data Fields

- **pfaddr_t ∗ ptr**
- **uint32_t count**

3.26.1 Detailed Description

Definition at line 39 of file pfalloc.h.

3.26.2 Field Documentation

3.26.2.1 uint32_t pfcache_t::count

Definition at line 41 of file pfalloc.h.

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

3.26.2.2 pfaddr_t∗ pfcache_t::ptr

Definition at line 40 of file pfalloc.h.

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

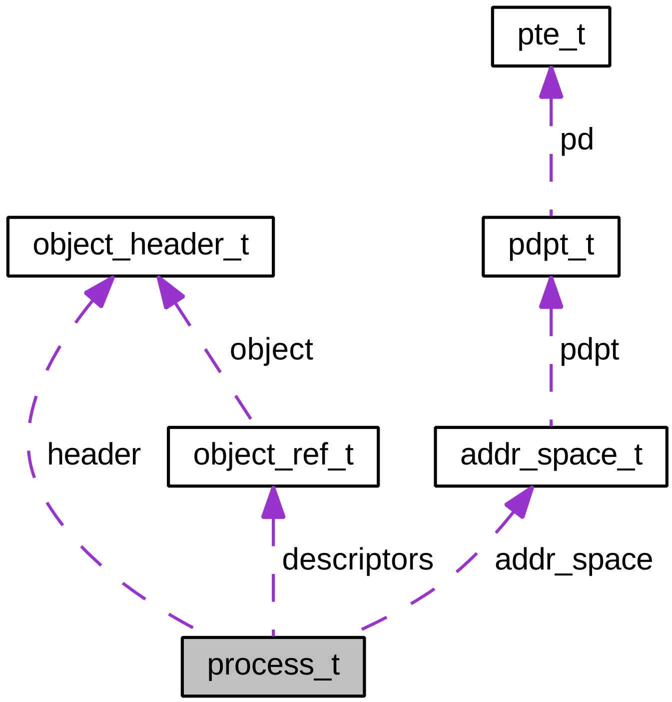
The documentation for this struct was generated from the following file:

- include/**pfalloc.h**

3.27 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**]

3.27.1 Detailed Description

Definition at line 59 of file types.h.

3.27.2 Field Documentation

3.27.2.1 addr_space_t process_t::addr_space

Definition at line 61 of file types.h.

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

3.27.2.2 object_ref_t process_t::descriptors[PROCESS_MAX_DESCRIPTORS]

Definition at line 62 of file types.h.

Referenced by process_create(), and process_get_descriptor().

3.27.2.3 object_header_t process_t::header

Definition at line 60 of file types.h.

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

- include/**types.h**

3.28 pseudo_descriptor_t Struct Reference

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

Data Fields

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

3.28.1 Detailed Description

Definition at line 114 of file types.h.

3.28.2 Field Documentation

3.28.2.1 addr_t pseudo_descriptor_t::addr

Definition at line 117 of file types.h.

Referenced by hal_init().

3.28.2.2 uint16_t pseudo_descriptor_t::limit

Definition at line 116 of file types.h.

Referenced by hal_init().

3.28.2.3 uint16_t pseudo_descriptor_t::padding

Definition at line 115 of file types.h.

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

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

3.29 pte_t Struct Reference

Data Fields

- **uint32_t entry**

- **uint64_t entry**

3.29.1 Detailed Description

Definition at line 636 of file vm.c.

3.29.2 Field Documentation

3.29.2.1 uint64_t pte_t::entry

Definition at line 54 of file vm_pae.c.

3.29.2.2 uint32_t pte_t::entry

Definition at line 637 of file vm.c.

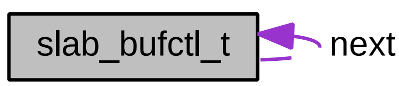
The documentation for this struct was generated from the following files:

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

3.30 slab_bufctl_t Struct Reference

`#include <slab.h>`

Collaboration diagram for slab_bufctl_t:



Data Fields

- struct **slab_bufctl_t** ∗ **next**

3.30.1 Detailed Description

Definition at line 86 of file slab.h.

3.30.2 Field Documentation

3.30.2.1 struct slab_bufctl_t∗ slab_bufctl_t::next

Definition at line 87 of file slab.h.

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

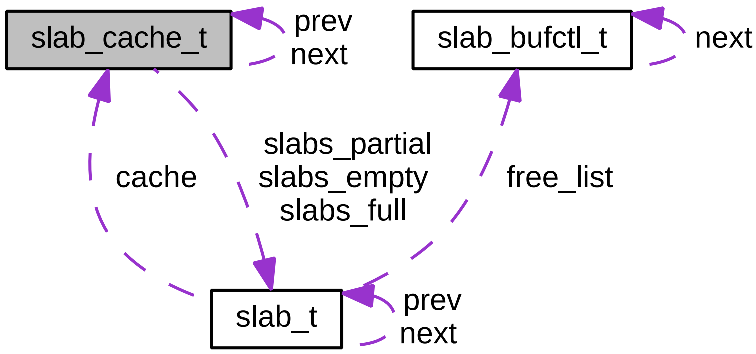
The documentation for this struct was generated from the following file:

- include/**slab.h**

3.31 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**
- struct **slab_cache_t** ∗ **prev**
- struct **slab_cache_t** ∗ **next**
- int **flags**

3.31.1 Detailed Description

Definition at line 64 of file slab.h.

3.31.2 Field Documentation

3.31.2.1 size_t slab_cache_t::alignment

Definition at line 71 of file slab.h.

Referenced by slab_cache_create(), and slab_cache_grow().

3.31.2.2 size_t slab_cache_t::alloc_size

Definition at line 70 of file slab.h.

Referenced by slab_cache_create(), and slab_cache_grow().

3.31.2.3 size_t slab_cache_t::bufctl_offset

Definition at line 72 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_create(), slab_cache_free(), and slab_cache_grow().

3.31.2.4 slab_ctor_t slab_cache_t::ctor

Definition at line 76 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_create(), and slab_cache_grow().

3.31.2.5 slab_ctor_t slab_cache_t::dtor

Definition at line 77 of file slab.h.

Referenced by slab_cache_create(), and slab_cache_free().

3.31.2.6 unsigned int slab_cache_t::empty_count

Definition at line 68 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_create(), slab_cache_destroy(), slab_cache_free(), slab_cache_grow(), and slab_cache_reap().

3.31.2.7 int slab_cache_t::flags

Definition at line 81 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_create(), slab_cache_free(), and slab_cache_grow().

3.31.2.8 size_t slab_cache_t::max_colour

Definition at line 74 of file slab.h.

Referenced by slab_cache_create(), and slab_cache_grow().

3.31.2.9 char* slab_cache_t::name

Definition at line 78 of file slab.h.

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

3.31.2.10 struct slab_cache_t* slab_cache_t::next

Definition at line 80 of file slab.h.

Referenced by slab_cache_create(), and slab_cache_destroy().

3.31.2.11 size_t slab_cache_t::next_colour

Definition at line 73 of file slab.h.

Referenced by slab_cache_create(), and slab_cache_grow().

3.31.2.12 size_t slab_cache_t::obj_size

Definition at line 69 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_create(), slab_cache_free(), and slab_cache_grow().

3.31.2.13 struct slab_cache_t* slab_cache_t::prev

Definition at line 79 of file slab.h.

Referenced by slab_cache_create(), and slab_cache_destroy().

3.31.2.14 struct slab_t* slab_cache_t::slabs_empty

Definition at line 65 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_create(), slab_cache_destroy(), slab_cache_free(), slab_cache_grow(), and slab_cache_reap().

3.31.2.15 struct slab_t* slab_cache_t::slabs_full

Definition at line 67 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_create(), slab_cache_destroy(), and slab_cache_free().

3.31.2.16 struct slab_t* slab_cache_t::slabs_partial

Definition at line 66 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_create(), slab_cache_destroy(), and slab_cache_free().

3.31.2.17 unsigned int slab_cache_t::working_set

Definition at line 75 of file slab.h.

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

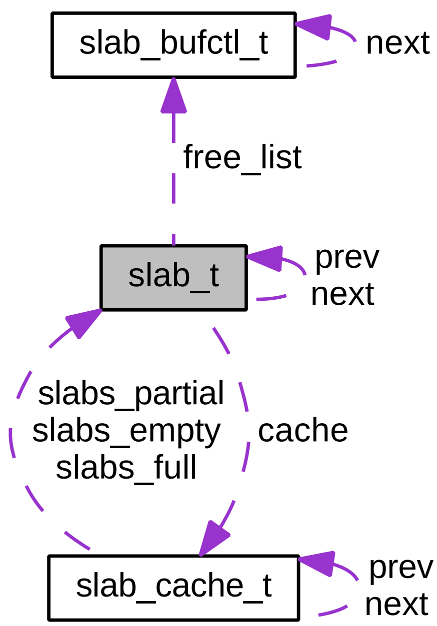
The documentation for this struct was generated from the following file:

- include/**slab.h**

3.32 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**

3.32.1 Detailed Description

Definition at line 92 of file slab.h.

3.32.2 Field Documentation

3.32.2.1 slab_cache_t* slab_t::cache

Definition at line 96 of file slab.h.

Referenced by slab_cache_free(), and slab_cache_grow().

3.32.2.2 size_t slab_t::colour

Definition at line 99 of file slab.h.

Referenced by slab_cache_grow().

3.32.2.3 slab_bufctl_t* slab_t::free_list

Definition at line 100 of file slab.h.

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

3.32.2.4 struct slab_t* slab_t::next

Definition at line 94 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_destroy(), slab_cache_free(), slab_cache_grow(), and slab_cache_-reap().

3.32.2.5 unsigned int slab_t::obj_count

Definition at line 98 of file slab.h.

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

3.32.2.6 struct slab_t* slab_t::prev

Definition at line 93 of file slab.h.

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

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

- include/**slab.h**

3.33 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**

3.33.1 Detailed Description

Definition at line 64 of file types.h.

3.33.2 Field Documentation

3.33.2.1 addr_t thread_context_t::local_storage_addr

Definition at line 68 of file types.h.

Referenced by thread_page_create().

3.33.2.2 size_t thread_context_t::local_storage_size

Definition at line 69 of file types.h.

3.33.2.3 addr_t thread_context_t::saved_stack_pointer

Definition at line 67 of file types.h.

Referenced by thread_page_create().

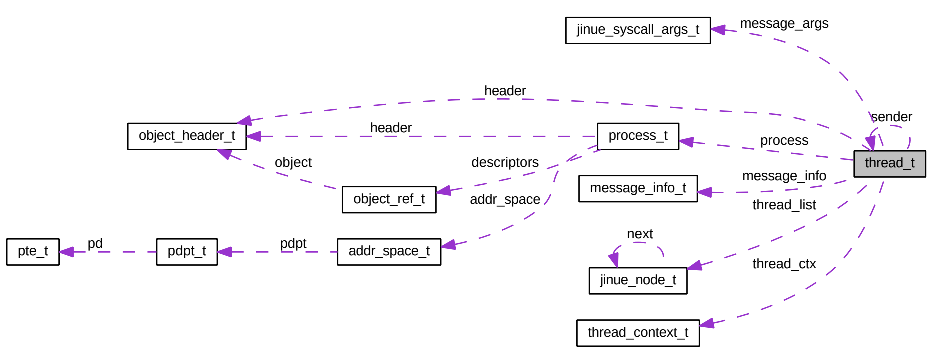
The documentation for this struct was generated from the following file:

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

3.34 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**]

3.34.1 Detailed Description

Definition at line 74 of file types.h.

3.34.2 Field Documentation

3.34.2.1 object_header_t thread_t::header

Definition at line 75 of file types.h.

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

3.34.2.2 jinue_syscall_args_t∗ thread_t::message_args

Definition at line 80 of file types.h.

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

3.34.2.3 char thread_t::message_buffer[JINUE_SEND_MAX_SIZE]

Definition at line 82 of file types.h.

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

3.34.2.4 message_info_t thread_t::message_info

Definition at line 81 of file types.h.

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

3.34.2.5 process_t∗ thread_t::process

Definition at line 78 of file types.h.

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

3.34.2.6 struct thread_t∗ thread_t::sender

Definition at line 79 of file types.h.

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

3.34.2.7 thread_context_t thread_t::thread_ctx

Definition at line 76 of file types.h.

Referenced by thread_switch().

3.34.2.8 jinue_node_t thread_t::thread_list

Definition at line 77 of file types.h.

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

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

- include/**types.h**

3.35 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**

3.35.1 Detailed Description

Definition at line 188 of file types.h.

3.35.2 Field Documentation

3.35.2.1 uint32_t trapframe_t::cs

Definition at line 208 of file types.h.

Referenced by thread_page_create().

3.35.2.2 uint32_t trapframe_t::ds

Definition at line 200 of file types.h.

Referenced by thread_page_create().

3.35.2.3 uint32_t trapframe_t::eax

Definition at line 191 of file types.h.

3.35.2.4 uint32_t trapframe_t::ebp

Definition at line 206 of file types.h.

3.35.2.5 uint32_t trapframe_t::ebx

Definition at line 193 of file types.h.

3.35.2.6 uint32_t trapframe_t::ecx

Definition at line 199 of file types.h.

3.35.2.7 uint32_t trapframe_t::edi

Definition at line 197 of file types.h.

3.35.2.8 uint32_t trapframe_t::edx

Definition at line 198 of file types.h.

3.35.2.9 uint32_t trapframe_t::eflags

Definition at line 209 of file types.h.

Referenced by thread_page_create().

3.35.2.10 uint32_t trapframe_t::eip

Definition at line 207 of file types.h.

Referenced by dispatch_interrupt(), and thread_page_create().

3.35.2.11 uint32_t trapframe_t::errcode

Definition at line 204 of file types.h.

Referenced by dispatch_interrupt().

3.35.2.12 uint32_t trapframe_t::es

Definition at line 201 of file types.h.

Referenced by thread_page_create().

3.35.2.13 uint32_t trapframe_t::esi

Definition at line 195 of file types.h.

3.35.2.14 uint32_t trapframe_t::esp

Definition at line 210 of file types.h.

Referenced by thread_page_create().

3.35.2.15 uint32_t trapframe_t::fs

Definition at line 202 of file types.h.

Referenced by thread_page_create().

3.35.2.16 uint32_t trapframe_t::gs

Definition at line 203 of file types.h.

Referenced by thread_page_create().

3.35.2.17 uint32_t trapframe_t::ivt

Definition at line 205 of file types.h.

Referenced by dispatch_interrupt().

3.35.2.18 uint32_t trapframe_t::ss

Definition at line 211 of file types.h.

Referenced by thread_page_create().

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

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

3.36 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**

3.36.1 Detailed Description

Definition at line 120 of file types.h.

3.36.2 Field Documentation

3.36.2.1 uint32_t tss_t::cr3

Definition at line 136 of file types.h.

3.36.2.2 uint16_t tss_t::cs

Definition at line 160 of file types.h.

3.36.2.3 uint16_t tss_t::debug

Definition at line 172 of file types.h.

3.36.2.4 uint16_t tss_t::ds

Definition at line 164 of file types.h.

3.36.2.5 uint32_t tss_t::eax

Definition at line 142 of file types.h.

3.36.2.6 uint32_t tss_t::ebp

Definition at line 152 of file types.h.

3.36.2.7 uint32_t tss_t::ebx

Definition at line 148 of file types.h.

3.36.2.8 uint32_t tss_t::ecx

Definition at line 144 of file types.h.

3.36.2.9 uint32_t tss_t::edi

Definition at line 156 of file types.h.

3.36.2.10 uint32_t tss_t::edx

Definition at line 146 of file types.h.

3.36.2.11 uint32_t tss_t::eflags

Definition at line 140 of file types.h.

3.36.2.12 uint32_t tss_t::eip

Definition at line 138 of file types.h.

3.36.2.13 uint16_t tss_t::es

Definition at line 158 of file types.h.

3.36.2.14 uint32_t tss_t::esi

Definition at line 154 of file types.h.

3.36.2.15 uint32_t tss_t::esp

Definition at line 150 of file types.h.

3.36.2.16 addr_t tss_t::esp0

Definition at line 124 of file types.h.

Referenced by cpu_init_data(), and thread_context_switch().

3.36.2.17 addr_t tss_t::esp1

Definition at line 128 of file types.h.

Referenced by cpu_init_data(), and thread_context_switch().

3.36.2.18 addr_t tss_t::esp2

Definition at line 132 of file types.h.

Referenced by cpu_init_data(), and thread_context_switch().

3.36.2.19 uint16_t tss_t::fs

Definition at line 166 of file types.h.

3.36.2.20 uint16_t tss_t::gs

Definition at line 168 of file types.h.

3.36.2.21 uint16_t tss_t::iomap

Definition at line 173 of file types.h.

3.36.2.22 uint16_t tss_t::ldt

Definition at line 170 of file types.h.

3.36.2.23 uint16_t tss_t::prev

Definition at line 122 of file types.h.

3.36.2.24 uint16_t tss_t::ss

Definition at line 162 of file types.h.

3.36.2.25 uint16_t tss_t::ss0

Definition at line 126 of file types.h.

Referenced by cpu_init_data().

3.36.2.26 uint16_t tss_t::ss1

Definition at line 130 of file types.h.

Referenced by cpu_init_data().

3.36.2.27 uint16_t tss_t::ss2

Definition at line 134 of file types.h.

Referenced by cpu_init_data().

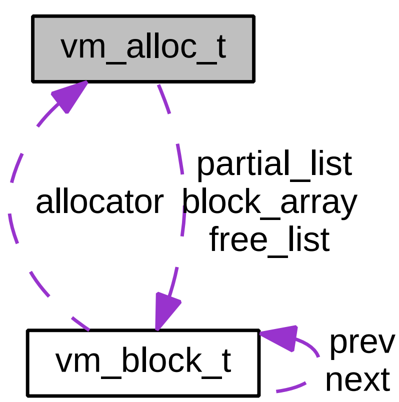
The documentation for this struct was generated from the following file:

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

3.37 vm_alloc_t Struct Reference

```
#include <vm_alloc.h>
```

Collaboration diagram for vm_alloc_t:



Data Fields

- **addr_t base_addr**

  base address of memory managed by allocator

- **addr_t start_addr**

  start address of memory actually available to the allocator
- **addr_t end_addr**

  end address of memory actually available to the allocator
- unsigned int **block_count**

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

  array of memory block descriptors
- unsigned int **array_pages**

  number of pages allocated for block array
- struct **vm_block_t** ∗ **free_list**

  list of completely free blocks
- struct **vm_block_t** ∗ **partial_list**

  list of partially free blocks

3.37.1 Detailed Description

Definition at line 64 of file vm_alloc.h.

3.37.2 Field Documentation

3.37.2.1 unsigned int vm_alloc_t::array_pages

number of pages allocated for block array

Definition at line 81 of file vm_alloc.h.

Referenced by vm_alloc_init_allocator().

3.37.2.2 addr_t vm_alloc_t::base_addr

base address of memory managed by allocator

Definition at line 66 of file vm_alloc.h.

Referenced by vm_alloc_add_region(), vm_alloc_init_allocator(), and vm_free().

3.37.2.3 struct vm_block_t∗ vm_alloc_t::block_array

array of memory block descriptors

Definition at line 78 of file vm_alloc.h.

Referenced by vm_alloc_add_region(), vm_alloc_destroy(), vm_alloc_init_allocator(), and vm_free().

3.37.2.4 unsigned int vm_alloc_t::block_count

number of memory blocks managed by this allocator

Definition at line 75 of file vm_alloc.h.

Referenced by vm_alloc_init_allocator().

3.37.2.5 addr_t vm_alloc_t::end_addr

end address of memory actually available to the allocator

Definition at line 72 of file vm_alloc.h.

Referenced by vm_alloc_init_allocator().

3.37.2.6 struct vm_block_t* vm_alloc_t::free_list

list of completely free blocks

Definition at line 84 of file vm_alloc.h.

Referenced by vm_alloc(), vm_alloc_free_block(), vm_alloc_init_allocator(), vm_alloc_low_latency(), and vm_alloc_-unlink_block().

3.37.2.7 struct vm_block_t* vm_alloc_t::partial_list

list of partially free blocks

Definition at line 87 of file vm_alloc.h.

Referenced by vm_alloc(), vm_alloc_destroy(), vm_alloc_init_allocator(), vm_alloc_low_latency(), vm_alloc_partial_-block(), and vm_alloc_unlink_block().

3.37.2.8 addr_t vm_alloc_t::start_addr

start address of memory actually available to the allocator

Definition at line 69 of file vm_alloc.h.

Referenced by vm_alloc_add_region(), vm_alloc_init_allocator(), and vm_free().

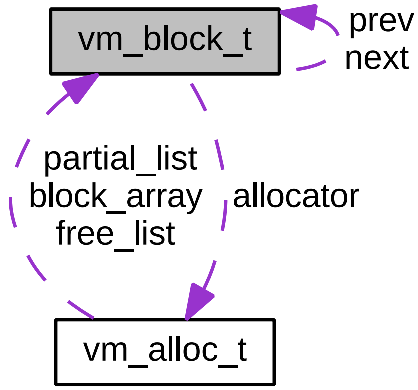
The documentation for this struct was generated from the following file:

- include/**vm_alloc.h**

3.38 vm_block_t Struct Reference

```
#include <vm_alloc.h>
```

Collaboration diagram for vm_block_t:



Data Fields

- **addr_t base_addr**

    base address of memory block
- struct **vm_alloc_t** ∗ **allocator**

    allocator to which this block belongs
- **addr_t** ∗ **stack_ptr**

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

    base address of free page stack
- **addr_t stack_next**

    next page address to add to stack, used for deferred stack initialization
- struct **vm_block_t** ∗ **prev**

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

    link next block in free list

3.38.1 Detailed Description

Definition at line 90 of file vm_alloc.h.

3.38.2 Field Documentation

3.38.2.1 struct vm_alloc_t∗ vm_block_t::allocator

allocator to which this block belongs

Definition at line 95 of file vm_alloc.h.

Referenced by vm_alloc_free_block(), vm_alloc_init_allocator(), vm_alloc_partial_block(), and vm_alloc_unlink_block().

3.38.2.2 addr_t vm_block_t::base_addr

base address of memory block

Definition at line 92 of file vm_alloc.h.

Referenced by vga_set_base_addr(), vm_alloc_add_region(), vm_alloc_custom_block(), vm_alloc_grow_stack(), vm_-alloc_init_allocator(), and vm_alloc_partial_block().

3.38.2.3 struct vm_block_t* vm_block_t::next

link next block in free list

Definition at line 110 of file vm_alloc.h.

Referenced by vm_alloc_destroy(), vm_alloc_free_block(), vm_alloc_grow_single(), vm_alloc_grow_stack(), vm_alloc-_init_allocator(), vm_alloc_partial_block(), and vm_alloc_unlink_block().

3.38.2.4 struct vm_block_t* vm_block_t::prev

link previous block in free list

Definition at line 107 of file vm_alloc.h.

Referenced by vm_alloc_free_block(), vm_alloc_grow_single(), vm_alloc_grow_stack(), vm_alloc_partial_block(), and vm_alloc_unlink_block().

3.38.2.5 addr_t* vm_block_t::stack_addr

base address of free page stack

Definition at line 101 of file vm_alloc.h.

Referenced by vm_alloc_custom_block(), vm_alloc_destroy(), vm_alloc_init_allocator(), vm_alloc_partial_block(), vm_-alloc_unlink_block(), and vm_free().

3.38.2.6 addr_t vm_block_t::stack_next

next page address to add to stack, used for deferred stack initialization

Definition at line 104 of file vm_alloc.h.

Referenced by vm_alloc_grow_single(), vm_alloc_grow_stack(), and vm_alloc_partial_block().

3.38.2.7 addr_t* vm_block_t::stack_ptr

stack pointer for stack of free pages in partially allocated blocks

Definition at line 98 of file vm_alloc.h.

Referenced by vm_alloc(), vm_alloc_custom_block(), vm_alloc_free_block(), vm_alloc_grow_single(), vm_alloc_grow_-stack(), vm_alloc_low_latency(), vm_alloc_partial_block(), vm_alloc_unlink_block(), and vm_free().

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

- include/**vm_alloc.h**

3.39 x86_cpuid_regs_t Struct Reference

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

Data Fields

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

3.39.1 Detailed Description

Definition at line 39 of file x86.h.

3.39.2 Field Documentation

3.39.2.1 uint32_t x86_cpuid_regs_t::eax

Definition at line 40 of file x86.h.

Referenced by cpu_detect_features().

3.39.2.2 uint32_t x86_cpuid_regs_t::ebx

Definition at line 41 of file x86.h.

Referenced by cpu_detect_features().

3.39.2.3 uint32_t x86_cpuid_regs_t::ecx

Definition at line 42 of file x86.h.

Referenced by cpu_detect_features().

3.39.2.4 uint32_t x86_cpuid_regs_t::edx

Definition at line 43 of file x86.h.

Referenced by cpu_detect_features().

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

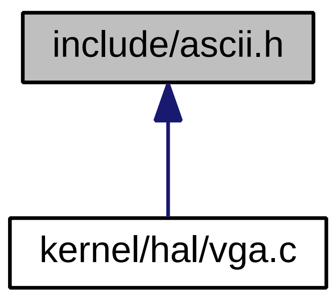
- include/hal/**x86.h**

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.

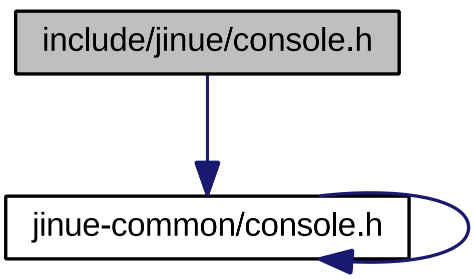
4.1.1.4 #define CHAR_LF 0x0a

Definition at line 39 of file ascii.h.

4.2 include/console.h File Reference

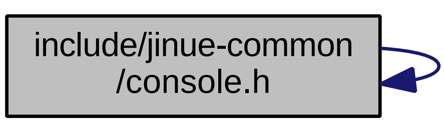
4.3 include/jinue/console.h File Reference

`#include <jinue-common/console.h>`
Include dependency graph for console.h:

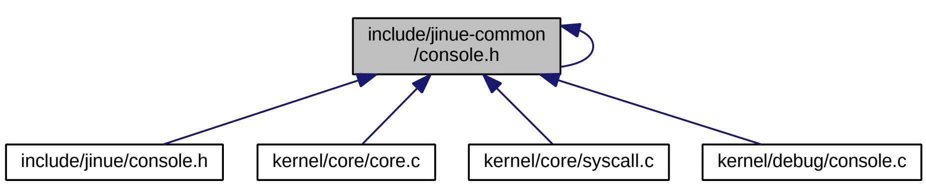


4.4 include/jinue-common/console.h File Reference

`#include <jinue-common/console.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

Functions

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

4.4.1 Macro Definition Documentation

4.4.1.1 #define CONSOLE_SERIAL_BAUD_RATE 115200

Definition at line 39 of file console.h.

4.4.1.2 #define CONSOLE_SERIAL_IOPORT SERIAL_COM1_IOPORT

Definition at line 37 of file console.h.

4.4.2 Function Documentation

4.4.2.1 void console_init (void)

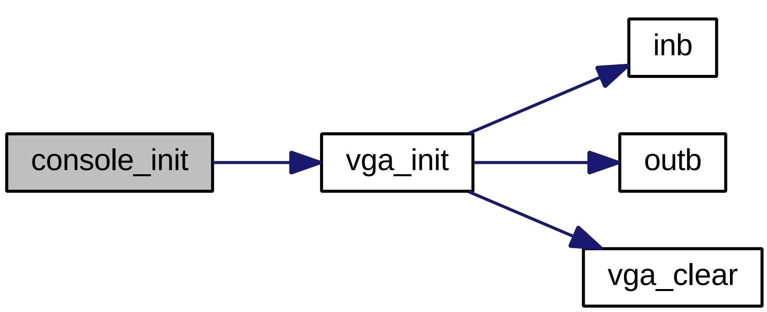
Definition at line 37 of file console.c.

References vga_init().

Referenced by kmain().

```
37                          {
38     vga_init();
39 }
```

Here is the call graph for this function:



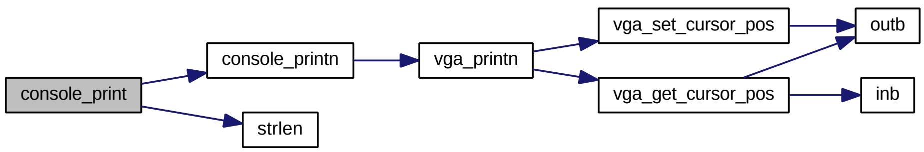
4.4.2.2 void console_print (const char * *message*)

Definition at line 49 of file console.c.

References console_printn(), and strlen().

```
49                                          {
50      console_printn(message, strlen(message));
51 }
```

Here is the call graph for this function:



4.4.2.3 void console_printn (const char * *message,* unsigned int *n*)

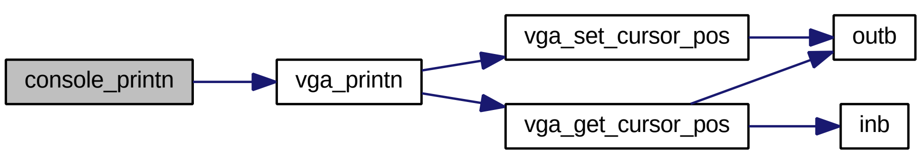
Definition at line 41 of file console.c.

References vga_printn().

Referenced by console_print(), and dispatch_syscall().

```
41                                                              {
42      vga_printn(message, n);
43 }
```

Here is the call graph for this function:



4.4.2.4 void console_putc (char *c*)

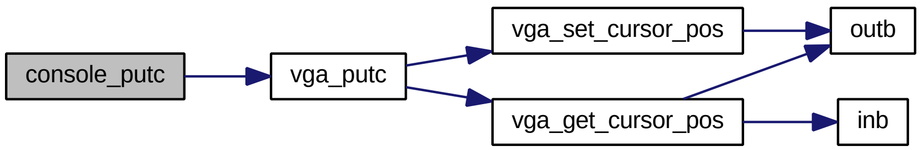
Definition at line 45 of file console.c.

References vga_putc().

Referenced by dispatch_syscall().

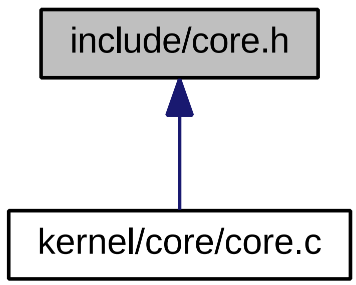
```
45                          {
46      vga_putc(c);
47 }
```

Here is the call graph for this function:



4.5 include/core.h File Reference

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



Functions

- void **kmain** (void)

4.5.1 Function Documentation

4.5.1.1 void kmain (void)

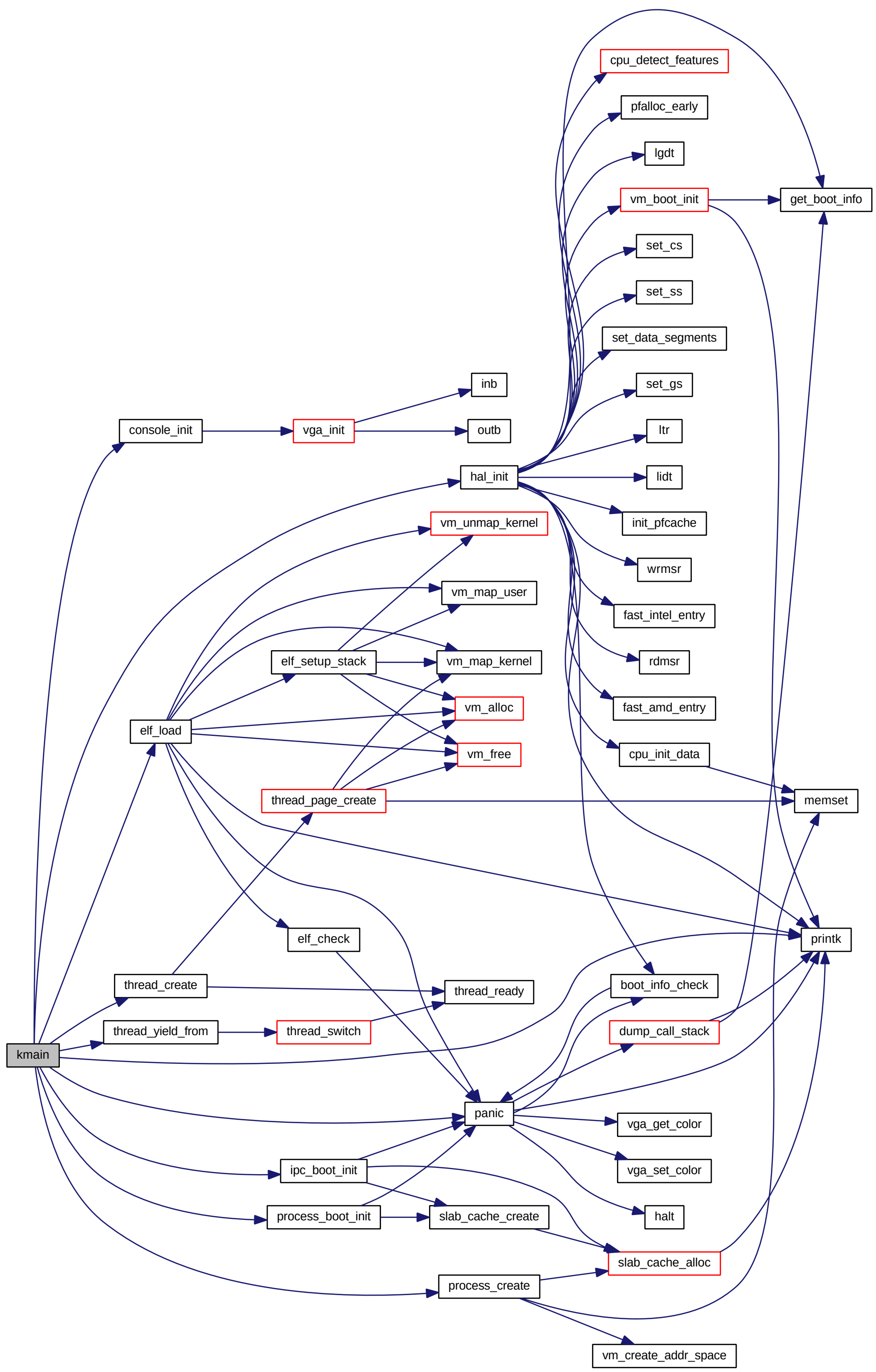
Definition at line 66 of file core.c.

References process_t::addr_space, console_init(), elf_load(), elf_info_t::entry, hal_init(), ipc_boot_init(), NULL, panic(), printk(), process_boot_init(), process_create(), elf_info_t::stack_addr, thread_create(), and thread_yield_from().

```
66                {
67      elf_info_t elf_info;
68
69      /* initialize console and say hello */
70      console_init();
71
72      printk("Kernel revision " GIT_REVISION " built " BUILD_TIME " on " BUILD_HOST "\n");
73
74      /* initialize hardware abstraction layer */
75      hal_init();
76
77      /* initialize caches */
78      ipc_boot_init();
79      process_boot_init();
80
81      /* create process for process manager */
82      process_t *process = process_create();
83
84      /* load process manager binary */
85      Elf32_Ehdr *elf = find_process_manager();
```

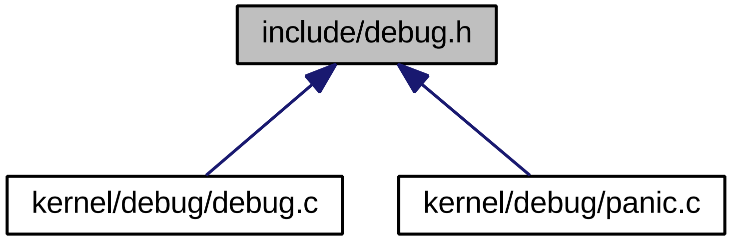
```
    elf_load(&elf_info, elf, &process->addr_space);

    /* create initial thread */
    thread_t *thread = thread_create(
            process,
            elf_info.entry,
            elf_info.stack_addr);

    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:



4.6 include/debug.h File Reference

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



Functions

- void **dump_call_stack** (void)

4.6.1 Function Documentation

4.6.1.1 void dump_call_stack (void)

Definition at line 42 of file debug.c.

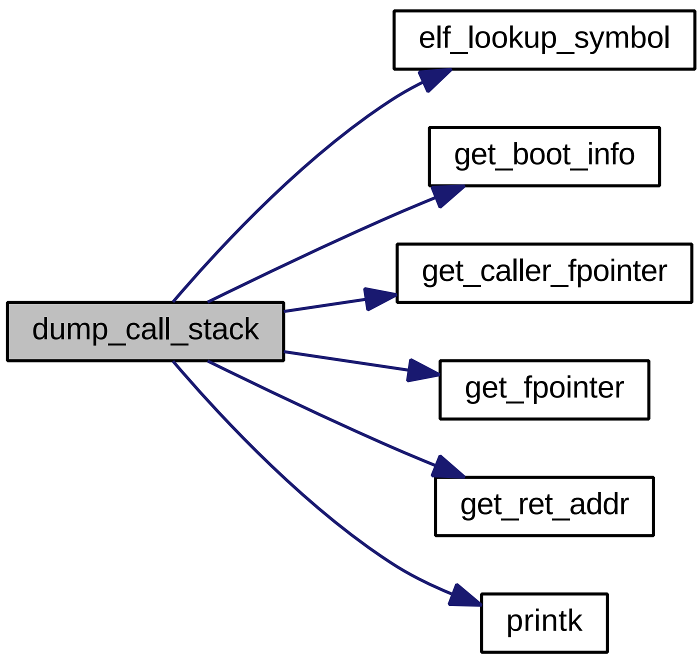
References elf_symbol_t::addr, boot_info, elf_lookup_symbol(), get_boot_info(), get_caller_fpointer(), get_fpointer(), get_ret_addr(), boot_info_t::kernel_start, elf_symbol_t::name, NULL, printk(), and STT_FUNCTION.

Referenced by panic().

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

```
                    "\t0x%x (%s+%u)\n",
                    return_addr,
                    name,
                    return_addr - symbol.addr);
        }

        fptr = get_caller_fpointer(fptr);
    }
}
```

Here is the call graph for this function:

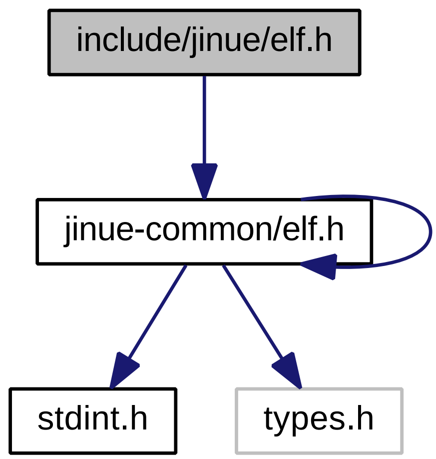


4.7 include/elf.h File Reference

4.8 include/jinue/elf.h File Reference

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

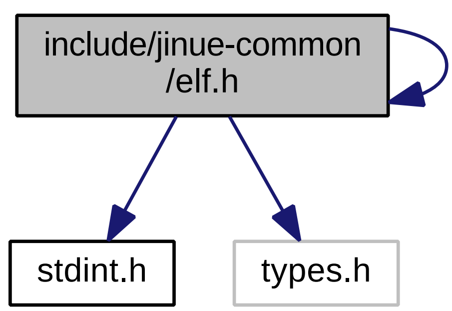
Include dependency graph for elf.h:



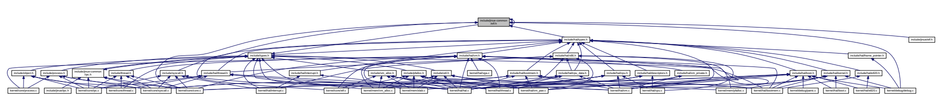
4.9 include/jinue-common/elf.h File Reference

```
#include <jinue-common/elf.h>
#include <stdint.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)
- void **elf_setup_stack** (**elf_info_t** ∗info)
- int **elf_lookup_symbol** (const **Elf32_Ehdr** ∗elf_header, **Elf32_Addr** addr, int type, **elf_symbol_t** ∗result)

4.9.1 Macro Definition Documentation

4.9.1.1 #define AT_BASE 7

Base address.

Definition at line 339 of file elf.h.

4.9.1.2 #define AT_DCACHEBSIZE 10

Data cache block size.

Definition at line 348 of file elf.h.

4.9.1.3 #define AT_ENTRY 9

Program entry point.

Definition at line 345 of file elf.h.

Referenced by elf_setup_stack().

4.9.1.4 #define AT_EXECFD 2

Program file descriptor.

Definition at line 324 of file elf.h.

4.9.1.5 #define AT_FLAGS 8

Flags.

Definition at line 342 of file elf.h.

4.9.1.6 #define AT_HWCAP 16

Machine-dependent processor feature flags.

Definition at line 360 of file elf.h.

4.9.1.7 #define AT_HWCAP2 26

More machine-dependent processor feature flags.

Definition at line 363 of file elf.h.

4.9.1.8 #define AT_ICACHEBSIZE 11

Instruction cache block size.

Definition at line 351 of file elf.h.

4.9.1.9 #define AT_IGNORE 1

Ignore entry.

Definition at line 321 of file elf.h.

4.9.1.10 #define AT_NULL 0

Last entry.

Definition at line 318 of file elf.h.

Referenced by elf_setup_stack().

4.9.1.11 #define AT_PAGESZ 6

Page size.

Definition at line 336 of file elf.h.

Referenced by elf_setup_stack().

4.9.1.12 #define AT_PHDR 3

Program headers address.

Definition at line 327 of file elf.h.

Referenced by elf_setup_stack().

4.9.1.13 #define AT_PHENT 4

Size of program header entry.

Definition at line 330 of file elf.h.

Referenced by elf_setup_stack().

4.9.1.14 #define AT_PHNUM 5

Number of program header entries.

Definition at line 333 of file elf.h.

Referenced by elf_setup_stack().

4.9.1.15 #define AT_STACKBASE 13

Stack base address for main thread.

Definition at line 357 of file elf.h.

Referenced by elf_setup_stack().

4.9.1.16 #define AT_SYSINFO_EHDR 33

Address of vDSO.

Definition at line 366 of file elf.h.

4.9.1.17 #define AT_UCACHEBSIZE 12

Unified cache block size.

Definition at line 354 of file elf.h.

4.9.1.18 #define EI_CLASS 4

File class.

Definition at line 50 of file elf.h.

Referenced by elf_check().

4.9.1.19 #define EI_DATA 5

Data encoding.

Definition at line 53 of file elf.h.

Referenced by elf_check().

4.9.1.20 #define EI_MAG0 0

Index of file identification - byte 0.

Definition at line 38 of file elf.h.

Referenced by elf_check().

4.9.1.21 #define EI_MAG1 1

Index of file identification - byte 1.

Definition at line 41 of file elf.h.

Referenced by elf_check().

4.9.1.22 #define EI_MAG2 2

Index of file identification - byte 2.

Definition at line 44 of file elf.h.

Referenced by elf_check().

4.9.1.23 #define EI_MAG3 3

Index of file identification - byte 3.

Definition at line 47 of file elf.h.

Referenced by elf_check().

4.9.1.24 #define EI_NIDENT 16

size of e_ident[]

Definition at line 62 of file elf.h.

4.9.1.25 #define EI_PAD 7

Start of padding bytes.

Definition at line 59 of file elf.h.

4.9.1.26 #define EI_VERSION 6

File version.

Definition at line 56 of file elf.h.

Referenced by elf_check().

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

Definition at line 233 of file elf.h.

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

Definition at line 235 of file elf.h.

Referenced by elf_lookup_symbol().

4.9.1.29 #define ELF_MAGIC0 0x7f

File identification - byte 0 (0x7f)

Definition at line 66 of file elf.h.

Referenced by elf_check().

4.9.1.30 #define ELF_MAGIC1 'E'

File identification - byte 1 ('E')

Definition at line 69 of file elf.h.

Referenced by elf_check().

4.9.1.31 #define ELF_MAGIC2 'L'

File identification - byte 2 ('L')

Definition at line 72 of file elf.h.

Referenced by elf_check().

4.9.1.32 #define ELF_MAGIC3 'F'

File identification - byte 3 ('F')

Definition at line 75 of file elf.h.

Referenced by elf_check().

4.9.1.33 #define ELFCLASS32 1

32-bit objects

Definition at line 132 of file elf.h.

Referenced by elf_check().

4.9.1.34 #define ELFCLASS64 2

64-bit objects

Definition at line 135 of file elf.h.

4.9.1.35 #define ELFCLASSNONE 0

Invalid class.

Definition at line 129 of file elf.h.

4.9.1.36 #define ELFDATA2LSB 1

Little-endian.

Definition at line 142 of file elf.h.

Referenced by elf_check().

4.9.1.37 #define ELFDATA2MSB 2

Big-endian.

Definition at line 145 of file elf.h.

4.9.1.38 #define ELFDATANONE 0

Invalid data encoding.

Definition at line 139 of file elf.h.

4.9.1.39 #define EM_386 3

Intel 80386.

Definition at line 85 of file elf.h.

Referenced by elf_check().

4.9.1.40 #define EM_AARCH64 183

64-bit AARCH64 ARM

Definition at line 106 of file elf.h.

4.9.1.41 #define EM_ALTERA_NIOS2 113

Altera Nios 2 32-bit soft processor.

Definition at line 103 of file elf.h.

4.9.1.42 #define EM_ARM 40

32-bit ARM

Definition at line 94 of file elf.h.

4.9.1.43 #define EM_MICROBLAZE 189

Xilinx MicroBlaze 32-bit soft processor.

Definition at line 109 of file elf.h.

4.9.1.44 #define EM_MIPS 8

MIPS RS3000.

Definition at line 88 of file elf.h.

4.9.1.45 #define EM_NONE 0

No machine.

Definition at line 79 of file elf.h.

4.9.1.46 #define EM_OPENRISC 92

OpenRISC 32-bit embedded processor.

Definition at line 100 of file elf.h.

4.9.1.47 #define EM_SPARC 2

SPARC.

Definition at line 82 of file elf.h.

4.9.1.48 #define EM_SPARC32PLUS 18

Enhanced instruction set SPARC.

Definition at line 91 of file elf.h.

4.9.1.49 #define EM_X86_64 62

AMD64/X86-64.

Definition at line 97 of file elf.h.

4.9.1.50 #define ET_CORE 4

Core file.

Definition at line 125 of file elf.h.

4.9.1.51 #define ET_DYN 3

Shared object file.

Definition at line 122 of file elf.h.

4.9.1.52 #define ET_EXEC 2

Executable file.

Definition at line 119 of file elf.h.

Referenced by elf_check().

4.9.1.53 #define ET_NONE 0

No file type.

Definition at line 113 of file elf.h.

4.9.1.54 #define ET_REL 1

Relocatable file.

Definition at line 116 of file elf.h.

4.9.1.55 #define PF_R (1 $<<$ 2)

Definition at line 242 of file elf.h.

4.9.1.56 #define PF_W (1 $<<$ 1)

Definition at line 244 of file elf.h.

Referenced by elf_load().

4.9.1.57 #define PF_X (1 $<<$ 0)

Definition at line 246 of file elf.h.

4.9.1.58 #define PT_DYNAMIC 2

Dynamic linking information.

Definition at line 155 of file elf.h.

4.9.1.59 #define PT_INTERP 3

Path to program interpreter.

Definition at line 158 of file elf.h.

4.9.1.60 #define PT_LOAD 1

Loadable segment.

Definition at line 152 of file elf.h.

Referenced by elf_load().

4.9.1.61 #define PT_NOTE 4

Location and size of notes.

Definition at line 161 of file elf.h.

4.9.1.62 #define PT_NULL 0

Unused entry.

Definition at line 149 of file elf.h.

4.9.1.63 #define PT_PHDR 6

Program header table.

Definition at line 167 of file elf.h.

4.9.1.64 #define PT_SHLIB 5

Unspecified semantics.

Definition at line 164 of file elf.h.

4.9.1.65 #define SHT_DYNAMIC 6

Information for dynamic linking.

Definition at line 189 of file elf.h.

4.9.1.66 #define SHT_DYNSYM 11

Dynamic symbols table.

Definition at line 204 of file elf.h.

4.9.1.67 #define SHT_HASH 5

Symbol hash table.

Definition at line 186 of file elf.h.

4.9.1.68 #define SHT_NOBITS 8

Section without data (.bss)

Definition at line 195 of file elf.h.

4.9.1.69 #define SHT_NOTE 7

Notes section.

Definition at line 192 of file elf.h.

4.9.1.70 #define SHT_NULL 0

Inactive section.

Definition at line 171 of file elf.h.

4.9.1.71 #define SHT_PROGBITS 1

Program data.

Definition at line 174 of file elf.h.

4.9.1.72 #define SHT_REL 9

Relocations without addends.

Definition at line 198 of file elf.h.

4.9.1.73 #define SHT_RELA 4

Relocations with addends.

Definition at line 183 of file elf.h.

4.9.1.74 #define SHT_SHLIB 10

Reserved, unspecified semantic, not ABI compliant.

Definition at line 201 of file elf.h.

4.9.1.75 #define SHT_STRTAB 3

String table.

Definition at line 180 of file elf.h.

4.9.1.76 #define SHT_SYMTAB 2

Symbol table.

Definition at line 177 of file elf.h.

Referenced by elf_lookup_symbol().

4.9.1.77 #define STB_GLOBAL 1

Global binding.

Definition at line 211 of file elf.h.

4.9.1.78 #define STB_LOCAL 0

Local binding.

Definition at line 208 of file elf.h.

4.9.1.79 #define STB_WEAK 2

Weak binding.

Definition at line 214 of file elf.h.

4.9.1.80 #define STN_UNDEF 0

Undefined symbol index.

Definition at line 239 of file elf.h.

4.9.1.81 #define STT_FILE 4

Source file.

Definition at line 230 of file elf.h.

4.9.1.82 #define STT_FUNCTION 2

Function or other executable code.

Definition at line 224 of file elf.h.

Referenced by dump_call_stack().

4.9.1.83 #define STT_NOTYPE 0

Unspecified type.

Definition at line 218 of file elf.h.

4.9.1.84 #define STT_OBJECT 1

Data object.

Definition at line 221 of file elf.h.

4.9.1.85 #define STT_SECTION 3

Section symbol.

Definition at line 227 of file elf.h.

4.9.2 Typedef Documentation

4.9.2.1 typedef Elf32_auxv_t auxv_t

Definition at line 315 of file elf.h.

4.9.2.2 typedef uint32_t Elf32_Addr

Definition at line 248 of file elf.h.

4.9.2.3 typedef uint16_t Elf32_Half

Definition at line 250 of file elf.h.

4.9.2.4 typedef uint32_t Elf32_Off

Definition at line 252 of file elf.h.

4.9.2.5 typedef int32_t Elf32_Sword

Definition at line 254 of file elf.h.

4.9.2.6 typedef uint32_t Elf32_Word

Definition at line 256 of file elf.h.

4.9.3 Function Documentation

4.9.3.1 void elf_check (Elf32_Ehdr * *elf*)

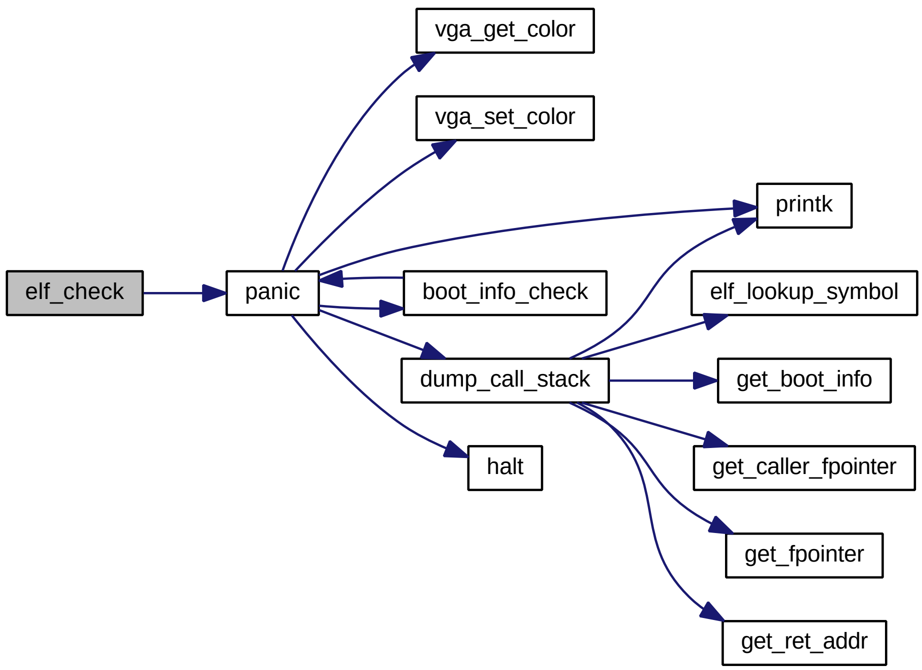
Definition at line 42 of file elf.c.

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().

```
42                                  {
43      /* check: valid ELF binary magic number */
44      if(     elf->e_ident[EI_MAG0] != ELF_MAGIC0 ||
45              elf->e_ident[EI_MAG1] != ELF_MAGIC1 ||
46              elf->e_ident[EI_MAG2] != ELF_MAGIC2 ||
47              elf->e_ident[EI_MAG3] != ELF_MAGIC3 ) {
48          panic("Not an ELF binary");
49      }
50
51      /* check: 32-bit objects */
52      if(elf->e_ident[EI_CLASS] != ELFCLASS32) {
53          panic("Bad file class");
54      }
55
56      /* check: endianess */
57      if(elf->e_ident[EI_DATA] != ELFDATA2LSB) {
58          panic("Bad endianess");
59      }
60
61      /* check: version */
62      if(elf->e_version != 1 || elf->e_ident[EI_VERSION] != 1) {
63          panic("Not ELF version 1");
64      }
65
66      /* check: machine */
67      if(elf->e_machine != EM_386) {
68          panic("This process manager binary does not target the x86 architecture");
69      }
70
71      /* check: the 32-bit Intel architecture defines no flags */
72      if(elf->e_flags != 0) {
73          panic("Invalid flags specified");
74      }
75
76      /* check: file type is executable */
77      if(elf->e_type != ET_EXEC) {
78          panic("process manager binary is not an an executable");
79      }
80
81      /* check: must have a program header */
82      if(elf->e_phoff == 0 || elf->e_phnum == 0) {
83          panic("No program headers");
84      }
85
86      /* check: must have an entry point */
87      if(elf->e_entry == 0) {
88          panic("No entry point for process manager");
89      }
90
91      /* check: program header entry size */
92      if(elf->e_phentsize != sizeof(Elf32_Phdr)) {
93          panic("Unsupported program header size");
94      }
95  }
```

Here is the call graph for this function:



4.9.3.2 void elf_load (elf_info_t ∗ *info,* Elf32_Ehdr ∗ *elf,* addr_space_t ∗ *addr_space*)

TODO: add exec flag once PAE is enabled

TODO: add exec flag once PAE is enabled

Definition at line 97 of file elf.c.

References elf_info_t::addr_space, elf_info_t::at_phdr, elf_info_t::at_phent, elf_info_t::at_phnum, Elf32_Ehdr::e_entry, Elf32_Ehdr::e_phentsize, Elf32_Ehdr::e_phnum, Elf32_Ehdr::e_phoff, EARLY_PTR_TO_PFADDR, elf_check(), elf-_setup_stack(), elf_info_t::entry, global_page_allocator, Elf32_Phdr::p_filesz, Elf32_Phdr::p_memsz, PAGE_MASK, page_offset_of, PAGE_SIZE, panic(), PF_W, pfalloc, printk(), PT_LOAD, vm_alloc(), VM_FLAG_READ_ONLY, VM_-FLAG_READ_WRITE, vm_free(), vm_map_kernel(), vm_map_user(), and vm_unmap_kernel().

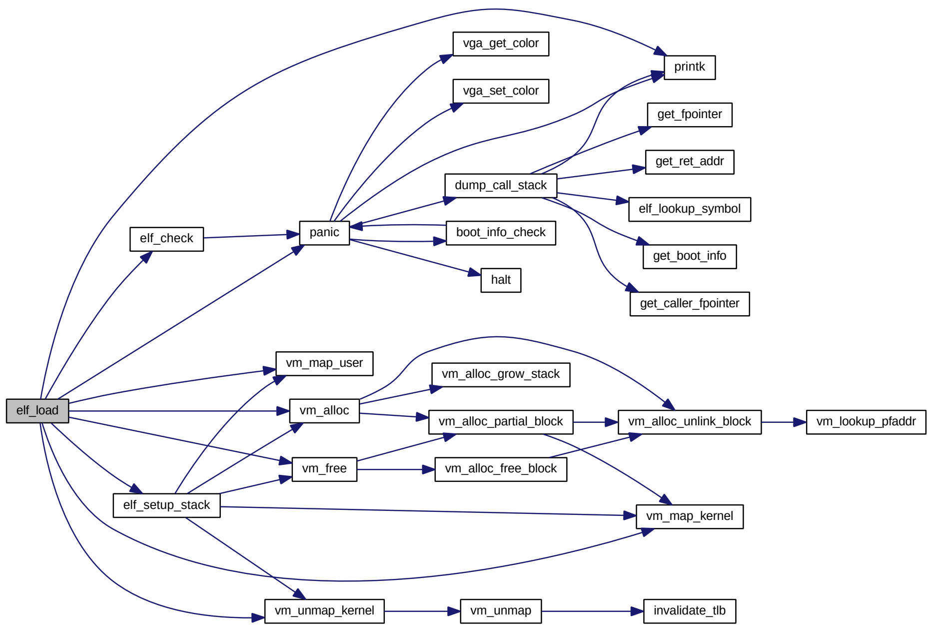
Referenced by kmain().

```
                                                                            {
      Elf32_Phdr *phdr;
      pfaddr_t page;
       addr_t vpage;
       char *vptr, *vend, *vfend, *vnext;
       char *file_ptr;
       char *stop;
       char *dest, *dest_page;
       unsigned int idx;
       unsigned long flags;


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

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

       info->at_phdr       = (addr_t)phdr;
       info->at_phnum      = elf->e_phnum;
```

```c
    info->at_phent      = elf->e_phentsize;
    info->addr_space    = addr_space;
    info->entry         = (addr_t)elf->e_entry;

    /* temporary page for copies */
    dest_page = (char *)vm_alloc(global_page_allocator);

    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 */
        file_ptr = (char *)elf + phdr[idx].p_offset;
        vptr     = (char *)phdr[idx].p_vaddr;
        vend     = vptr  + phdr[idx].p_memsz;  /* limit for padding */
        vfend    = vptr  + phdr[idx].p_filesz; /* limit for copy */

        /* align on page boundaries, be inclusive,
           note that vfend is not aligned        */
        file_ptr = (char *) ( (uintptr_t)file_ptr & ~PAGE_MASK );
        vptr     = (char *) ( (uintptr_t)vptr  & ~PAGE_MASK );

        if(page_offset_of(vend) != 0) {
            vend  = (char *) ( (uintptr_t)vend  & ~PAGE_MASK );
            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) {
                /* start of this page and next page */
                vpage = (addr_t)vptr;
                vnext = vptr + PAGE_SIZE;

                /* allocate and map the new page */
                page = pfalloc();
                vm_map_kernel((addr_t)dest_page, page, VM_FLAG_READ_WRITE);

                dest = dest_page;

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

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

                /* pad */
                while(vptr < vnext) {
                    *(dest++) = 0;
                    ++vptr;
                }

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

                /* undo temporary mapping and map page in proper address
                 * space */
                vm_unmap_kernel((addr_t)dest_page);
                vm_map_user(addr_space, (addr_t)vpage, page, flags);
            }
        }
        else {
            while(vptr < vend) {
                /* perform mapping */
```

```
                vm_map_user(addr_space, (addr_t)vptr, EARLY_PTR_TO_PFADDR(file_ptr),
      VM_FLAG_READ_ONLY);

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

    vm_free(global_page_allocator, (addr_t)dest_page);

    elf_setup_stack(info);

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

Here is the call graph for this function:



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

Definition at line 284 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;
}
```

4.9.3.4 void elf_setup_stack (elf_info_t * *info*)

TODO: check for overlap of stack with loaded segments

Definition at line 214 of file elf.c.

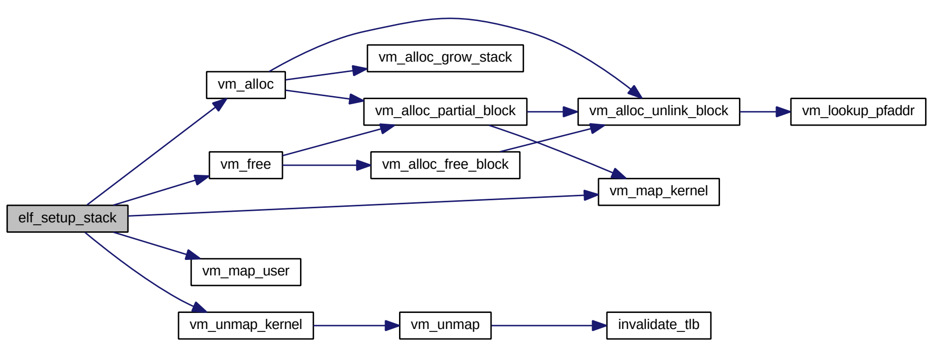
References Elf32_auxv_t::a_type, Elf32_auxv_t::a_un, Elf32_auxv_t::a_val, elf_info_t::addr_space, AT_ENTRY, AT_N-ULL, AT_PAGESZ, elf_info_t::at_phdr, AT_PHDR, elf_info_t::at_phent, AT_PHENT, elf_info_t::at_phnum, AT_PHNUM, AT_STACKBASE, elf_info_t::entry, global_page_allocator, PAGE_SIZE, pfalloc, elf_info_t::stack_addr, STACK_BAS-E, STACK_START, vm_alloc(), VM_FLAG_READ_WRITE, vm_free(), vm_map_kernel(), vm_map_user(), and vm_-unmap_kernel().

Referenced by elf_load().

```
214                                       {
215     pfaddr_t page;
216     addr_t vpage;
217 
220     /* initial stack allocation */
221     for(vpage = (addr_t)STACK_START; vpage < (addr_t)STACK_BASE; vpage +=
      PAGE_SIZE) {
222         page  = pfalloc();
223         vm_map_user(info->addr_space, vpage, page, VM_FLAG_READ_WRITE);
224     }
225 
226     /* At this point, page has the address of the stack's top-most page frame,
227      * 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 = vm_alloc(global_page_allocator);
    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);
    vm_free(global_page_allocator, top_page);
}
```

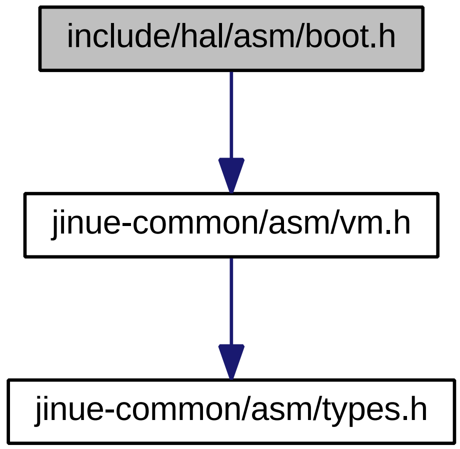
Here is the call graph for this function:



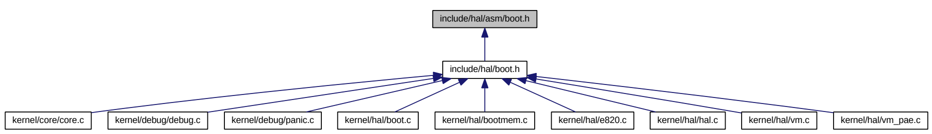
4.10 include/hal/asm/boot.h File Reference

`#include <jinue-common/asm/vm.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_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** 0x100000
- #define **BOOT_SETUP32_SIZE** **PAGE_SIZE**
- #define **BOOT_SETUP_ADDR**(x) ((x) - **BOOT_SETUP**)
- #define **BOOT_DATA_STRUCT** **BOOT_E820_ENTRIES**
- #define **BOOT_STACK_SIZE** (2 ∗ **PAGE_SIZE**)

4.10.1 Macro Definition Documentation

4.10.1.1 #define BOOT_DATA_STRUCT BOOT_E820_ENTRIES

Definition at line 66 of file boot.h.

4.10.1.2 #define BOOT_E820_ENTRIES 0x1e8

Definition at line 38 of file boot.h.

4.10.1.3 #define BOOT_E820_MAP 0x2d0

Definition at line 54 of file boot.h.

4.10.1.4 #define BOOT_E820_MAP_END 0xd00

Definition at line 56 of file boot.h.

4.10.1.5 #define BOOT_E820_MAP_SIZE (BOOT_E820_MAP_END - BOOT_E820_MAP)

Definition at line 58 of file boot.h.

4.10.1.6 #define BOOT_MAGIC 0xaa55

Definition at line 46 of file boot.h.

4.10.1.7 #define BOOT_SETUP 0x200

Definition at line 48 of file boot.h.

4.10.1.8 #define BOOT_SETUP32_ADDR 0x100000

Definition at line 60 of file boot.h.

4.10.1.9 #define BOOT_SETUP32_SIZE PAGE_SIZE

Definition at line 62 of file boot.h.

4.10.1.10 #define BOOT_SETUP_ADDR(*x*) ((x) - BOOT_SETUP)

Definition at line 64 of file boot.h.

4.10.1.11 #define BOOT_SETUP_HEADER 0x202

Definition at line 50 of file boot.h.

4.10.1.12 #define BOOT_SETUP_MAGIC 0x53726448 /∗ "HdrS", reversed ∗/

Definition at line 52 of file boot.h.

Referenced by boot_info_check().

4.10.1.13 #define BOOT_SETUP_SECTS 0x1f1

Definition at line 40 of file boot.h.

4.10.1.14 #define BOOT_SIGNATURE 0x1fe

Definition at line 44 of file boot.h.

4.10.1.15 #define BOOT_STACK_SIZE (2 ∗ PAGE_SIZE)

Definition at line 68 of file boot.h.

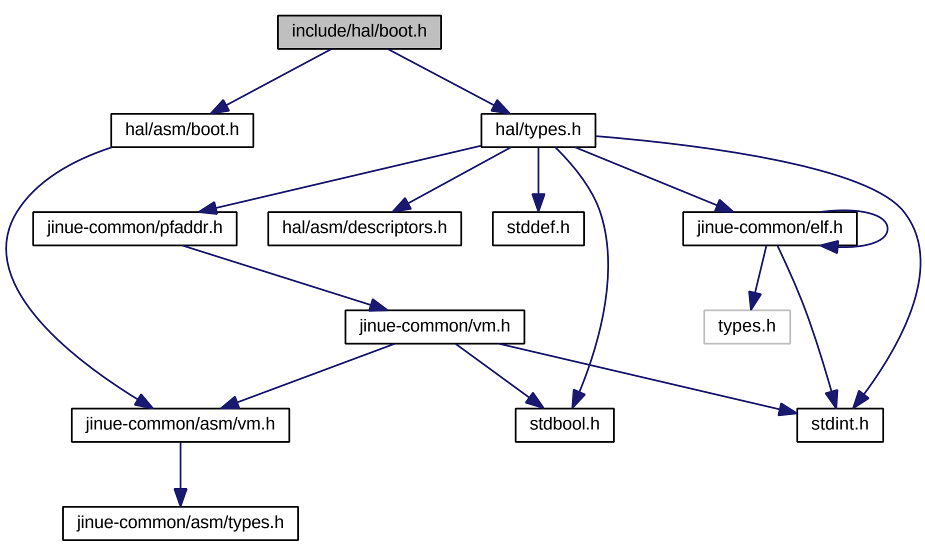
4.10.1.16 #define BOOT_SYSIZE 0x1f4

Definition at line 42 of file boot.h.

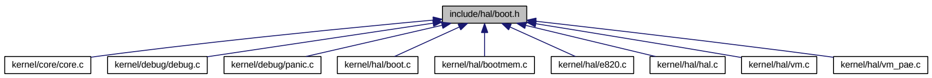
4.11 include/hal/boot.h File Reference

```
#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:



Functions

- **bool boot_info_check** (**bool** panic_on_failure)
- const **boot_info_t** ∗ **get_boot_info** (void)
- void **boot_info_dump** (void)

4.11.1 Function Documentation

4.11.1.1 bool boot_info_check (bool *panic_on_failure*)

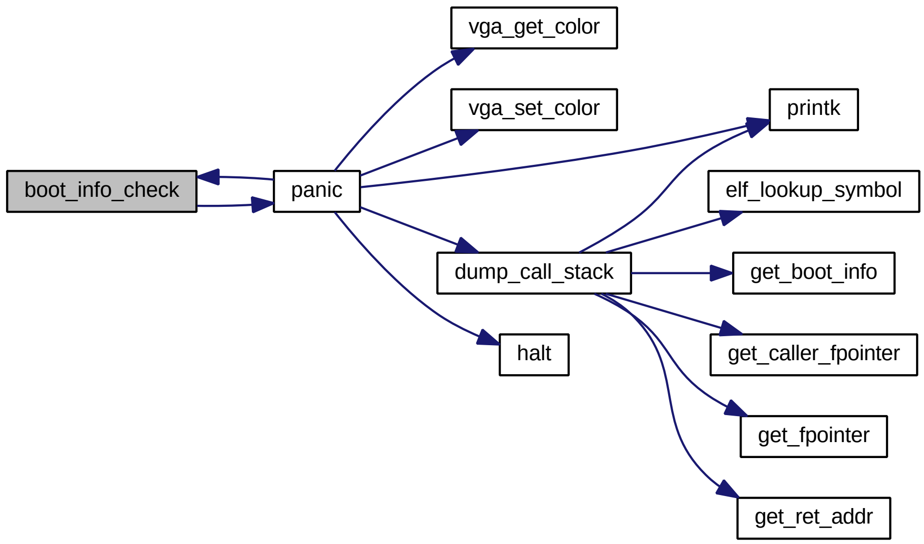
Definition at line 41 of file boot.c.

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

Referenced by hal_init(), and panic().

```
41                                         {
42      /* This data structure is accessed early during the boot process, before
43       * paging is enabled. What this means is that, if boot_info is NULL and we
44       * dereference it, it does *not* cause a page fault or any other CPU
45       * exception. */
46      if(boot_info == NULL) {
47          if(panic_on_failure) {
48              panic("Boot information structure pointer is NULL.");
49          }
50
51          return false;
52      }
53
54      if(boot_info->setup_signature != BOOT_SETUP_MAGIC) {
55          if(panic_on_failure) {
56              panic("Bad setup header signature.");
57          }
58
59          return false;
60      }
61
62      return true;
63 }
```

Here is the call graph for this function:



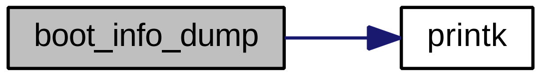
4.11.1.2 void boot_info_dump (void)

Definition at line 69 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.

```
69                            {
70      printk("Boot information structure:\n");
71      printk("    kernel_start    %x  %u\n", boot_info->kernel_start   , boot_info->kernel_start   );
72      printk("    kernel_size     %x  %u\n", boot_info->kernel_size    , boot_info->kernel_size    );
73      printk("    proc_start      %x  %u\n", boot_info->proc_start     , boot_info->proc_start     );
74      printk("    proc_size       %x  %u\n", boot_info->proc_size      , boot_info->proc_size      );
75      printk("    image_start     %x  %u\n", boot_info->image_start    , boot_info->image_start    );
76      printk("    image_top       %x  %u\n", boot_info->image_top      , boot_info->image_top      );
77      printk("    e820_entries    %x  %u\n", boot_info->e820_entries   , boot_info->e820_entries   );
78      printk("    e820_map        %x  %u\n", boot_info->e820_map       , boot_info->e820_map       );
79      printk("    boot_heap       %x  %u\n", boot_info->boot_heap      , boot_info->boot_heap      );
80      printk("    boot_end        %x  %u\n", boot_info->boot_end       , boot_info->boot_end       );
81      printk("    page_table      %x  %u\n", boot_info->page_table     , boot_info->page_table     );
82      printk("    page_directory  %x  %u\n", boot_info->page_directory , boot_info->page_directory );
83      printk("    setup_signature %x  %u\n", boot_info->setup_signature, boot_info->setup_signature);
84 }
```

Here is the call graph for this function:



4.11.1.3 const **boot_info_t**∗ get_boot_info (void)

Definition at line 65 of file boot.c.

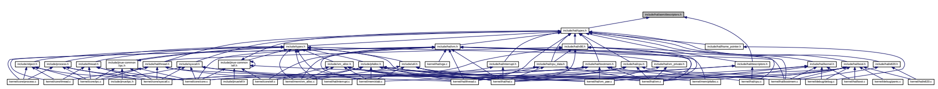
References boot_info.

Referenced by bootmem_init(), dump_call_stack(), e820_dump(), hal_init(), and vm_boot_init().

```
65                                    {
66     return boot_info;
67 }
```

4.12 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** | **S-EG_FLAG_PRESENT**)

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

  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

4.12.1 Macro Definition Documentation

4.12.1.1 #define GDT_KERNEL_CODE 1

GDT entry for kernel code segment.

Definition at line 46 of file descriptors.h.

Referenced by cpu_init_data(), and hal_init().

4.12.1.2 #define GDT_KERNEL_DATA 2

GDT entry for kernel data segment.

Definition at line 49 of file descriptors.h.

Referenced by cpu_init_data(), and hal_init().

4.12.1.3 #define GDT_LENGTH 8

number of descriptors in GDT

Definition at line 67 of file descriptors.h.

Referenced by hal_init().

4.12.1.4 #define GDT_NULL 0

GDT entry for the null descriptor.

Definition at line 43 of file descriptors.h.

Referenced by cpu_init_data().

4.12.1.5 #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(), and hal_init().

4.12.1.6 #define GDT_TSS 5

GDT entry for task-state segment (TSS)

Definition at line 58 of file descriptors.h.

Referenced by cpu_init_data(), and hal_init().

4.12.1.7 #define GDT_USER_CODE 3

GDT entry for user code segment.

Definition at line 52 of file descriptors.h.

Referenced by cpu_init_data(), hal_init(), and thread_page_create().

4.12.1.8 #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_create().

4.12.1.9 #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().

4.12.1.10 #define RPL_KERNEL 0

Definition at line 38 of file descriptors.h.

Referenced by cpu_init_data(), and hal_init().

4.12.1.11 #define RPL_USER 3

Definition at line 40 of file descriptors.h.

Referenced by hal_init(), and thread_page_create().

4.12.1.12 #define SEG_FLAG_16BIT 0

16-bit segment

Definition at line 88 of file descriptors.h.

4.12.1.13 #define SEG_FLAG_16BIT_GATE 0

16-bit gate

Definition at line 94 of file descriptors.h.

4.12.1.14 #define SEG_FLAG_32BIT (1<<14)

32-bit segment

Definition at line 85 of file descriptors.h.

Referenced by cpu_init_data().

4.12.1.15 #define SEG_FLAG_32BIT_GATE (1<<3)

32-bit gate

Definition at line 91 of file descriptors.h.

4.12.1.16 #define SEG_FLAG_BUSY (1<<1)

task is busy (for TSS descriptor)

Definition at line 97 of file descriptors.h.

4.12.1.17 #define SEG_FLAG_IN_BYTES 0

limit has byte granularity

Definition at line 103 of file descriptors.h.

Referenced by cpu_init_data().

4.12.1.18 #define SEG_FLAG_IN_PAGES (1<<15)

limit has page granularity

Definition at line 100 of file descriptors.h.

4.12.1.19 #define SEG_FLAG_KERNEL 0

kernel/supervisor segment (privilege level 0)

Definition at line 106 of file descriptors.h.

Referenced by cpu_init_data(), and hal_init().

4.12.1.20 #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().

4.12.1.21 #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.

Referenced by hal_init().

4.12.1.22 #define SEG_FLAG_NOSYSTEM (1<<4)

code/data/stack segment

Definition at line 82 of file descriptors.h.

Referenced by cpu_init_data().

4.12.1.23 #define SEG_FLAG_PRESENT (1<<7)

segment is present

Definition at line 76 of file descriptors.h.

Referenced by cpu_init_data().

4.12.1.24 #define SEG_FLAG_SYSTEM 0

system segment (i.e.

call-gate, etc.)

Definition at line 79 of file descriptors.h.

4.12.1.25 #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().

4.12.1.26 #define SEG_FLAG_USER (3<<5)

user segment (privilege level 3)

Definition at line 109 of file descriptors.h.

Referenced by cpu_init_data(), and hal_init().

4.12.1.27 #define SEG_FLAGS_OFFSET 40

offset of descriptor type in descriptor

Definition at line 70 of file descriptors.h.

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

Definition at line 35 of file descriptors.h.

Referenced by cpu_init_data(), hal_init(), and thread_page_create().

4.12.1.29 #define SEG_TYPE_CALL_GATE 12

call gate

Definition at line 146 of file descriptors.h.

4.12.1.30 #define SEG_TYPE_CODE 10

code segment

Definition at line 143 of file descriptors.h.

Referenced by cpu_init_data().

4.12.1.31 #define SEG_TYPE_DATA 2

read/write data segment

Definition at line 128 of file descriptors.h.

Referenced by cpu_init_data().

4.12.1.32 #define SEG_TYPE_INTERRUPT_GATE 6

interrupt gate

Definition at line 134 of file descriptors.h.

Referenced by hal_init().

4.12.1.33 #define SEG_TYPE_READ_ONLY 0

read-only data segment

Definition at line 125 of file descriptors.h.

4.12.1.34 #define SEG_TYPE_TASK_GATE 5

task gate

Definition at line 131 of file descriptors.h.

4.12.1.35 #define SEG_TYPE_TRAP_GATE 7

trap gate

Definition at line 137 of file descriptors.h.

4.12.1.36 #define SEG_TYPE_TSS 9

task-state segment (TSS)

Definition at line 140 of file descriptors.h.

Referenced by cpu_init_data().

4.12.1.37 #define TSS_LIMIT 104

size of the task-state segment (TSS)

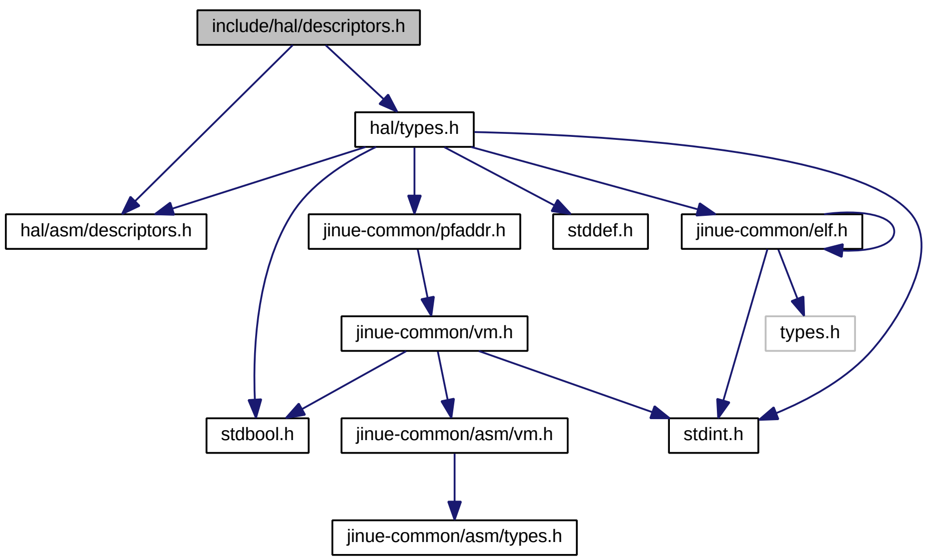
Definition at line 73 of file descriptors.h.

Referenced by cpu_init_data().

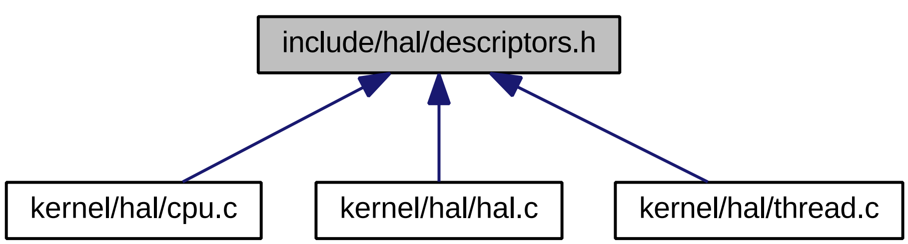
4.13 include/hal/descriptors.h File Reference

```
#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)

4.13.1 Macro Definition Documentation

4.13.1.1 #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.

Referenced by hal_init().

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

Definition at line 40 of file descriptors.h.

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

Value:

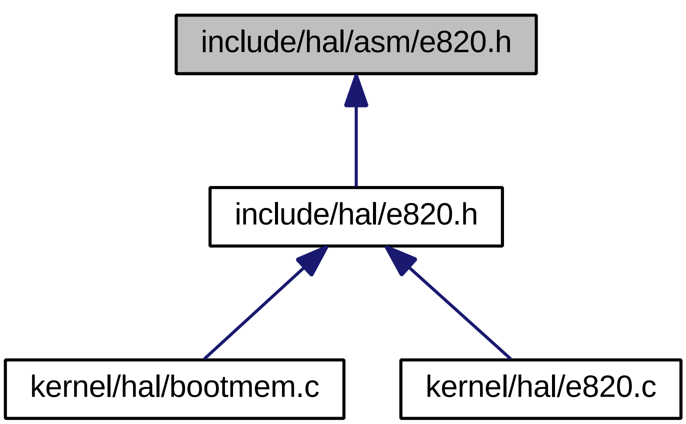
```
(       PACK_DESCRIPTOR((type),  0xf0ff,  0, SEG_FLAGS_OFFSET) \
    | PACK_DESCRIPTOR((base),  0xff,   24, 56) \
    | PACK_DESCRIPTOR((base),  0xff,   16, 32) \
    | PACK_DESCRIPTOR((base),  0xffff,  0, 16) \
    | PACK_DESCRIPTOR((limit), 0xf,    16, 48) \
    | PACK_DESCRIPTOR((limit), 0xffff,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

Definition at line 43 of file descriptors.h.

Referenced by cpu_init_data().

4.14 include/hal/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

4.14.1 Macro Definition Documentation

4.14.1.1 #define E820_ACPI 3

Definition at line 39 of file e820.h.

Referenced by e820_type_description().

4.14.1.2 #define E820_RAM 1

Definition at line 35 of file e820.h.

Referenced by e820_is_available(), and e820_type_description().

4.14.1.3 #define E820_RESERVED 2

Definition at line 37 of file e820.h.

Referenced by e820_type_description().

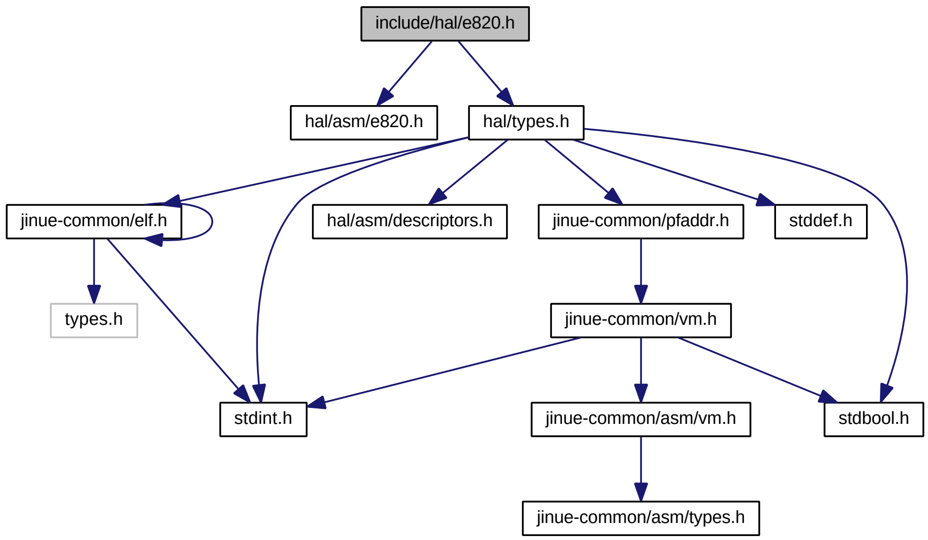
4.14.1.4 #define E820_SMAP 0x534d4150

Definition at line 41 of file e820.h.

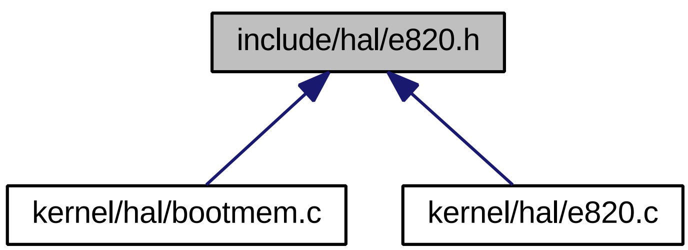
4.15 include/hal/e820.h File Reference

```
#include <hal/asm/e820.h>
#include <hal/types.h>
```

Include dependency graph for e820.h:



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



Functions

- **bool e820_is_valid** (const **e820_t** ∗e820_entry)
- **bool e820_is_available** (const **e820_t** ∗e820_entry)
- const char ∗ **e820_type_description** (**e820_type_t** type)
- void **e820_dump** (void)

4.15.1 Function Documentation

4.15.1.1 void e820_dump (void)

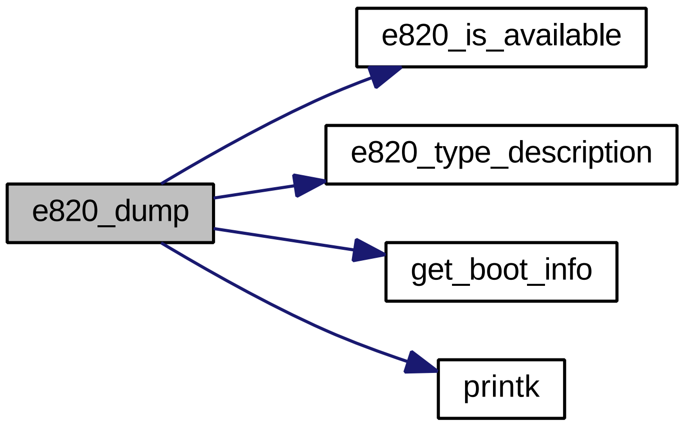
Definition at line 61 of file e820.c.

References e820_t::addr, boot_info, boot_info_t::e820_entries, e820_is_available(), boot_info_t::e820_map, e820_type_description(), get_boot_info(), printk(), e820_t::size, and e820_t::type.

```
61                     {
62     unsigned int idx;
63
64     printk("Dump of the BIOS memory map:\n");
```

```
    const boot_info_t *boot_info = get_boot_info();

    for(idx = 0; idx < boot_info->e820_entries; ++idx) {
        const e820_t *e820_entry = &boot_info->e820_map[idx];

        printk("%c [%q-%q] %s\n",
            e820_is_available(e820_entry)?'*':' ',
            e820_entry->addr,
            e820_entry->addr + e820_entry->size - 1,
            e820_type_description(e820_entry->type)
        );
    }
}
```

Here is the call graph for this function:



4.15.1.2 bool e820_is_available (const e820_t * *e820_entry*)

Definition at line 40 of file e820.c.

References E820_RAM, and e820_t::type.

Referenced by bootmem_init(), and e820_dump().

```
40                                          {
41     return e820_entry->type == E820_RAM;
42 }
```

4.15.1.3 bool e820_is_valid (const e820_t * *e820_entry*)

Definition at line 36 of file e820.c.

References e820_t::size.

Referenced by bootmem_init().

```
36                                      {
37     return e820_entry->size != 0;
38 }
```

4.15.1.4 const char* e820_type_description (e820_type_t *type*)

Definition at line 44 of file e820.c.

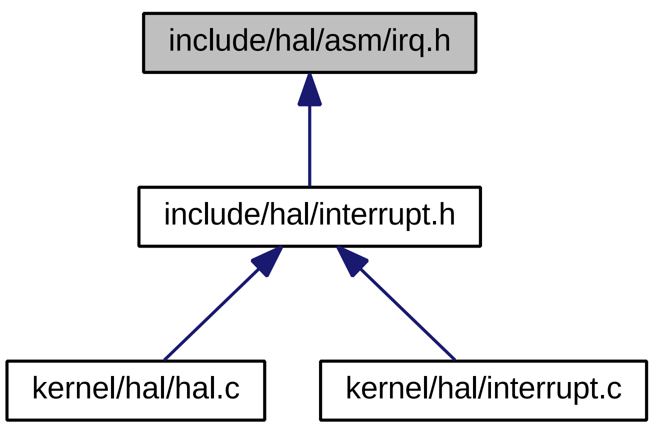
References E820_ACPI, E820_RAM, and E820_RESERVED.

Referenced by e820_dump().

```
44                                                    {
45     switch(type) {
46
47     case E820_RAM:
48         return "available";
49
50     case E820_RESERVED:
51         return "unavailable/reserved";
52
53     case E820_ACPI:
54         return "unavailable/acpi";
55
56     default:
57         return "unavailable/other";
58     }
59 }
```

4.16 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_FIRST_IRQ** 32
- #define **IDT_IRQ_COUNT** (**IDT_VECTOR_COUNT** - **IDT_FIRST_IRQ**)
- #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**))

4.16.1 Macro Definition Documentation

4.16.1.1 #define EXCEPTION_ALIGNMENT 17

Alignment Check.

Definition at line 85 of file irq.h.

4.16.1.2 #define EXCEPTION_BOUND 5

BOUND Range Exceeded.

Definition at line 55 of file irq.h.

4.16.1.3 #define EXCEPTION_BREAK 3

Breakpoint.

Definition at line 49 of file irq.h.

4.16.1.4 #define EXCEPTION_DIV_ZERO 0

Divide Error.

Definition at line 43 of file irq.h.

4.16.1.5 #define EXCEPTION_DOUBLE_FAULT 8

Double Fault.

Definition at line 64 of file irq.h.

4.16.1.6 #define EXCEPTION_GENERAL_PROTECTION 13

General Protection.

Definition at line 76 of file irq.h.

4.16.1.7 #define EXCEPTION_INVALID_OP 6

Invalid Opcode (Undefined Opcode)

Definition at line 58 of file irq.h.

4.16.1.8 #define EXCEPTION_INVALID_TSS 10

Invalid TSS.

Definition at line 67 of file irq.h.

4.16.1.9 #define EXCEPTION_MACHINE_CHECK 18

Machine Check.

Definition at line 88 of file irq.h.

4.16.1.10 #define EXCEPTION_MATH 16

x87 FPU Floating-Point Error (Math Fault)

Definition at line 82 of file irq.h.

4.16.1.11 #define EXCEPTION_NMI 2

NMI Interrupt.

Definition at line 46 of file irq.h.

4.16.1.12 #define EXCEPTION_NO_COPROC 7

Device Not Available (No Math Coprocessor)

Definition at line 61 of file irq.h.

4.16.1.13 #define EXCEPTION_OVERFLOW 4

Overflow.

Definition at line 52 of file irq.h.

4.16.1.14 #define EXCEPTION_PAGE_FAULT 14

Page Fault.

Definition at line 79 of file irq.h.

4.16.1.15 #define EXCEPTION_SEGMENT_NOT_PRESENT 11

Segment Not Present.

Definition at line 70 of file irq.h.

4.16.1.16 #define EXCEPTION_SIMD 19

SIMD Floating-Point Exception.

Definition at line 91 of file irq.h.

4.16.1.17 #define EXCEPTION_STACK_SEGMENT 12

Stack-Segment Fault.

Definition at line 73 of file irq.h.

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

Definition at line 93 of file irq.h.

4.16.1.19 #define IDT_FIRST_IRQ 32

Definition at line 37 of file irq.h.

Referenced by dispatch_interrupt().

4.16.1.20 #define IDT_IRQ_COUNT (IDT_VECTOR_COUNT - IDT_FIRST_IRQ)

Definition at line 39 of file irq.h.

4.16.1.21 #define IDT_VECTOR_COUNT 256

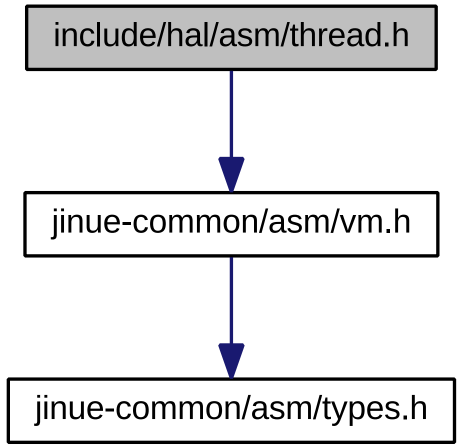
Definition at line 35 of file irq.h.

Referenced by hal_init().

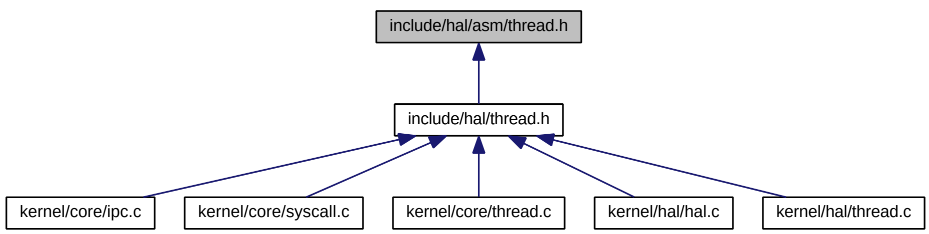
4.17 include/hal/asm/thread.h File Reference

`#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))

4.17.1 Macro Definition Documentation

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

Definition at line 40 of file thread.h.

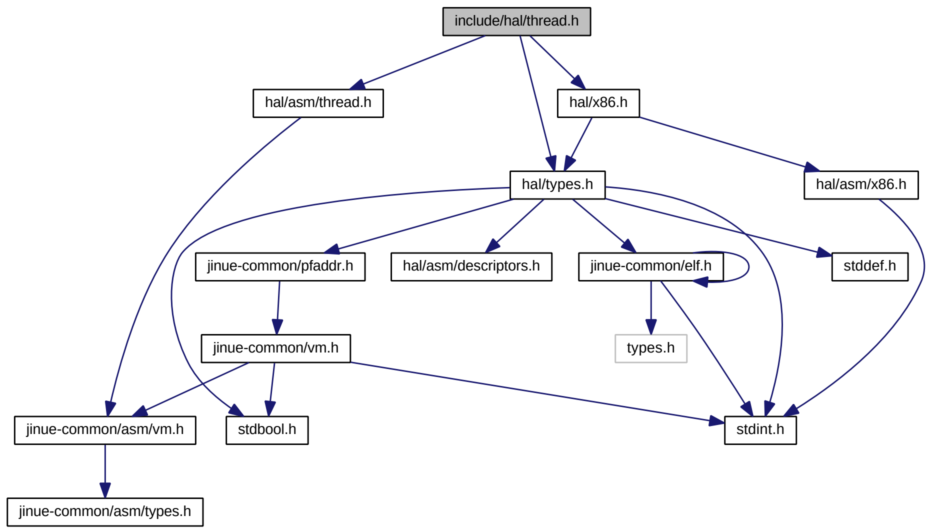
4.17.1.2 #define THREAD_CONTEXT_SIZE PAGE_SIZE

Definition at line 38 of file thread.h.

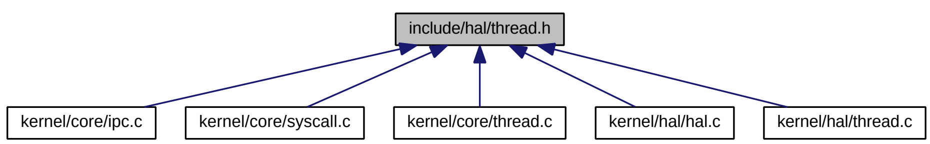
4.18 include/hal/thread.h File Reference

```
#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_create** (**addr_t** entry, **addr_t** user_stack)
- void **thread_page_destroy** (**thread_t** ∗thread)
- void **thread_context_switch** (**thread_context_t** ∗from_ctx, **thread_context_t** ∗to_ctx, **bool** destroy_from)

4.18.1 Function Documentation

4.18.1.1 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

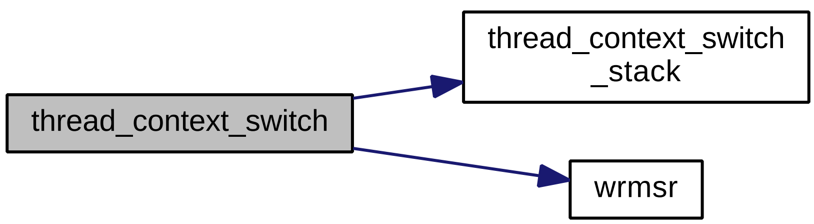
Definition at line 145 of file thread.c.

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:



4.18.1.2 thread_t* thread_page_create (addr_t *entry*, addr_t *user_stack*)

Definition at line 85 of file thread.c.

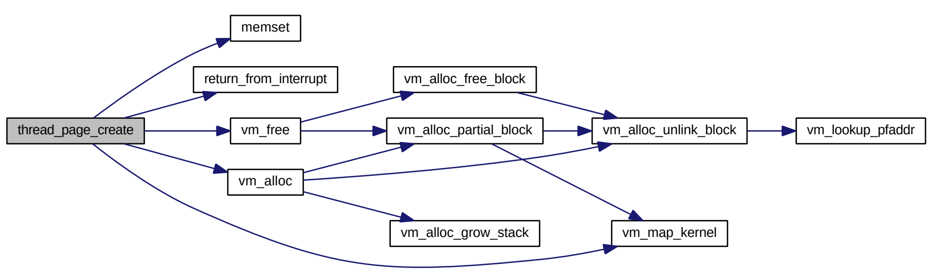
References trapframe_t::cs, trapframe_t::ds, trapframe_t::eflags, trapframe_t::eip, kernel_context_t::eip, trapframe_t::es, trapframe_t::esp, trapframe_t::fs, GDT_USER_CODE, GDT_USER_DATA, global_page_allocator, trapframe_t::gs, thread_context_t::local_storage_addr, memset(), NULL, pfalloc, PFNULL, return_from_interrupt(), RPL_USER, thread_context_t::saved_stack_pointer, SEG_SELECTOR, trapframe_t::ss, vm_alloc(), VM_FLAG_READ_WRITE, vm_free(), and vm_map_kernel().

Referenced by thread_create().

```
                                    {

    /* allocate thread context */
    thread_t *thread = (thread_t *)vm_alloc( global_page_allocator );

    if(thread != NULL) {
        pfaddr_t pf = pfalloc();

        if(pf == PFNULL) {
            vm_free(global_page_allocator, (addr_t)thread);
            return NULL;
        }

        vm_map_kernel((addr_t)thread, pf, VM_FLAG_READ_WRITE);

        /* initialize fields */
        thread_context_t *thread_ctx = &thread->thread_ctx;

```

```
105         thread_ctx->local_storage_addr  = NULL;
106 
107         /* setup stack for initial return to user space */
108         void *kernel_stack_base = get_kernel_stack_base(thread_ctx);
109 
110         trapframe_t *trapframe = (trapframe_t *)kernel_stack_base - 1;
111 
112         memset(trapframe, 0, sizeof(trapframe_t));
113 
114         trapframe->eip      = (uint32_t)entry;
115         trapframe->esp      = (uint32_t)user_stack;
116         trapframe->eflags   = 2;
117         trapframe->cs       = SEG_SELECTOR(GDT_USER_CODE, RPL_USER);
118         trapframe->ss       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);
119         trapframe->ds       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);
120         trapframe->es       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);
121         trapframe->fs       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);
122         trapframe->gs       = SEG_SELECTOR(GDT_USER_DATA, RPL_USER);
123 
124         kernel_context_t *kernel_context = (kernel_context_t *)trapframe - 1;
125 
126         memset(kernel_context, 0, sizeof(kernel_context_t));
127 
128         /* This is the address to which thread_context_switch_stack() will return. */
129         kernel_context->eip = (uint32_t)return_from_interrupt;
130 
131         /* set thread stack pointer */
132         thread_ctx->saved_stack_pointer = (addr_t)kernel_context;
133     }
134 
135     return thread;
136 }
```

Here is the call graph for this function:



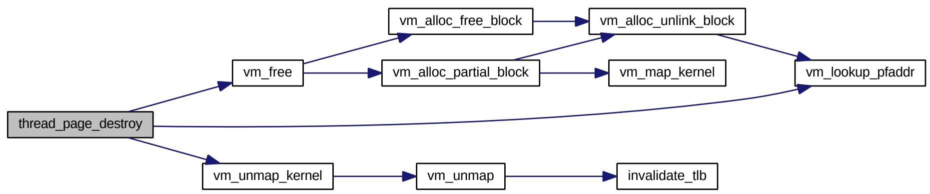
4.18.1.3 void thread_page_destroy (thread_t * *thread*)

Definition at line 138 of file thread.c.

References global_page_allocator, NULL, pffree, vm_free(), vm_lookup_pfaddr(), and vm_unmap_kernel().

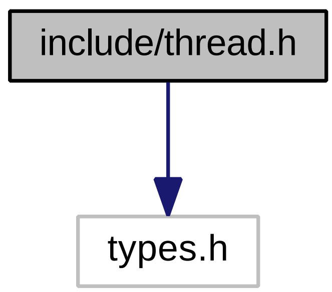
```
138                                         {
139     pfaddr_t pfaddr = vm_lookup_pfaddr(NULL, (addr_t)thread);
140     vm_unmap_kernel((addr_t)thread);
141     vm_free(global_page_allocator, (addr_t)thread);
142     pffree(pfaddr);
143 }
```

Here is the call graph for this function:

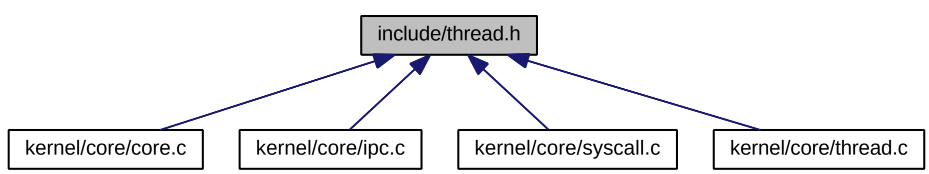


4.19 include/thread.h File Reference

`#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)
- 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)

4.19.1 Function Documentation

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

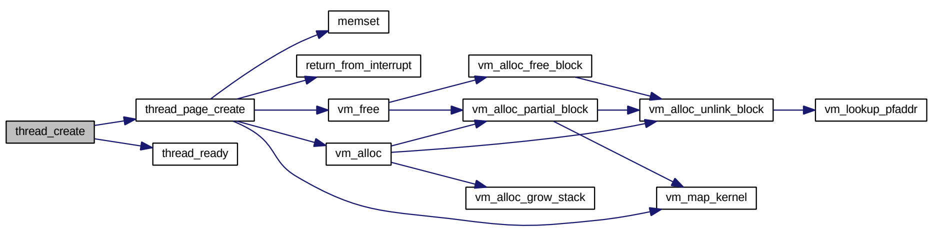
Definition at line 42 of file thread.c.

References thread_t::header, NULL, OBJECT_TYPE_THREAD, thread_t::process, thread_t::sender, thread_t::thread-_list, thread_page_create(), and thread_ready().

Referenced by dispatch_syscall(), and kmain().

```
45                                       {
46 
47     thread_t *thread = thread_page_create(entry, user_stack);
48 
49     if(thread != NULL) {
50         object_header_init(&thread->header, OBJECT_TYPE_THREAD);
51 
52         jinue_node_init(&thread->thread_list);
53 
54         thread->process     = process;
55         thread->sender      = NULL;
56 
57         thread_ready(thread);
58     }
59 
60     return thread;
61 }
```

Here is the call graph for this function:



4.19.1.2 void thread_ready (thread_t * *thread*)

Definition at line 63 of file thread.c.

References thread_t::thread_list.

Referenced by thread_create(), and thread_switch().

```
63                                      {
64     /* add thread to the tail of the ready list to give other threads a chance
65      * to run */
66     jinue_list_enqueue(&ready_list, &thread->thread_list);
67 }
```

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

Definition at line 69 of file thread.c.

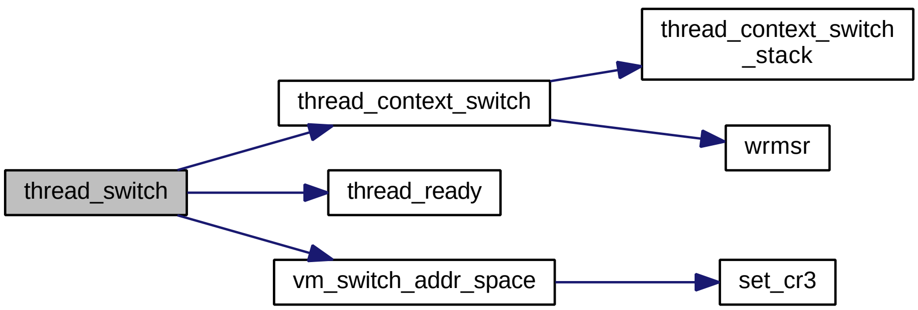
References process_t::addr_space, NULL, thread_t::process, thread_context_switch(), thread_t::thread_ctx, thread_ready(), and vm_switch_addr_space().

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

```
73                           {
74 
75     if(to_thread != from_thread) {
76         thread_context_t    *from_context;
77         process_t           *from_process;
78 
79         if(from_thread == NULL) {
80             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);
        }

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

Here is the call graph for this function:



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

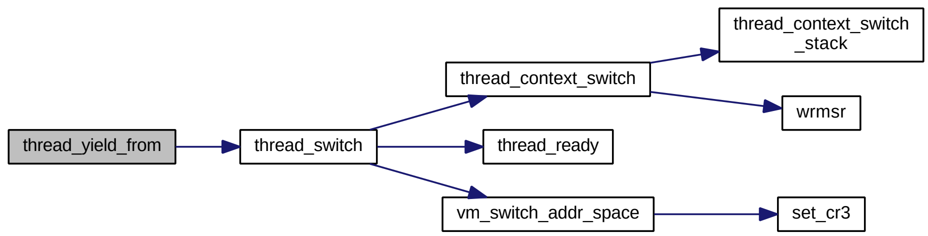
Definition at line 130 of file thread.c.

References thread_switch().

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

```
130                                                                             {
131     bool from_can_run = ! (blocked || do_destroy);
132 
133     thread_switch(
134             from_thread,
135             reschedule(from_thread, from_can_run),
136             blocked,
137             do_destroy);
138 }
```

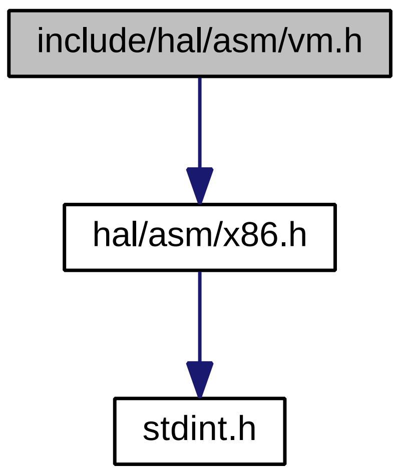
Here is the call graph for this function:



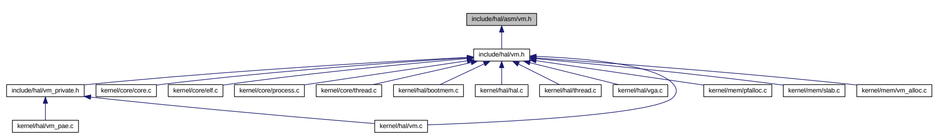
4.20 include/hal/asm/vm.h File Reference

`#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

4.20.1 Macro Definition Documentation

4.20.1.1 #define VM_FLAG_ACCESSED X86_PTE_ACCESSED

page was accessed (read)

Definition at line 53 of file vm.h.

4.20.1.2 #define VM_FLAG_DIRTY X86_PTE_DIRTY

page was written to

Definition at line 56 of file vm.h.

4.20.1.3 #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().

4.20.1.4 #define VM_FLAG_PRESENT X86_PTE_PRESENT

page is present in memory

Definition at line 38 of file vm.h.

4.20.1.5 #define VM_FLAG_READ_ONLY 0

page is read only

Definition at line 41 of file vm.h.

Referenced by elf_load().

4.20.1.6 #define VM_FLAG_READ_WRITE X86_PTE_READ_WRITE

page is read/write accessible

Definition at line 44 of file vm.h.

Referenced by elf_load(), elf_setup_stack(), slab_cache_grow(), thread_page_create(), vm_alloc_init_allocator(), vm_alloc_partial_block(), vm_allocate_page_directory(), vm_boot_init(), vm_clone_page_directory(), and vm_destroy_page_directory().

4.20.1.7 #define VM_FLAG_USER X86_PTE_USER

user mode page

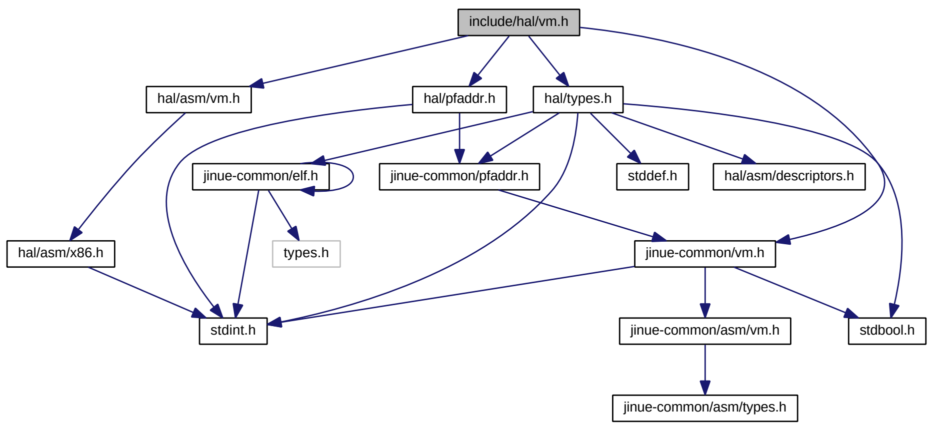
Definition at line 50 of file vm.h.

Referenced by vm_map_user().

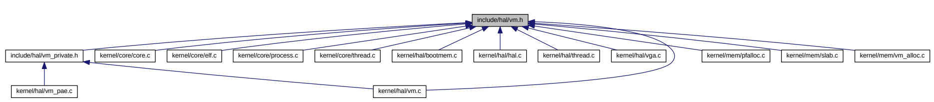
4.21 include/hal/vm.h File Reference

```
#include <hal/asm/vm.h>
#include <jinue-common/vm.h>
#include <hal/pfaddr.h>
#include <hal/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. 330) 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_PFADDR**(x) ((**pfaddr_t**)((**EARLY_VIRT_TO_PHYS**(x) >> **PFADDR_SHIFT**)))

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

- #define **ADDR_4GB UINT64_C**(0x100000000)

Functions

- void **vm_boot_init** (void)
- void **vm_map_kernel** (**addr_t** vaddr, **pfaddr_t** paddr, int flags)
- void **vm_map_user** (**addr_space_t** ∗addr_space, **addr_t** vaddr, **pfaddr_t** paddr, int flags)
- void **vm_unmap_kernel** (**addr_t** addr)
- void **vm_unmap_user** (**addr_space_t** ∗addr_space, **addr_t** addr)
- **pfaddr_t vm_lookup_pfaddr** (**addr_space_t** ∗addr_space, **addr_t** addr)
- void **vm_change_flags** (**addr_space_t** ∗addr_space, **addr_t** addr, int flags)
- void **vm_map_early** (**addr_t** vaddr, **pfaddr_t** paddr, int flags)
- **addr_space_t** ∗ **vm_create_addr_space** (**addr_space_t** ∗addr_space)
- **addr_space_t** ∗ **vm_create_initial_addr_space** (void)
- void **vm_destroy_addr_space** (**addr_space_t** ∗addr_space)
- void **vm_switch_addr_space** (**addr_space_t** ∗addr_space)

4.21.1 Macro Definition Documentation

4.21.1.1 #define ADDR_4GB UINT64_C(0x100000000)

Definition at line 53 of file vm.h.

Referenced by bootmem_init().

4.21.1.2 #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. 330) 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 45 of file vm.h.

4.21.1.3 #define EARLY_PTR_TO_PFADDR(*x*) ((pfaddr_t)((EARLY_VIRT_TO_PHYS(x) >> PFADDR_SHIFT)))

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

Definition at line 51 of file vm.h.

Referenced by elf_load(), hal_init(), vm_allocate_page_directory(), vm_boot_init(), and vm_x86_create_initial_addr_-space().

4.21.1.4 #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 48 of file vm.h.

Referenced by vm_map_early(), and vm_x86_create_initial_addr_space().

4.21.2 Function Documentation

4.21.2.1 void vm_boot_init (void)

below this point, it is no longer safe to call **pfalloc_early()** (p. 211)

Definition at line 87 of file vm.c.

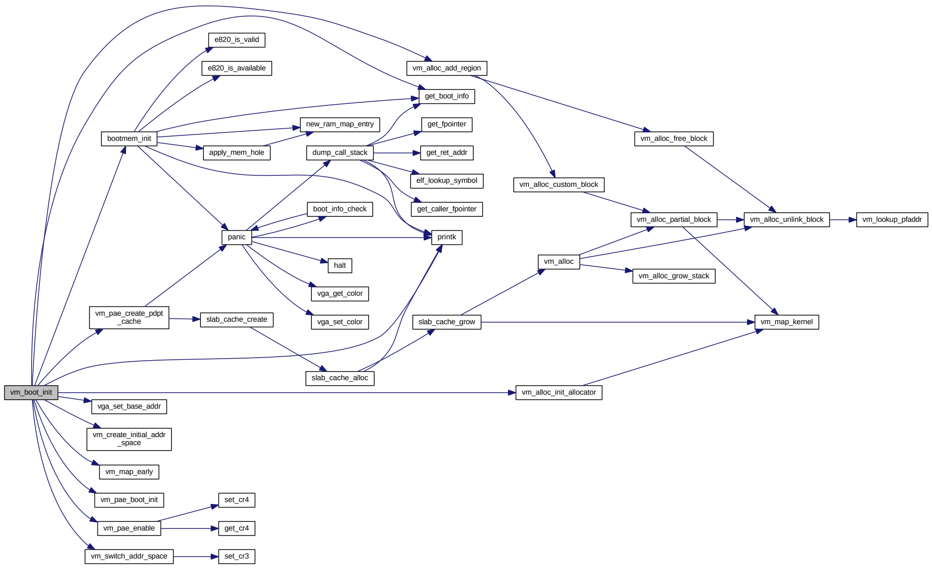
References ADDR_TO_PFADDR, boot_info, bootmem_init(), CPU_FEATURE_PAE, EARLY_PTR_TO_PFADDR, get-_boot_info(), boot_info_t::image_start, kernel_region_top, KLIMIT, MB, PAGE_SIZE, printk(), use_pfalloc_early, vga-_set_base_addr(), VGA_TEXT_VID_BASE, VGA_TEXT_VID_TOP, vm_alloc_add_region(), vm_alloc_init_allocator(), vm_create_initial_addr_space(), VM_FLAG_KERNEL, VM_FLAG_READ_WRITE, vm_map_early(), vm_pae_boot_-init(), vm_pae_create_pdpt_cache(), vm_pae_enable(), and vm_switch_addr_space().

Referenced by hal_init().

```
87                          {
88      bool             use_pae;
89      addr_t           addr;
90      addr_space_t    *addr_space;
91 
92      if(cpu_has_feature(CPU_FEATURE_PAE)) {
93          printk("Enabling Physical Address Extension (PAE).\n");
94          vm_pae_boot_init();
95 
96          use_pae = true;
97      }
98      else {
99          use_pae = false;
100     }
101 
102     /* create initial address space */
103     addr_space = vm_create_initial_addr_space();
104 
106     use_pfalloc_early = false;
107 
108     /* create system usable physical memory (RAM) map
109      *
110      * Among other things, this function marks the memory used by the kernel
111      * (i.e. image_start..kernel_region_top) as in use. This must be done after
112      * all early page frame allocations with fpalloc_early() have been done.
113      *
114      * This function needs to know whether Physical Address Extension (PAE) is
115      * enabled (use_pae) because, if it isn't, all memory above the 4GB mark is
116      * excluded from the usable memory map. */
117     bootmem_init(use_pae);
118 
119     /* perform 1:1 mapping of kernel image and data
120 
121        note: page tables for memory region (0..KLIMIT) are contiguous in
122        physical memory */
123     const boot_info_t *boot_info = get_boot_info();
124 
125     for(addr = (addr_t)boot_info->image_start; addr < kernel_region_top; addr +=
      PAGE_SIZE) {
126         vm_map_early((addr_t)addr, EARLY_PTR_TO_PFADDR(addr), VM_FLAG_KERNEL |
      VM_FLAG_READ_WRITE);
127     }
128 
129     /* map VGA text buffer in the new address space
130      *
131      * This is a good place to do this because:
132      *
133      * 1) It is our last chance to allocate a continuous region of virtual memory.
134      *    Once the page allocator is initialized (see call to vm_alloc_init_allocator()
135      *    below) and we start using vm_alloc() to allocate memory, pages can only
136      *    be allocated one at a time.
137      *
138      * 2) Doing this last makes things simpler because this is the only place where
139      *    we have to allocate a continuous region of virtual memory but no physical
140      *    memory to back it. To allocate it, we just have to increase kernel_vm_top,
141      *    which represents the end of the virtual memory region that is used by the
142      *    kernel. */
143     addr_t kernel_vm_top = kernel_region_top;
144     addr = (addr_t)VGA_TEXT_VID_BASE;
145 
```

```c
    addr_t vga_text_base = kernel_vm_top;

    while(addr < (addr_t)VGA_TEXT_VID_TOP) {
        vm_map_early(kernel_vm_top, ADDR_TO_PFADDR((uintptr_t)addr), VM_FLAG_KERNEL | VM_FLAG_READ_WRITE);
        kernel_vm_top   += PAGE_SIZE;
        addr            += 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", kernel_vm_top);

    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.
         *
         * From Intel 64 and IA-32 Architectures Software Developer's Manual
         * Volume 3: System Programming Guide, section 4.4.1 "PDPTE Registers":
         *
         *  " The logical processor loads [the PDPTE] registers from the PDPTEs
         *    in memory as part of certain operations:
         *      * If PAE paging would be in use following an execution of MOV to
         *        CR0 or MOV to CR4 (see Section 4.1.1) and the instruction is
         *        modifying any of (...) CR4.PAE, (...); then the PDPTEs are
         *        loaded from the address in CR3. "
         *
         * There are bootstrapping issues when enabling PAE while paging is enabled.
         * See the comment at the top of the vm_pae_create_initial_addr_space()
         * function in vm_pae.c for more detail. */
        vm_pae_enable();
    }

    /* switch to new address space */
    vm_switch_addr_space(addr_space);

    /* initialize global page allocator (region starting at KLIMIT)
     *
     * TODO Some work needs to be done in the page allocator to support allocating
     * up to the top of memory (i.e. 0x100000000, which cannot be represented on
     * 32 bits). In the mean time, we leave a 4MB gap. */
    global_page_allocator = &__global_page_allocator;
    vm_alloc_init_allocator(global_page_allocator, (addr_t)KLIMIT, (addr_t)0 - 4 * MB);

    vm_alloc_add_region(global_page_allocator, (addr_t)KLIMIT,          (addr_t)boot_info->image_start);
    vm_alloc_add_region(global_page_allocator, (addr_t)kernel_vm_top,   (addr_t)0 - 4 * MB);

    /* 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();
    }
}
```

Here is the call graph for this function:



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

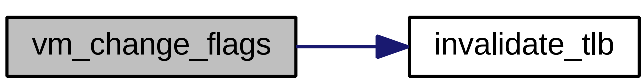
ASSERTION: there is a page table entry marked present for this address

Definition at line 444 of file vm.c.

References assert, get_pte_flags, invalidate_tlb(), NULL, and set_pte_flags.

```
444                                                                           {
445     pte_t *pte = vm_lookup_page_table_entry(addr_space, addr, false);
446
448     assert(pte != NULL && (get_pte_flags(pte) & VM_FLAG_PRESENT));
449
450     /* perform the flags change */
451     set_pte_flags(pte, flags | VM_FLAG_PRESENT);
452
453     vm_free_page_table_entry(addr, pte);
454
455     /* invalidate TLB entry for the affected page */
456     invalidate_tlb(addr);
457 }
```

Here is the call graph for this function:



4.21.2.3 **addr_space_t**∗ vm_create_addr_space (**addr_space_t** ∗ ***addr_space***)

Definition at line 520 of file vm.c.

References create_addr_space.

Referenced by process_create().

```
520                                                               {
521     return create_addr_space(addr_space);
522 }
```

4.21.2.4 addr_space_t* vm_create_initial_addr_space (void)

Definition at line 577 of file vm.c.

References create_initial_addr_space.

Referenced by vm_boot_init().

```
577                                              {
578     return create_initial_addr_space();
579 }
```

4.21.2.5 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 609 of file vm.c.

References assert, destroy_addr_space, and NULL.

```
609                                                  {
611     assert(addr_space != NULL);
612
614     assert(addr_space != &initial_addr_space);
615
617     assert( addr_space != get_current_addr_space() );
618
619     destroy_addr_space(addr_space);
620 }
```

4.21.2.6 pfaddr_t vm_lookup_pfaddr (addr_space_t * *addr_space,* addr_t *addr*)

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

Definition at line 431 of file vm.c.

References assert, get_pte_flags, get_pte_pfaddr, and NULL.

Referenced by thread_page_destroy(), vm_alloc_destroy(), and vm_alloc_unlink_block().

```
431                                                                    {
432     pte_t *pte = vm_lookup_page_table_entry(addr_space, addr, false);
433
435     assert(pte != NULL && (get_pte_flags(pte) & VM_FLAG_PRESENT));
436
437     pfaddr_t pfaddr = get_pte_pfaddr(pte);
438
439     vm_free_page_table_entry(addr, pte);
440
441     return pfaddr;
442 }
```

4.21.2.7 void vm_map_early (addr_t *vaddr*, pfaddr_t *paddr*, int *flags*)

ASSERTION: we are mapping in the kernel region

ASSERTION: we assume vaddr is aligned on a page boundary

Definition at line 459 of file vm.c.

References assert, EARLY_VIRT_TO_PHYS, get_pte_with_offset, page_number_of, page_offset_of, and set_pte.

Referenced by vm_boot_init().

```
459                                                     {
460     pte_t *pte;
461
463     assert( is_fast_map_pointer(vaddr) );
464
466     assert( page_offset_of(vaddr) == 0 );
467
468     pte = get_pte_with_offset(global_page_tables, page_number_of(
       EARLY_VIRT_TO_PHYS((uintptr_t)vaddr) ));
469     set_pte(pte, paddr, flags | VM_FLAG_PRESENT);
470 }
```

4.21.2.8 void vm_map_kernel (addr_t *vaddr*, pfaddr_t *paddr*, int *flags*)

Definition at line 415 of file vm.c.

References NULL, and VM_FLAG_KERNEL.

Referenced by elf_load(), elf_setup_stack(), slab_cache_grow(), thread_page_create(), vm_alloc_init_allocator(), vm_-alloc_partial_block(), vm_clone_page_directory(), and vm_destroy_page_directory().

```
415                                                      {
416     vm_map(NULL, vaddr, paddr, flags | VM_FLAG_KERNEL);
417 }
```

4.21.2.9 void vm_map_user (addr_space_t ∗ *addr_space*, addr_t *vaddr*, pfaddr_t *paddr*, int *flags*)

Definition at line 419 of file vm.c.

References VM_FLAG_USER.

Referenced by elf_load(), and elf_setup_stack().

```
419                                                                                 {
420     vm_map(addr_space, vaddr, paddr, flags | VM_FLAG_USER);
421 }
```

4.21.2.10 void vm_switch_addr_space (addr_space_t ∗ *addr_space*)

Definition at line 622 of file vm.c.

References addr_space_t::cr3, and set_cr3().

Referenced by thread_switch(), and vm_boot_init().

```
622                                                 {
623     set_cr3(addr_space->cr3);
624
625     get_cpu_local_data()->current_addr_space = addr_space;
626 }
```

Here is the call graph for this function:

4.21.2.11 void vm_unmap_kernel (**addr_t** *addr*)

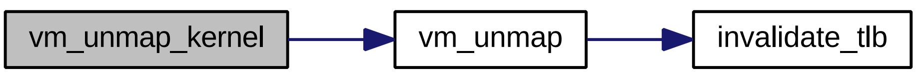
Definition at line 423 of file vm.c.

References NULL, and vm_unmap().

Referenced by elf_load(), elf_setup_stack(), thread_page_destroy(), vm_clone_page_directory(), and vm_destroy_-page_directory().

```
                                {
    vm_unmap(NULL, addr);
}
```

Here is the call graph for this function:



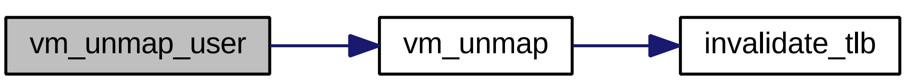
4.21.2.12 void vm_unmap_user (**addr_space_t** ∗ *addr_space,* **addr_t** *addr*)

Definition at line 427 of file vm.c.

References vm_unmap().

```
                                                   {
    vm_unmap(addr_space, addr);
}
```

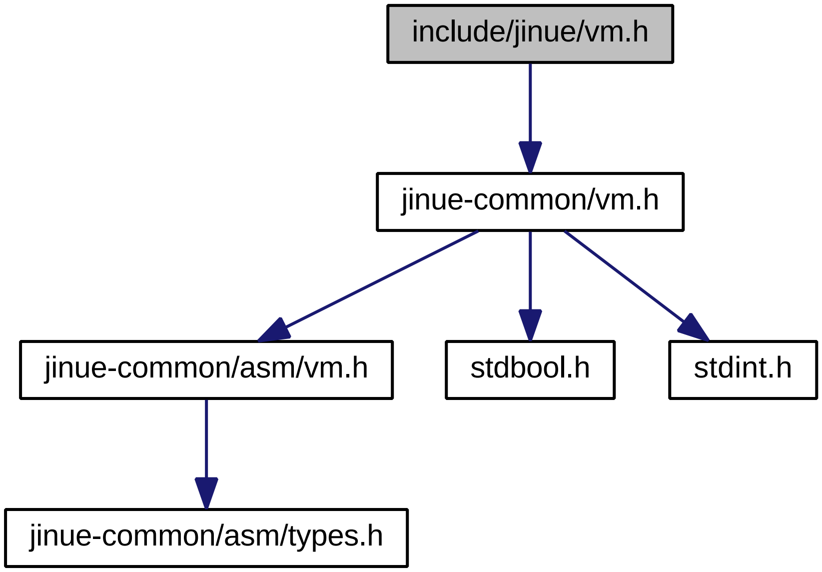
Here is the call graph for this function:



4.22 include/jinue/vm.h File Reference

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

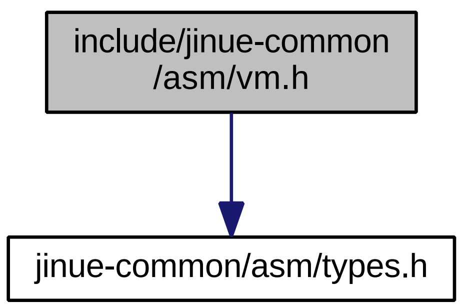
Include dependency graph for vm.h:



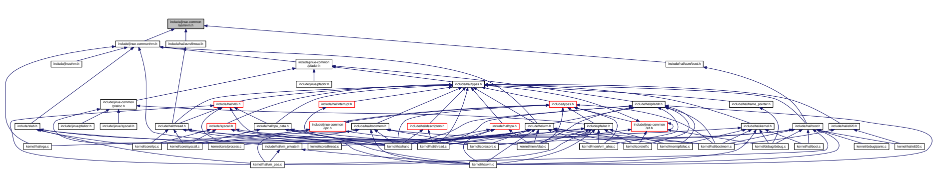
4.23 include/jinue-common/asm/vm.h File Reference

`#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** 0xe0000000

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

4.23.1 Macro Definition Documentation

4.23.1.1 #define KLIMIT 0xe0000000

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 4MB boundary.

Definition at line 50 of file vm.h.

Referenced by vm_boot_init(), and vm_x86_create_initial_addr_space().

4.23.1.2 #define PAGE_BITS 12

number of bits in virtual address for offset inside page

Definition at line 39 of file vm.h.

4.23.1.3 #define PAGE_MASK (PAGE_SIZE - 1)

bit mask for offset in page

Definition at line 45 of file vm.h.

Referenced by apply_mem_hole(), bootmem_init(), and elf_load().

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

size of page

Definition at line 42 of file vm.h.

Referenced by apply_mem_hole(), bootmem_init(), elf_load(), elf_setup_stack(), hal_init(), pfalloc_early(), vm_alloc_custom_block(), vm_alloc_destroy(), vm_alloc_grow_single(), vm_alloc_grow_stack(), vm_alloc_init_allocator(), vm_alloc_partial_block(), and vm_boot_init().

4.23.1.5 #define STACK_BASE KLIMIT

stack base address (stack top)

Definition at line 53 of file vm.h.

Referenced by elf_setup_stack().

4.23.1.6 #define STACK_SIZE (8 ∗ PAGE_SIZE)

initial stack size

Definition at line 56 of file vm.h.

4.23.1.7 #define STACK_START (STACK_BASE - STACK_SIZE)

initial stack lower address

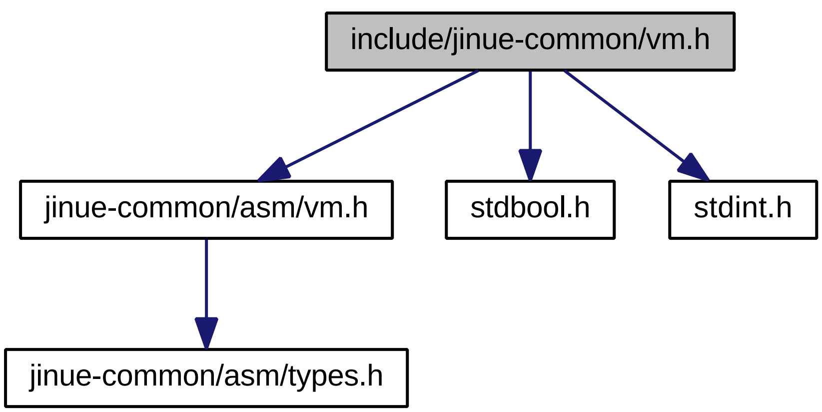
Definition at line 59 of file vm.h.

Referenced by elf_setup_stack().

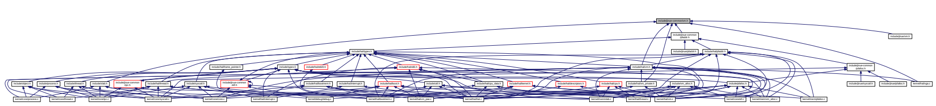
4.24 include/jinue-common/vm.h File Reference

```
#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

4.24.1 Macro Definition Documentation

4.24.1.1 #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.

4.24.1.2 #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().

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

byte offset in page of virtual (linear) address

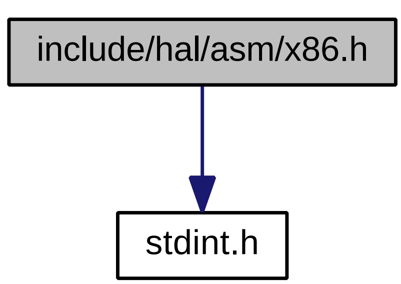
Definition at line 42 of file vm.h.

Referenced by elf_load(), hal_init(), vm_alloc_custom_block(), vm_alloc_init_allocator(), vm_free(), vm_map_early(), and vm_unmap().

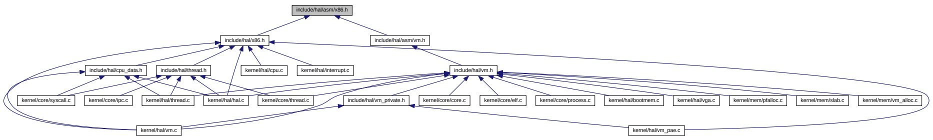
4.25 include/hal/asm/x86.h File Reference

```
#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

4.25.1 Macro Definition Documentation

4.25.1.1 #define X86_CR0_PG (1<<31)

CR0 register: Paging.

Definition at line 43 of file x86.h.

4.25.1.2 #define X86_CR0_WP (1<<16)

CR0 register: Write Protect.

Definition at line 40 of file x86.h.

4.25.1.3 #define X86_CR4_PAE (1<<5)

CR4 register: Physical Address Extension (PAE)

Definition at line 50 of file x86.h.

Referenced by vm_pae_enable().

4.25.1.4 #define X86_CR4_PGE (1<<7)

CR4 register: global pages.

Definition at line 53 of file x86.h.

4.25.1.5 #define X86_CR4_PSE (1<<4)

CR4 register: Page Size Extension (PSE)

Definition at line 47 of file x86.h.

4.25.1.6 #define X86_PDE_PAGE_SIZE (1<< 7)

page directory entry describes a 4M page

Definition at line 78 of file x86.h.

4.25.1.7 #define X86_PTE_ACCESSED (1<< 5)

page was accessed (read)

Definition at line 72 of file x86.h.

4.25.1.8 #define X86_PTE_CACHE_DISABLE (1<< 4)

uncached page

Definition at line 69 of file x86.h.

4.25.1.9 #define X86_PTE_DIRTY (1<< 6)

page was written to

Definition at line 75 of file x86.h.

4.25.1.10 #define X86_PTE_GLOBAL (1<< 8)

page is global (mapped in every address space)

Definition at line 81 of file x86.h.

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

do not execute bit

Definition at line 84 of file x86.h.

4.25.1.12 #define X86_PTE_PRESENT (1<< 0)

page is present in memory

Definition at line 57 of file x86.h.

4.25.1.13 #define X86_PTE_READ_WRITE (1<< 1)

page is read/write accessible

Definition at line 60 of file x86.h.

4.25.1.14 #define X86_PTE_USER (1<< 2)

user mode page

Definition at line 63 of file x86.h.

4.25.1.15 #define X86_PTE_WRITE_THROUGH (1<< 3)

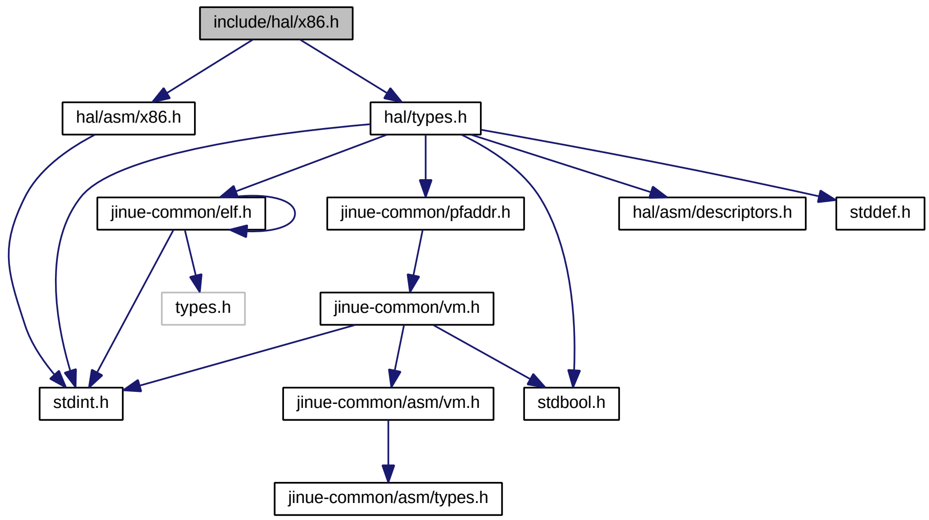
write-through cache policy for page

Definition at line 66 of file x86.h.

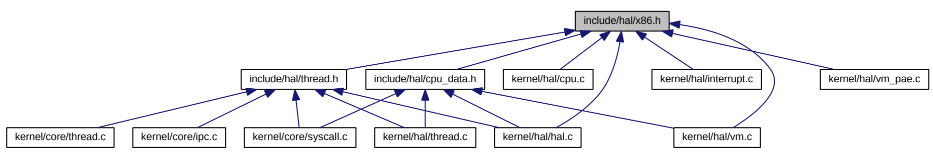
4.26 include/hal/x86.h File Reference

```
#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_cr1** (void)
- **uint32_t get_cr2** (void)

- **uint32_t get_cr3** (void)
- **uint32_t get_cr4** (void)
- void **set_cr0** (**uint32_t** val)
- void **set_cr1** (**uint32_t** val)
- void **set_cr2** (**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)
- void **set_data_segments** (**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)

4.26.1 Typedef Documentation

4.26.1.1 typedef uint32_t msr_addr_t

Definition at line 46 of file x86.h.

4.26.2 Function Documentation

4.26.2.1 void cli (void)

4.26.2.2 uint32_t cpuid (x86_cpuid_regs_t * *regs*)

Referenced by cpu_detect_features().

4.26.2.3 uint32_t get_cr0 (void)

4.26.2.4 uint32_t get_cr1 (void)

4.26.2.5 uint32_t get_cr2 (void)

Referenced by dispatch_interrupt().

4.26.2.6 uint32_t get_cr3 (void)

4.26.2.7 uint32_t get_cr4 (void)

Referenced by vm_pae_enable().

4.26.2.8 uint32_t get_eflags (void)

Referenced by cpu_detect_features().

4.26.2.9 uint32_t get_esp (void)

4.26.2.10 uint32_t get_gs_ptr (uint32_t * *ptr*)

4.26.2.11 void invalidate_tlb (addr_t *vaddr*)

Referenced by vm_change_flags(), and vm_unmap().

4.26.2.12 void lgdt (pseudo_descriptor_t * *gdt_info*)

Referenced by hal_init().

4.26.2.13 void lidt (pseudo_descriptor_t * *idt_info*)

Referenced by hal_init().

4.26.2.14 void ltr (seg_selector_t *sel*)

Referenced by hal_init().

4.26.2.15 uint64_t rdmsr (msr_addr_t *addr*)

Referenced by hal_init().

4.26.2.16 void set_cr0 (uint32_t *val*)

4.26.2.17 void set_cr1 (uint32_t *val*)

4.26.2.18 void set_cr2 (uint32_t *val*)

4.26.2.19 void set_cr3 (uint32_t *val*)

Referenced by vm_switch_addr_space().

4.26.2.20 void set_cr4 (uint32_t *val*)

Referenced by vm_pae_enable().

4.26.2.21 void set_cs (uint32_t *val*)

Referenced by hal_init().

4.26.2.22 void set_data_segments (uint32_t *val*)

Referenced by hal_init().

4.26.2.23 void set_ds (uint32_t *val*)

4.26.2.24 void set_eflags (uint32_t *val*)

Referenced by cpu_detect_features().

4.26.2.25 void set_es (uint32_t *val*)

4.26.2.26 void set_fs (uint32_t *val*)

4.26.2.27 void set_gs (uint32_t *val*)

Referenced by hal_init().

4.26.2.28 void set_ss (uint32_t *val*)

Referenced by hal_init().

4.26.2.29 void sti (void)

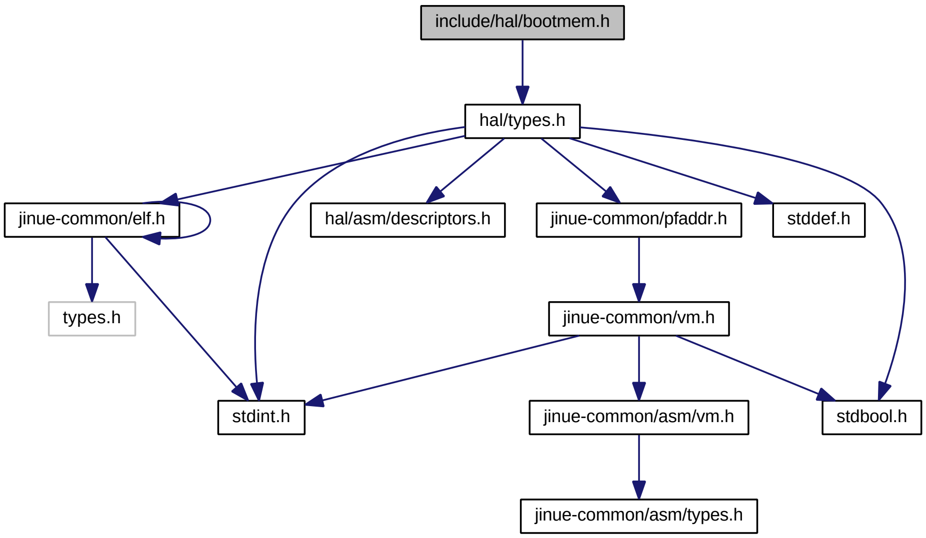
4.26.2.30 void wrmsr (msr_addr_t *addr,* uint64_t *val*)

Referenced by hal_init(), and thread_context_switch().

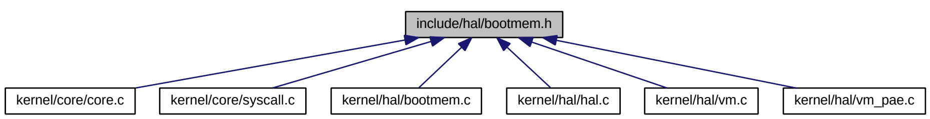
4.27 include/hal/bootmem.h File Reference

`#include <hal/types.h>`

Include dependency graph for bootmem.h:



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



Data Structures

- struct **bootmem_t**

Typedefs

- typedef struct **bootmem_t bootmem_t**

Functions

- void **new_ram_map_entry** (**pfaddr_t** addr, **uint32_t** count, **bootmem_t** ∗∗head)
- void **apply_mem_hole** (**e820_addr_t** hole_start, **e820_addr_t** hole_end, **bootmem_t** ∗∗head)
- void **bootmem_init** (**bool** use_pae)
- **bootmem_t** ∗ **bootmem_get_block** (void)

Variables

- **bootmem_t** ∗ **ram_map**

kernel memory map

- **bootmem_t** ∗ **bootmem_root**

  available memory map (allocator)

- void ∗ **boot_heap**

  current top of boot heap

4.27.1 Typedef Documentation

4.27.1.1 typedef struct bootmem_t bootmem_t

Definition at line 44 of file bootmem.h.

4.27.2 Function Documentation

4.27.2.1 void apply_mem_hole (e820_addr_t *hole_start,* e820_addr_t *hole_end,* bootmem_t ∗∗ *head*)

Definition at line 68 of file bootmem.c.

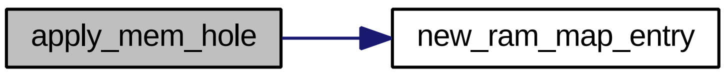
References bootmem_t::addr, bootmem_t::count, new_ram_map_entry(), bootmem_t::next, NULL, OFFSET_OF, PAGE_MASK, PAGE_SIZE, and PFADDR_SHIFT.

Referenced by bootmem_init().

```
68                                                                            {
69      bootmem_t *ptr, **dptr;
70      pfaddr_t addr, top;
71      pfaddr_t hole_addr, hole_top;
72
73      hole_addr = hole_start >> PFADDR_SHIFT;
74      hole_top  = hole_end   >> PFADDR_SHIFT;
75
76      /* align on page boundaries */
77      if( OFFSET_OF(hole_start, PAGE_SIZE) != 0 ) {
78          hole_addr = (hole_addr & (e820_addr_t)~(PAGE_MASK >> PFADDR_SHIFT));
79      }
80
81      if( OFFSET_OF(hole_end, PAGE_SIZE) != 0 ) {
82          hole_top = (hole_top & (e820_addr_t)~(PAGE_MASK >> PFADDR_SHIFT)) + (
        PAGE_SIZE >> PFADDR_SHIFT);
83      }
84
85      /* apply hole to all available memory blocks */
86      for(dptr = head, ptr = *head; ptr != NULL; dptr = &ptr->next, ptr = ptr->
        next) {
87          addr  = ptr->addr;
88          top   = addr + ptr->count * (PAGE_SIZE >> PFADDR_SHIFT);
89
90          /* case where the block is completely inside the hole */
91          if(addr >= hole_addr && top <= hole_top) {
92              /* remove this block */
93              *dptr = ptr->next;
94
95              return;
96          }
97
98          /* case where the block must be split in two because the hole is
99           * inside it */
100          if(addr < hole_addr && top > hole_top) {
101              /* first block: below the hole */
102              ptr->count = (hole_addr - addr) / (PAGE_SIZE >> PFADDR_SHIFT);
103
104              /* second block: above the hole */
105              new_ram_map_entry(hole_top, (top - hole_top) / (PAGE_SIZE >> PFADDR_SHIFT), head);
106
107              return;
108          }
109
```

```
        /* fix size or addr if block overlaps hole */
        if(addr >= hole_addr && addr < hole_top) {
            ptr->addr = hole_top;
            ptr->count = (top - hole_top) / (PAGE_SIZE >> PFADDR_SHIFT);

            return;
        }

        if(top > hole_addr && top <= hole_top) {
            ptr->count = (hole_addr - addr) / (PAGE_SIZE >> PFADDR_SHIFT);
        }
    }
}
```

Here is the call graph for this function:



4.27.2.2 bootmem_t∗ bootmem_get_block (void)

Definition at line 244 of file bootmem.c.

References bootmem_root, bootmem_t::next, and NULL.

Referenced by dispatch_syscall().

```
244                                     {
245     bootmem_t *block;
246
247     block = bootmem_root;
248
249     if(block != NULL) {
250         bootmem_root = block->next;
251     }
252
253     return block;
254 }
```

4.27.2.3 void bootmem_init (bool *use_pae*)

TODO check for available regions overlap

TODO this won't work for available memory > 4GB

Definition at line 124 of file bootmem.c.

References bootmem_t::addr, e820_t::addr, ADDR_4GB, ADDR_TO_PFADDR, apply_mem_hole(), boot_heap, boot_-info, bootmem_t::count, boot_info_t::e820_entries, e820_is_available(), e820_is_valid(), boot_info_t::e820_map, get_-boot_info(), boot_info_t::image_start, KB, kernel_region_top, new_ram_map_entry(), bootmem_t::next, NULL, OFFSE-T_OF, PAGE_MASK, PAGE_SIZE, panic(), printk(), ram_map, and e820_t::size.

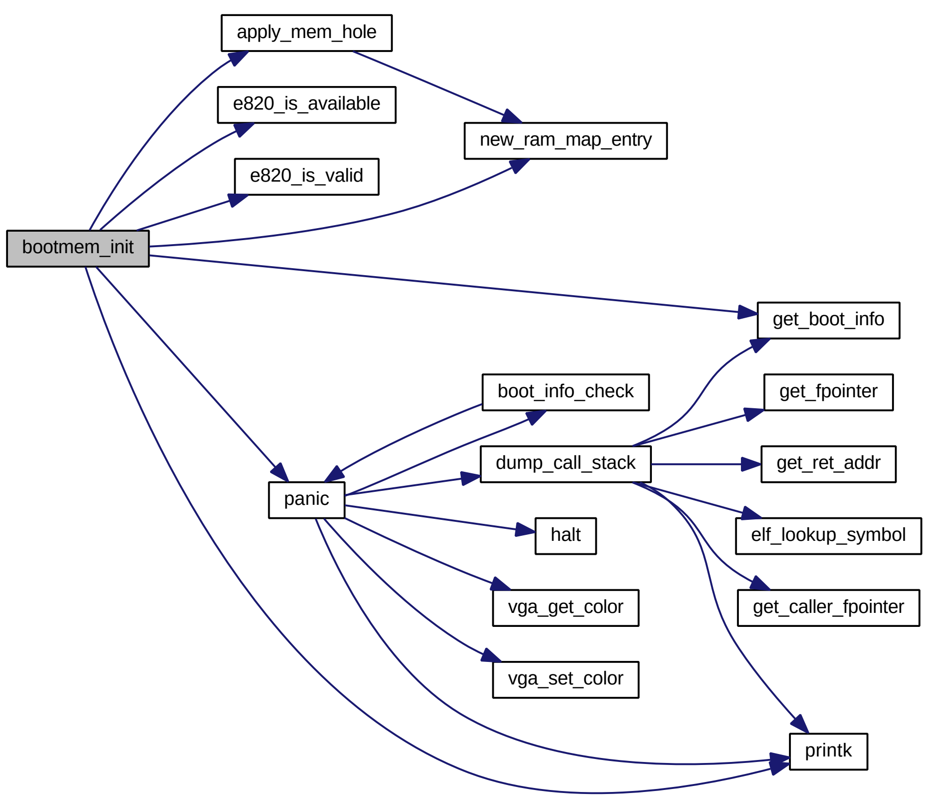
Referenced by vm_boot_init().

```
124                                 {
125     const addr_t initial_boot_heap = boot_heap;
126
127     bootmem_t *ptr;
128     bootmem_t *temp_root;
129     unsigned int idx;
130
131     const boot_info_t *boot_info = get_boot_info();
```

```

    /* copy the available ram entries from the e820 map and insert them
     * in a linked list */
    ram_map = NULL;

    for(idx = 0; idx < boot_info->e820_entries; ++idx) {
        const e820_t *e820_entry = &boot_info->e820_map[idx];

        if(! e820_is_valid(e820_entry)) {
            continue;
        }

        if( e820_is_available(e820_entry) ) {
            /* get memory entry start and end addresses */
            e820_addr_t start = e820_entry->addr;
            e820_addr_t end   = start + e820_entry->size;

            /* align on page boundaries */
            if( OFFSET_OF(start, PAGE_SIZE) != 0 ) {
                start = (start & (e820_addr_t)~PAGE_MASK) + PAGE_SIZE;
            }

            if( OFFSET_OF(end, PAGE_SIZE) != 0 ) {
                end = (end & (e820_addr_t)~PAGE_MASK);
            }

            /* If Physical Address Extension (PAE) is disabled, memory above the
             * 4GB mark is not usable. */
            if(! use_pae) {
                /* If this memory region is completely above the 4GB mark, exclude it. */
                if(start >= ADDR_4GB) {
                    continue;
                }

                /* If this memory region starts below the 4GB mark but extends
                 * beyond it, crop at 4GB. */
                if(end > ADDR_4GB) {
                    end = ADDR_4GB;
                }
            }

            /* add entry to linked list */
            if(end > start) {
                new_ram_map_entry(ADDR_TO_PFADDR(start), (uint32_t)(end - start) / PAGE_SIZE, &ram_map);
            }
        }
    }

    /* apply every unavailable entries from the e820 map as holes */
    for(idx = 0; idx < boot_info->e820_entries; ++idx) {
        const e820_t *e820_entry = &boot_info->e820_map[idx];

        if(! e820_is_valid(e820_entry)) {
            continue;
        }

        if( e820_is_available(e820_entry) ) {
            continue;
        }

        e820_addr_t start = e820_entry->addr;
        e820_addr_t end   = start + e820_entry->size;

        apply_mem_hole(start, end, &ram_map);
    }

    /* Apparently, the first 64k of memory are corrupted by some BIOSes.
         * It would be nice to try to detect this. In the meantime, let's
         * assume the problem is present. */
    apply_mem_hole(0, 0x10000, &ram_map);

    /* the kernel image, its heap and stack, and early-allocated pages */
    apply_mem_hole((uint32_t)boot_info->image_start, (uint32_t)kernel_region_top, &ram_map);

    /* Entry removal may have left garbage on the heap (bootmem_t
     * structures which were allocated on the heap but are no longer
     * linked). Let's clean up. */
    temp_root = NULL;

```

```
    for(ptr = ram_map; ptr != NULL; ptr = ptr->next) {
        new_ram_map_entry(ptr->addr, ptr->count, &temp_root);
    }

    ram_map   = NULL;
    boot_heap = initial_boot_heap;

    for(ptr = temp_root; ptr != NULL; ptr = ptr->next) {
        new_ram_map_entry(ptr->addr, ptr->count, &ram_map);
    }

    /* at this point, we should have at least one block of available RAM */
    if( ram_map == NULL ) {
        panic("no available memory.");
    }

    /* Let's count and display the total amount of available memory */
    uint32_t page_count = 0;
    for(ptr = ram_map; ptr != NULL; ptr = ptr->next) {
        page_count += ptr->count;
    }

    printk("%u kilobytes (%u pages) of memory available.\n",
        (uint32_t)(page_count * PAGE_SIZE / KB),
        (uint32_t)(page_count) );

    /* head pointer for bootmem_get_block() */
    bootmem_root = ram_map;
}
```

Here is the call graph for this function:



4.27.2.4 void new_ram_map_entry (pfaddr_t *addr*, uint32_t *count*, bootmem_t ** *head*)

Definition at line 55 of file bootmem.c.

References bootmem_t::addr, boot_heap, bootmem_t::count, and bootmem_t::next.

Referenced by apply_mem_hole(), and bootmem_init().

```
                                                                    {
    bootmem_t   *entry;

    entry     = (bootmem_t *)boot_heap;
    boot_heap = (bootmem_t *)boot_heap + 1;

    entry->next  = *head;
    entry->addr  = addr;
    entry->count = count;

    *head = entry;
}
```

4.27.3 Variable Documentation

4.27.3.1 void* boot_heap

current top of boot heap

Definition at line 52 of file bootmem.c.

Referenced by bootmem_init(), hal_init(), and new_ram_map_entry().

4.27.3.2 bootmem_t* bootmem_root

available memory map (allocator)

Definition at line 49 of file bootmem.c.

Referenced by bootmem_get_block(), and dispatch_syscall().

4.27.3.3 bootmem_t* ram_map

kernel memory map

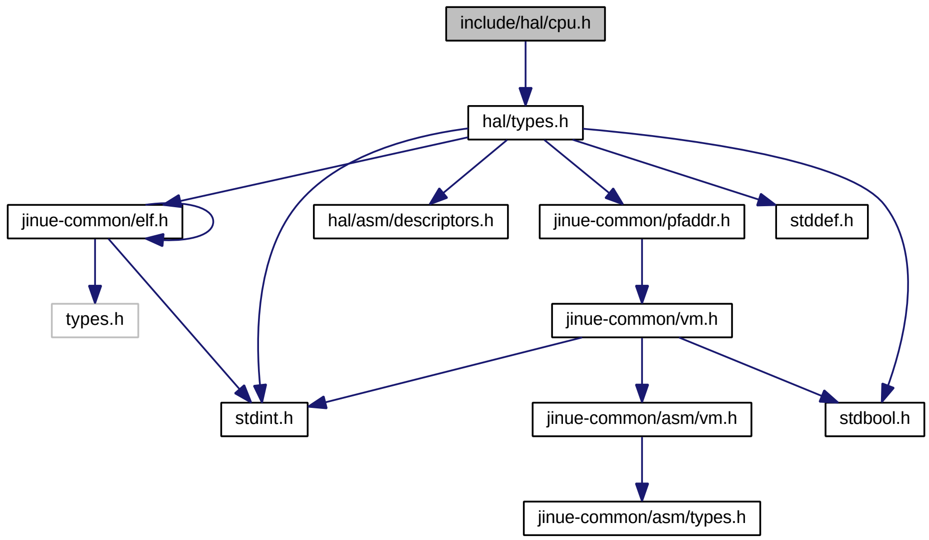
Definition at line 46 of file bootmem.c.

Referenced by bootmem_init().

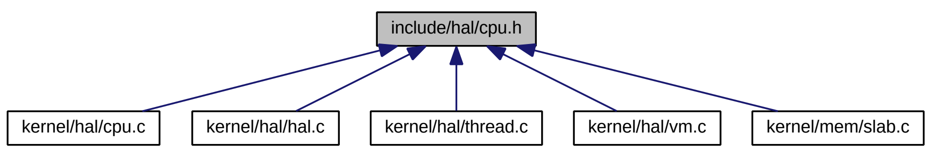
4.28 include/hal/cpu.h File Reference

```
#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, **addr_t** kernel_stack)
- void **cpu_detect_features** (void)

Variables

- **cpu_info_t cpu_info**

4.28.1 Macro Definition Documentation

4.28.1.1 #define CPU_EFLAGS_ID (1<<21)

Definition at line 63 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.2 #define CPU_FEATURE_CPUID (1<<0)

Definition at line 52 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.3 #define CPU_FEATURE_LOCAL_APIC (1<<3)

Definition at line 58 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.4 #define CPU_FEATURE_PAE (1<<4)

Definition at line 60 of file cpu.h.

Referenced by cpu_detect_features(), and vm_boot_init().

4.28.1.5 #define CPU_FEATURE_SYSCALL (1<<2)

Definition at line 56 of file cpu.h.

Referenced by cpu_detect_features(), and hal_init().

4.28.1.6 #define CPU_FEATURE_SYSENTER (1<<1)

Definition at line 54 of file cpu.h.

Referenced by cpu_detect_features(), hal_init(), and thread_context_switch().

4.28.1.7 #define CPU_VENDOR_AMD 1

Definition at line 84 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.8 #define CPU_VENDOR_AMD_DW0 0x68747541 /* Auth */

Definition at line 89 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.9 #define CPU_VENDOR_AMD_DW1 0x69746e65 /* enti */

Definition at line 90 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.10 #define CPU_VENDOR_AMD_DW2 0x444d4163 /* cAMD */

Definition at line 91 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.11 #define CPU_VENDOR_GENERIC 0

Definition at line 82 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.12 #define CPU_VENDOR_INTEL 2

Definition at line 86 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.13 #define CPU_VENDOR_INTEL_DW0 0x756e6547 /* Genu */

Definition at line 93 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.14 #define CPU_VENDOR_INTEL_DW1 0x49656e69 /* inel */

Definition at line 94 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.15 #define CPU_VENDOR_INTEL_DW2 0x6c65746e /* ntel */

Definition at line 95 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.16 #define CPUID_EXT_FEATURE_SYSCALL (1<<11)

Definition at line 79 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.17 #define CPUID_FEATURE_APIC (1<<9)

Definition at line 70 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.18 #define CPUID_FEATURE_CLFLUSH (1<<19)

Definition at line 74 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.19 #define CPUID_FEATURE_FPU (1<<0)

Definition at line 66 of file cpu.h.

4.28.1.20 #define CPUID_FEATURE_HTT (1<<28)

Definition at line 76 of file cpu.h.

4.28.1.21 #define CPUID_FEATURE_PAE (1<<6)

Definition at line 68 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.22 #define CPUID_FEATURE_SEP (1<<11)

Definition at line 72 of file cpu.h.

Referenced by cpu_detect_features().

4.28.1.23 #define MSR_EFER 0xC0000080

Definition at line 44 of file cpu.h.

Referenced by hal_init().

4.28.1.24 #define MSR_FLAG_STAR_SCE (1<<0)

Definition at line 49 of file cpu.h.

Referenced by hal_init().

4.28.1.25 #define MSR_IA32_SYSENTER_CS 0x174

Definition at line 38 of file cpu.h.

Referenced by hal_init().

4.28.1.26 #define MSR_IA32_SYSENTER_EIP 0x176

Definition at line 42 of file cpu.h.

Referenced by hal_init().

4.28.1.27 #define MSR_IA32_SYSENTER_ESP 0x175

Definition at line 40 of file cpu.h.

Referenced by hal_init(), and thread_context_switch().

4.28.1.28 #define MSR_STAR 0xC0000081

Definition at line 46 of file cpu.h.

Referenced by hal_init().

4.28.2 Function Documentation

4.28.2.1 void cpu_detect_features (void)

Definition at line 87 of file cpu.c.

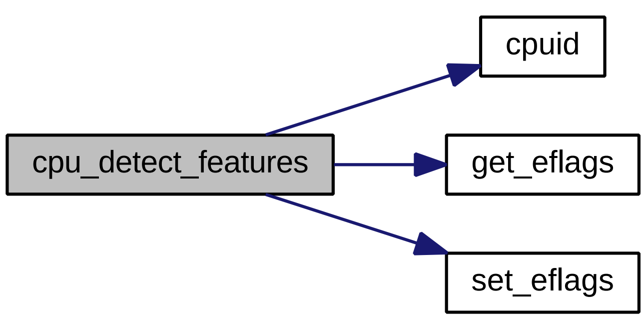
References CPU_EFLAGS_ID, CPU_FEATURE_CPUID, CPU_FEATURE_LOCAL_APIC, CPU_FEATURE_PAE, CP-U_FEATURE_SYSCALL, CPU_FEATURE_SYSENTER, CPU_VENDOR_AMD, CPU_VENDOR_AMD_DW0, CPU_V-ENDOR_AMD_DW1, CPU_VENDOR_AMD_DW2, CPU_VENDOR_GENERIC, CPU_VENDOR_INTEL, CPU_VEND-OR_INTEL_DW0, CPU_VENDOR_INTEL_DW1, CPU_VENDOR_INTEL_DW2, 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:



4.28.2.2 void cpu_init_data (cpu_data_t * *data,* addr_t *kernel_stack*)

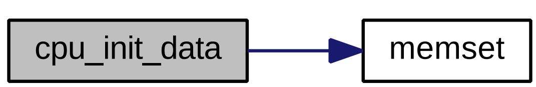
Definition at line 42 of file cpu.c.

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:



4.28.3 Variable Documentation

4.28.3.1 cpu_info_t cpu_info

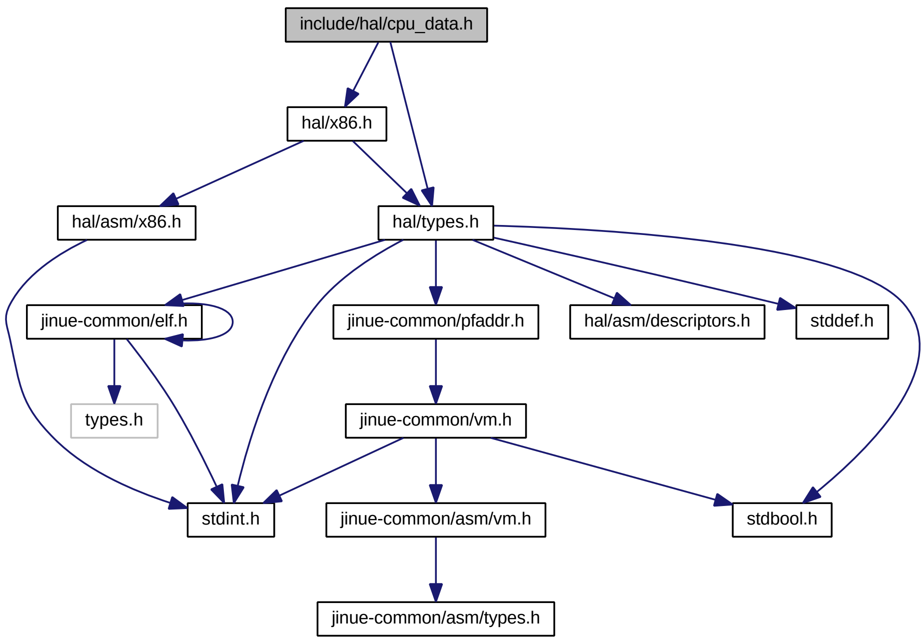
Definition at line 39 of file cpu.c.

Referenced by slab_cache_create().

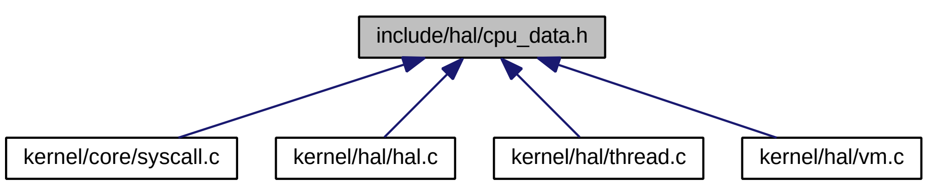
4.29 include/hal/cpu_data.h File Reference

```
#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

4.29.1 Macro Definition Documentation

4.29.1.1 #define CPU_DATA_ALIGNMENT 256

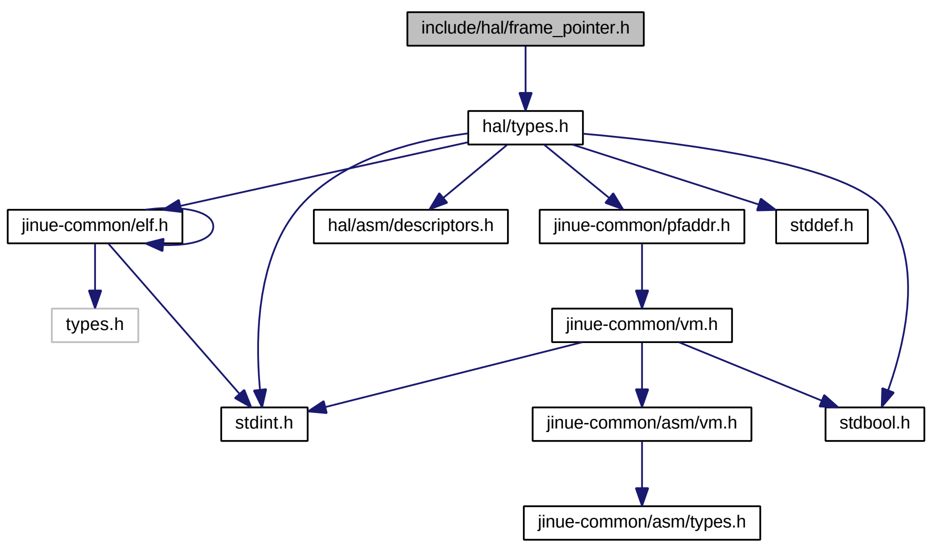
Definition at line 39 of file cpu_data.h.

Referenced by hal_init().

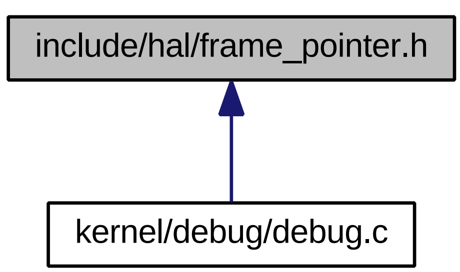
4.30 include/hal/frame_pointer.h File Reference

`#include <hal/types.h>`

Include dependency graph for frame_pointer.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)

4.30.1 Function Documentation

4.30.1.1 addr_t get_caller_fpointer (addr_t *fptr*)

Referenced by dump_call_stack().

4.30.1.2 addr_t get_fpointer (void)

Referenced by dump_call_stack().

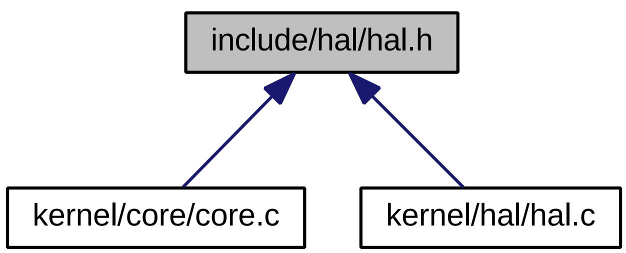
4.30.1.3 addr_t get_program_counter (void)

4.30.1.4 addr_t get_ret_addr (addr_t *fptr*)

Referenced by dump_call_stack().

4.31 include/hal/hal.h File Reference

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



Functions

- void **hal_init** (void)

4.31.1 Function Documentation

4.31.1.1 void hal_init (void)

ASSERTION: we assume the image starts on a page boundary

ASSERTION: we assume the kernel starts on a page boundary

Definition at line 64 of file hal.c.

References pseudo_descriptor_t::addr, ALIGN_END, assert, boot_info_t::boot_end, boot_heap, boot_info_t::boot_-heap, boot_info, boot_info_check(), CPU_DATA_ALIGNMENT, cpu_detect_features(), CPU_FEATURE_SYSCALL, C-PU_FEATURE_SYSENTER, cpu_init_data(), EARLY_PTR_TO_PFADDR, fast_amd_entry(), fast_intel_entry(), GATE-_DESCRIPTOR, cpu_data_t::gdt, GDT_KERNEL_CODE, GDT_KERNEL_DATA, GDT_LENGTH, GDT_PER_CPU_D-ATA, GDT_TSS, GDT_USER_CODE, get_boot_info(), global_pfcache, idt, IDT_VECTOR_COUNT, boot_info_t::image-_start, init_pfcache(), KERNEL_PAGE_STACK_INIT, kernel_region_top, boot_info_t::kernel_size, boot_info_t::kernel_-start, lgdt(), lidt(), pseudo_descriptor_t::limit, ltr(), MSR_EFER, MSR_FLAG_STAR_SCE, MSR_IA32_SYSENTER_CS,

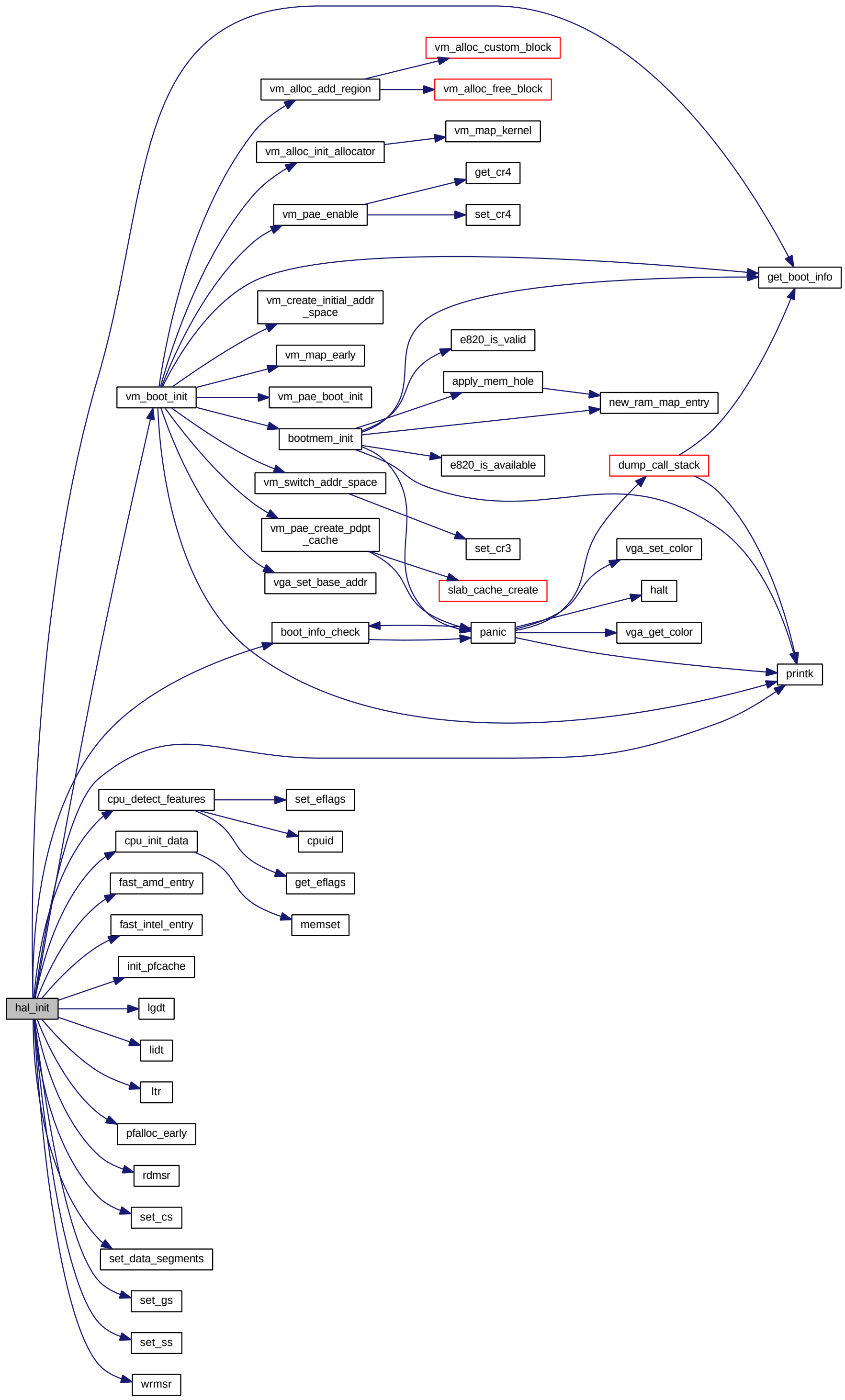
MSR_IA32_SYSENTER_EIP, MSR_IA32_SYSENTER_ESP, MSR_STAR, NULL, page_offset_of, PAGE_SIZE, pfalloc-_early(), pffree, printk(), rdmsr(), RPL_KERNEL, RPL_USER, SEG_FLAG_KERNEL, SEG_FLAG_NORMAL_GATE, S-EG_FLAG_USER, SEG_SELECTOR, SEG_TYPE_INTERRUPT_GATE, set_cs(), set_data_segments(), set_gs(), set-_ss(), SYSCALL_IRQ, syscall_method, SYSCALL_METHOD_FAST_AMD, SYSCALL_METHOD_FAST_INTEL, SYS-CALL_METHOD_INTR, use_pfalloc_early, vm_boot_init(), and wrmsr().

Referenced by kmain().

```
                {
    addr_t addr;
    addr_t               stack;
    cpu_data_t          *cpu_data;
    pseudo_descriptor_t *pseudo;
    unsigned int         idx;
    unsigned int         flags;
    uint64_t             msrval;
    pfaddr_t            *page_stack_buffer;
    addr_t               boot_heap_old;

    /* pfalloc() should not be called yet -- use pfalloc_early() instead */
    use_pfalloc_early = true;

    (void)boot_info_check(true);

    const boot_info_t *boot_info = get_boot_info();

    assert(page_offset_of(boot_info->image_start) == 0);

    assert(page_offset_of(boot_info->kernel_start) == 0);

    printk("Kernel size is %u bytes.\n", boot_info->kernel_size);

    /* This must be done before any boot heap allocation. */
    boot_heap = boot_info->boot_heap;

    /* This must be done before any call to pfalloc_early(). */
    kernel_region_top = boot_info->boot_end;

    /* get cpu info */
    cpu_detect_features();

    /* allocate new kernel stack */
     stack = pfalloc_early();
     stack += PAGE_SIZE;

     /* 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);

     boot_heap = ALIGN_END(boot_heap, CPU_DATA_ALIGNMENT);

     cpu_data  = boot_heap;
     boot_heap = cpu_data + 1;

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

     /* allocate pseudo-descriptor for GDT and IDT (temporary allocation) */
     boot_heap_old = boot_heap;

     boot_heap = ALIGN_END(boot_heap, sizeof(pseudo_descriptor_t));

     pseudo    = (pseudo_descriptor_t *)boot_heap;
     boot_heap = (pseudo_descriptor_t *)boot_heap + 1;

     /* load new GDT and TSS */
     pseudo->addr    = (addr_t)&cpu_data->gdt;
     pseudo->limit   = GDT_LENGTH * 8 - 1;

     lgdt(pseudo);

     set_cs( SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL) );
     set_ss( SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL) );
     set_data_segments( SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL) );
     set_gs( SEG_SELECTOR(GDT_PER_CPU_DATA, RPL_KERNEL) );

```

```
    ltr( SEG_SELECTOR(GDT_TSS, RPL_KERNEL) );

    /* initialize IDT */
    for(idx = 0; idx < IDT_VECTOR_COUNT; ++idx) {
        /* get address, which is already stored in the IDT entry */
        addr  = (addr_t)(uintptr_t)idt[idx];

        /* set interrupt gate flags */
        flags = SEG_TYPE_INTERRUPT_GATE | SEG_FLAG_NORMAL_GATE;

        if(idx == SYSCALL_IRQ) {
            flags |= SEG_FLAG_USER;
        }
        else {
            flags |= SEG_FLAG_KERNEL;
        }

        /* create interrupt gate descriptor */
        idt[idx] = GATE_DESCRIPTOR(
            SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL),
            addr,
            flags,
            NULL );
    }

    pseudo->addr  = (addr_t)idt;
    pseudo->limit = IDT_VECTOR_COUNT * sizeof(seg_descriptor_t) - 1;
    lidt(pseudo);

    /* de-allocate pseudo-descriptor */
    boot_heap = boot_heap_old;

    /* initialize the page frame allocator */
    page_stack_buffer = (pfaddr_t *)pfalloc_early();
    init_pfcache(&global_pfcache, page_stack_buffer);

    for(idx = 0; idx < KERNEL_PAGE_STACK_INIT; ++idx) {
        pffree( EARLY_PTR_TO_PFADDR( pfalloc_early() ) );
    }

    /* initialize virtual memory management, enable paging
     *
     * below this point, it is no longer safe to call pfalloc_early() */
    vm_boot_init();

    /* choose system call method */
    syscall_method = SYSCALL_METHOD_INTR;

    if(cpu_has_feature(CPU_FEATURE_SYSENTER)) {
        syscall_method = SYSCALL_METHOD_FAST_INTEL;

        wrmsr(MSR_IA32_SYSENTER_CS,  SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL));
        wrmsr(MSR_IA32_SYSENTER_EIP, (uint64_t)(uintptr_t)fast_intel_entry);

        /* kernel stack address is set when switching thread context */
        wrmsr(MSR_IA32_SYSENTER_ESP, (uint64_t)(uintptr_t)NULL);
    }

    if(cpu_has_feature(CPU_FEATURE_SYSCALL)) {
        syscall_method = SYSCALL_METHOD_FAST_AMD;

        msrval  = rdmsr(MSR_EFER);
        msrval |= MSR_FLAG_STAR_SCE;
        wrmsr(MSR_EFER, msrval);

        msrval  = (uint64_t)(uintptr_t)fast_amd_entry;
        msrval |= (uint64_t)SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL)   << 32;
        msrval |= (uint64_t)SEG_SELECTOR(GDT_USER_CODE,   RPL_USER)     << 48;

        wrmsr(MSR_STAR, msrval);
    }
}
```

Here is the call graph for this function:

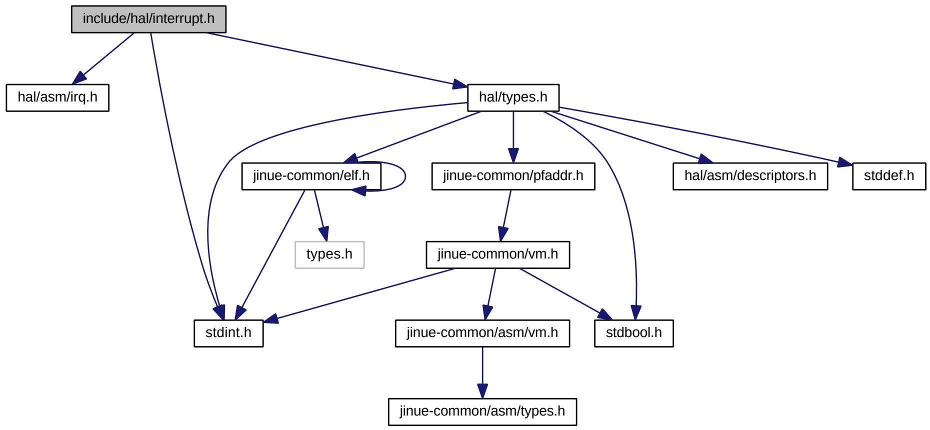


4.32 include/hal/interrupt.h File Reference

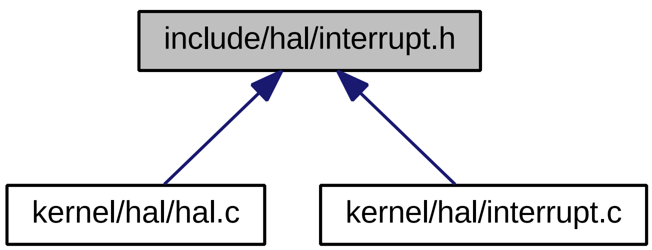
```
#include <hal/asm/irq.h>
```

```
#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** []

4.32.1 Function Documentation

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

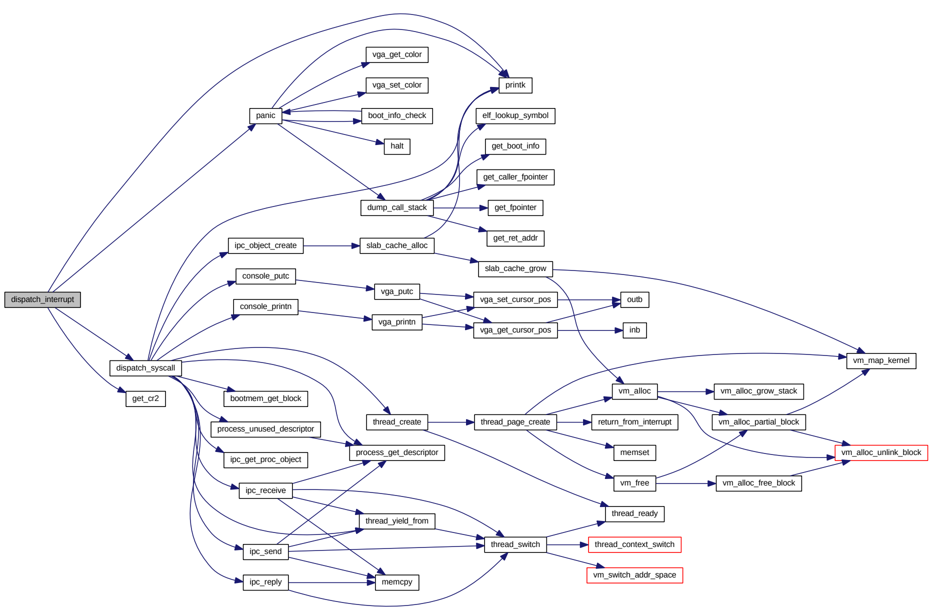
Definition at line 40 of file interrupt.c.

References dispatch_syscall(), trapframe_t::eip, trapframe_t::errcode, get_cr2(), IDT_FIRST_IRQ, trapframe_t::ivt, panic(), printk(), and SYSCALL_IRQ.

```
40                                                      {
41      unsigned int     ivt          = trapframe->ivt;
42      uintptr_t        eip          = trapframe->eip;
43      uint32_t         errcode      = trapframe->errcode;
```

```

    /* exceptions */
    if(ivt < IDT_FIRST_IRQ) {
        printk("EXCEPT: %u cr2=0x%x errcode=0x%x eip=0x%x\n", ivt, get_cr2(), errcode, eip);

        /* never returns */
        panic("caught exception");
    }

    /* slow system call method */
    if(ivt == SYSCALL_IRQ) {
        dispatch_syscall(trapframe);
    }
    else {
        printk("INTR: ivt %u (vector %u)\n", ivt - IDT_FIRST_IRQ, ivt);
    }
}
```

Here is the call graph for this function:



4.32.2 Variable Documentation

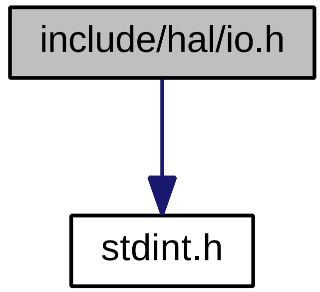
4.32.2.1 seg_descriptor_t idt[ ]

Referenced by hal_init().

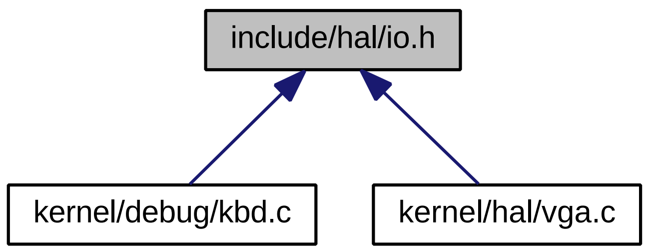
4.33 include/hal/io.h File Reference

```
#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)

4.33.1 Function Documentation

4.33.1.1 uint8_t inb (uint16_t *port*)

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

4.33.1.2 uint32_t inl (uint16_t *port*)

4.33.1.3 uint16_t inw (uint16_t *port*)

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

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

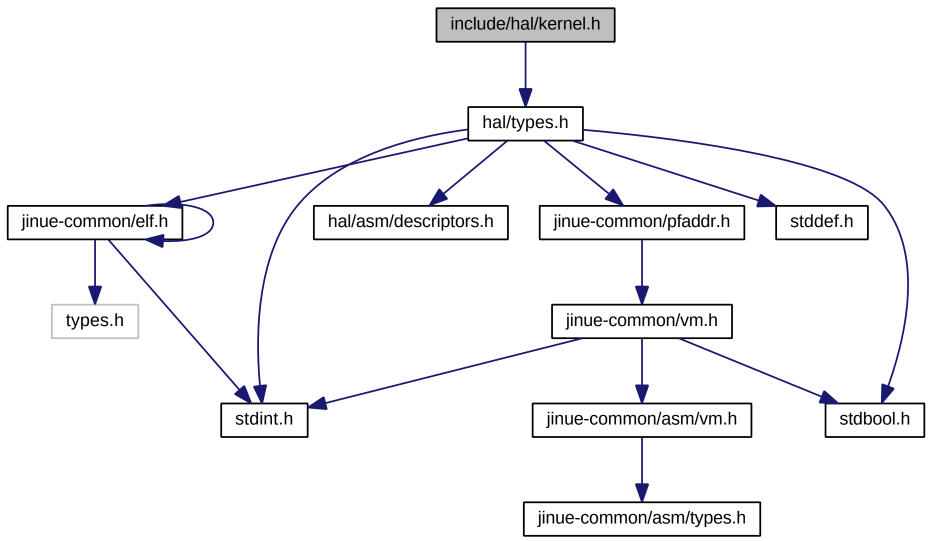
4.33.1.5 void outl (uint16_t *port*, uint32_t *value*)

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

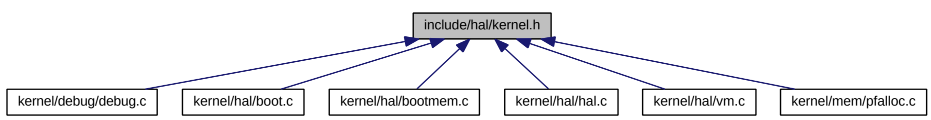
4.34 include/hal/kernel.h File Reference

`#include <hal/types.h>`

Include dependency graph for kernel.h:



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



Variables

- **addr_t kernel_region_top**

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

4.34.1 Variable Documentation

4.34.1.1 addr_t kernel_region_top

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

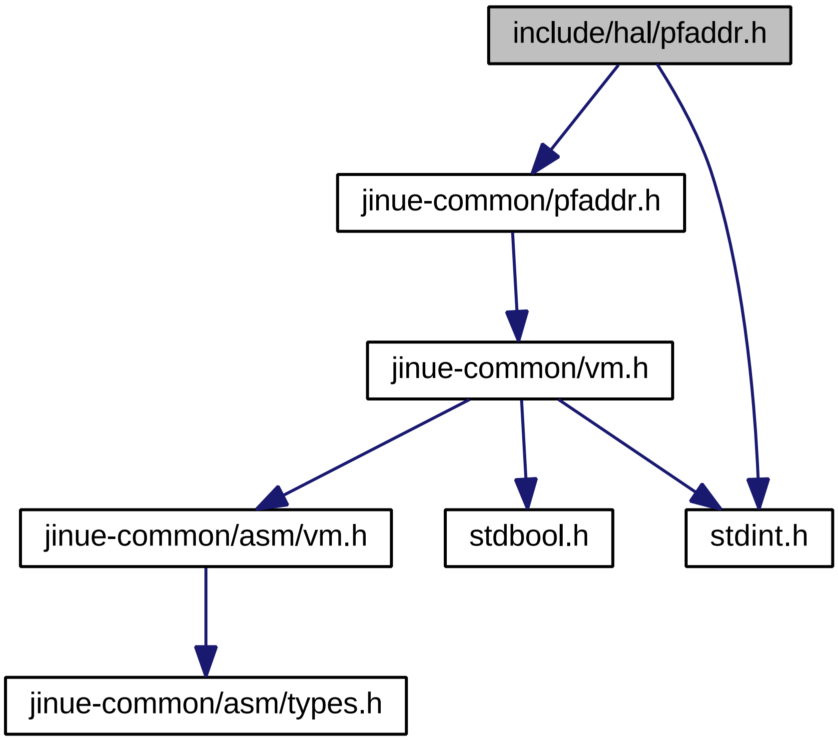
Definition at line 59 of file hal.c.

Referenced by bootmem_init(), hal_init(), pfalloc_early(), and vm_boot_init().

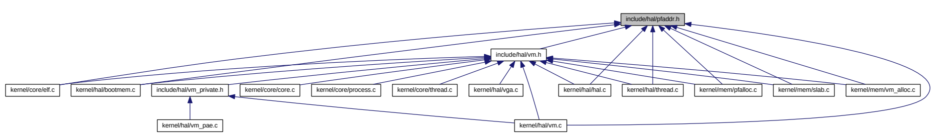
4.35 include/hal/pfaddr.h File Reference

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

Include dependency graph for pfaddr.h:



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



Macros

- #define **ADDR_TO_PFADDR**(x) ((**pfaddr_t**)((**uint64_t**)(x) >> **PFADDR_SHIFT**))

  convert an address in an integer to a page frame address
- #define **PFADDR_TO_ADDR**(x) ((**uint64_t**)(x) << **PFADDR_SHIFT**)

  convert a page frame address to an address in an integer
- #define **PFADDR_CHECK**(x) (((**uint32_t**)(x) << (32 - **PAGE_BITS** + **PFADDR_SHIFT**)) == 0)

  ensure page frame address is valid (LSBs zero)
- #define **PFADDR_CHECK_4GB**(x) (((**uint32_t**)(x) >> (32 - **PFADDR_SHIFT**)) == 0)

  check if the page frame address is below the 4GB (32-bit) limit

4.35.1 Macro Definition Documentation

4.35.1.1 #define ADDR_TO_PFADDR(x) ((pfaddr_t)((uint64_t)(x) >> PFADDR_SHIFT))

convert an address in an integer to a page frame address

Definition at line 39 of file pfaddr.h.

Referenced by bootmem_init(), and vm_boot_init().

4.35.1.2 #define PFADDR_CHECK(x) (((uint32_t)(x) << (32 - PAGE_BITS + PFADDR_SHIFT)) == 0)

ensure page frame address is valid (LSBs zero)

Definition at line 45 of file pfaddr.h.

4.35.1.3 #define PFADDR_CHECK_4GB(x) (((uint32_t)(x) >> (32 - PFADDR_SHIFT)) == 0)

check if the page frame address is below the 4GB (32-bit) limit

Definition at line 48 of file pfaddr.h.

4.35.1.4 #define PFADDR_TO_ADDR(x) ((uint64_t)(x) << PFADDR_SHIFT)

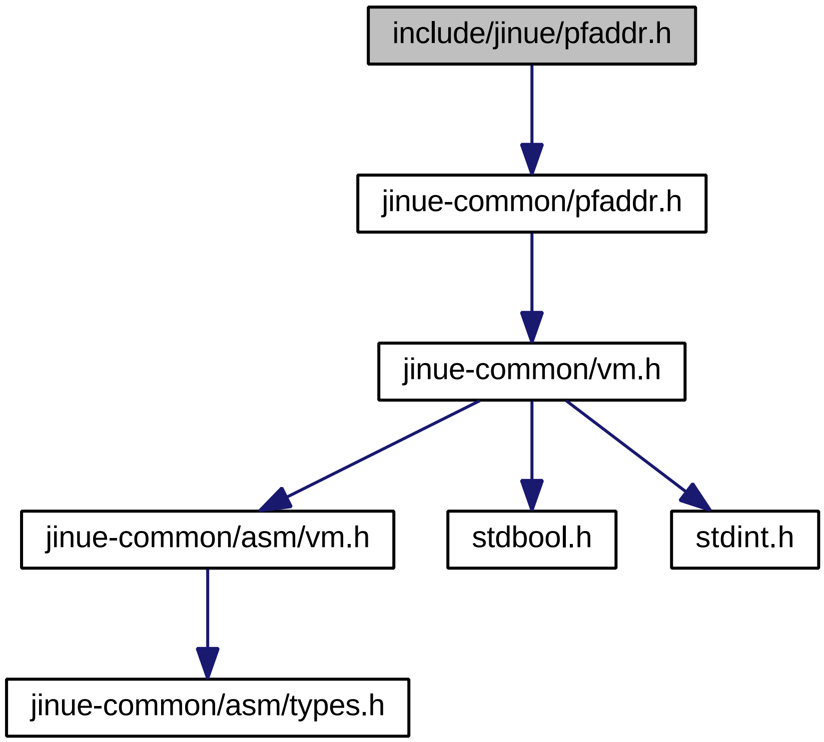
convert a page frame address to an address in an integer

Definition at line 42 of file pfaddr.h.

4.36 include/jinue/pfaddr.h File Reference

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

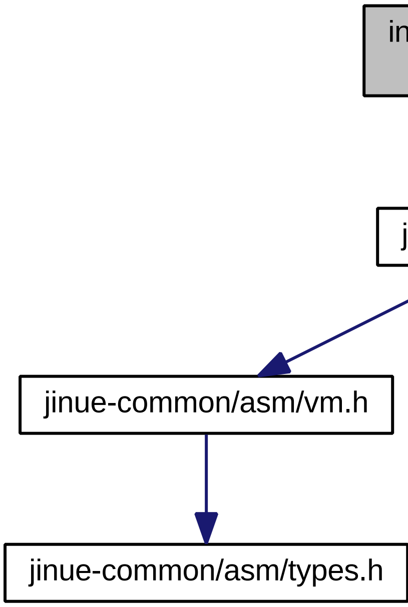
Include dependency graph for pfaddr.h:



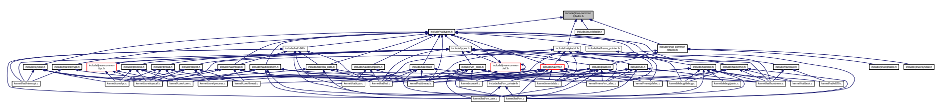
4.37 include/jinue-common/pfaddr.h File Reference

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

Include dependency graph for pfaddr.h:



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



Macros

- #define **PFADDR_SHIFT PAGE_BITS**

    number of bits by which the page frame address is shifted to the right
- #define **PFNULL** ((**pfaddr_t**)-1)

    an invalid page frame address used as null value

Typedefs

- typedef **uint32_t pfaddr_t**

    type for a page frame address (32-bit value)

4.37.1 Macro Definition Documentation

4.37.1.1 #define PFADDR_SHIFT PAGE_BITS

number of bits by which the page frame address is shifted to the right

Definition at line 41 of file pfaddr.h.

Referenced by apply_mem_hole().

4.37.1.2 #define PFNULL ((pfaddr_t)-1)

an invalid page frame address used as null value

Definition at line 44 of file pfaddr.h.

Referenced by init_pfcache(), and thread_page_create().

4.37.2 Typedef Documentation

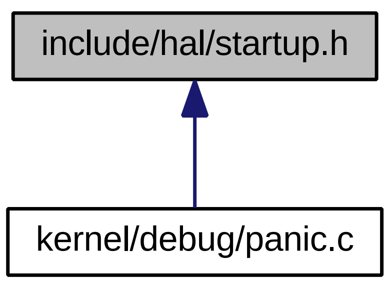
4.37.2.1 typedef uint32_t pfaddr_t

type for a page frame address (32-bit value)

Definition at line 38 of file pfaddr.h.

4.38 include/hal/startup.h File Reference

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



Functions

- void **halt** (void)

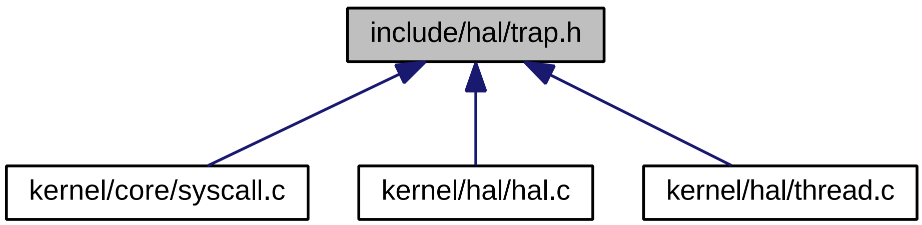
4.38.1 Function Documentation

4.38.1.1 void halt (void)

Referenced by panic().

4.39 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.

4.39.1 Function Documentation

4.39.1.1 void fast_amd_entry (void)

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

Referenced by hal_init().

4.39.1.2 void fast_intel_entry (void)

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

Referenced by hal_init().

4.39.1.3 void return_from_interrupt (void)

Referenced by thread_page_create().

4.39.2 Variable Documentation

4.39.2.1 int syscall_method

Specifies the entry point to use for system calls.

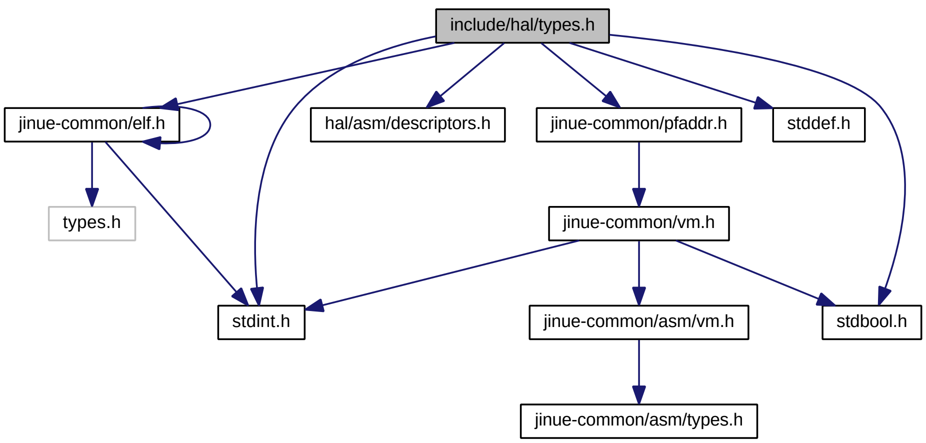
Definition at line 62 of file hal.c.

Referenced by dispatch_syscall(), and hal_init().

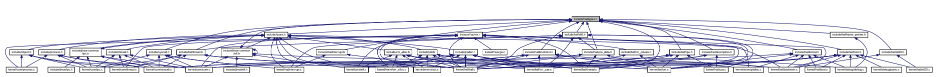
4.40 include/hal/types.h File Reference

```
#include <jinue-common/elf.h>
#include <jinue-common/pfaddr.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 **msg_arg0** eax

- #define **msg_arg1** ebx
- #define **msg_arg2** esi
- #define **msg_arg3** edi

Typedefs

- typedef unsigned char ∗ **addr_t**
- typedef struct **pte_t pte_t**

  type of a page table entry
- typedef struct **pdpt_t pdpt_t**
- typedef **uint32_t e820_type_t**
- typedef **uint64_t e820_size_t**
- typedef **uint64_t e820_addr_t**
- typedef **uint64_t seg_descriptor_t**
- typedef **uint32_t seg_selector_t**
- typedef struct **cpu_data_t cpu_data_t**

4.40.1 Macro Definition Documentation

4.40.1.1 #define msg_arg0 eax

Definition at line 190 of file types.h.

4.40.1.2 #define msg_arg1 ebx

Definition at line 192 of file types.h.

4.40.1.3 #define msg_arg2 esi

Definition at line 194 of file types.h.

4.40.1.4 #define msg_arg3 edi

Definition at line 196 of file types.h.

4.40.2 Typedef Documentation

4.40.2.1 typedef unsigned char∗ addr_t

Definition at line 43 of file types.h.

4.40.2.2 typedef struct cpu_data_t cpu_data_t

Definition at line 186 of file types.h.

4.40.2.3 typedef uint64_t e820_addr_t

Definition at line 84 of file types.h.

4.40.2.4 typedef uint64_t e820_size_t

Definition at line 82 of file types.h.

4.40.2.5 typedef uint32_t e820_type_t

Definition at line 80 of file types.h.

4.40.2.6 typedef struct pdpt_t pdpt_t

Definition at line 62 of file types.h.

4.40.2.7 typedef struct pte_t pte_t

type of a page table entry

Definition at line 58 of file types.h.

4.40.2.8 typedef uint64_t seg_descriptor_t

Definition at line 110 of file types.h.

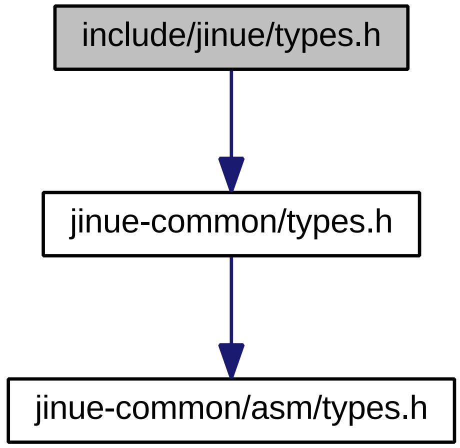
4.40.2.9 typedef uint32_t seg_selector_t

Definition at line 112 of file types.h.

4.41 include/jinue/types.h File Reference

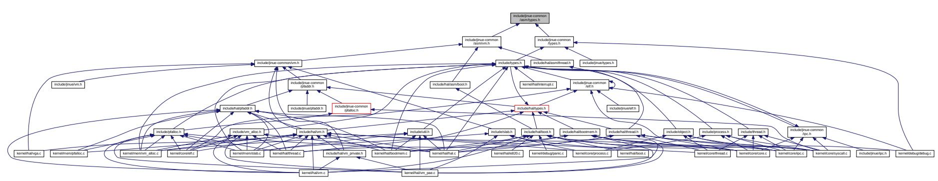
`#include <jinue-common/types.h>`

Include dependency graph for types.h:



4.42 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 ∗ 1024)

4.42.1 Macro Definition Documentation

4.42.1.1 #define KB (1024)

Definition at line 35 of file types.h.

Referenced by bootmem_init().

4.42.1.2 #define MB (1024 ∗ 1024)

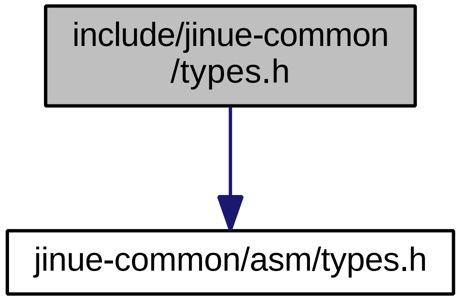
Definition at line 37 of file types.h.

Referenced by vm_boot_init().

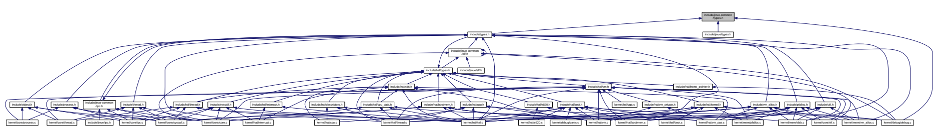
4.43 include/jinue-common/types.h File Reference

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

Include dependency graph for types.h:



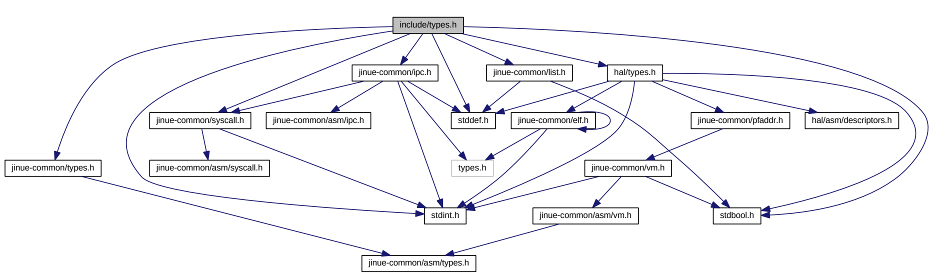
This graph shows which files directly or indirectly include this file:



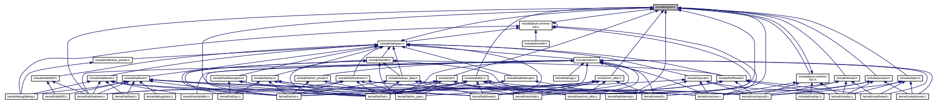
4.44 include/types.h File Reference

```
#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 **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**

4.44.1 Macro Definition Documentation

4.44.1.1 #define PROCESS_MAX_DESCRIPTORS 12

Definition at line 57 of file types.h.

Referenced by process_get_descriptor(), and process_unused_descriptor().

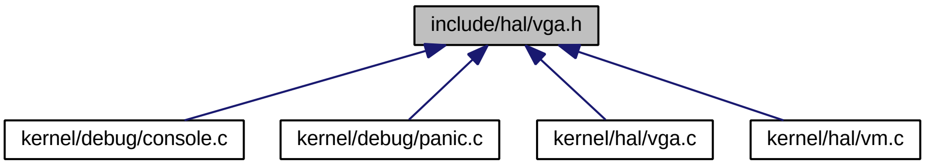
4.44.2 Typedef Documentation

4.44.2.1 typedef struct thread_t thread_t

Definition at line 85 of file types.h.

4.45 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_DEFAULT** **VGA_COLOR_BRIGHTGREEN**
- #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)
- void **vga_printn** (const char ∗message, unsigned int n)
- void **vga_putc** (char c)
- void **vga_scroll** (void)
- unsigned int **vga_get_color** (void)
- void **vga_set_color** (unsigned int color)
- **vga_pos_t** **vga_get_cursor_pos** (void)
- void **vga_set_cursor_pos** (**vga_pos_t** pos)

4.45.1 Macro Definition Documentation

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

Definition at line 69 of file vga.h.

4.45.1.2 #define VGA_COLOR_BLACK 0x00

Definition at line 45 of file vga.h.

4.45.1.3 #define VGA_COLOR_BLUE 0x01

Definition at line 46 of file vga.h.

4.45.1.4 #define VGA_COLOR_BRIGHTBLUE 0x09

Definition at line 54 of file vga.h.

4.45.1.5 #define VGA_COLOR_BRIGHTCYAN 0x0b

Definition at line 56 of file vga.h.

4.45.1.6 #define VGA_COLOR_BRIGHTGREEN 0x0a

Definition at line 55 of file vga.h.

4.45.1.7 #define VGA_COLOR_BRIGHTMAGENTA 0x0d

Definition at line 58 of file vga.h.

4.45.1.8 #define VGA_COLOR_BRIGHTRED 0x0c

Definition at line 57 of file vga.h.

4.45.1.9 #define VGA_COLOR_BRIGHTWHITE 0x0f

Definition at line 60 of file vga.h.

4.45.1.10 #define VGA_COLOR_BROWN 0x06

Definition at line 51 of file vga.h.

4.45.1.11 #define VGA_COLOR_CYAN 0x03

Definition at line 48 of file vga.h.

4.45.1.12 #define VGA_COLOR_DEFAULT VGA_COLOR_BRIGHTGREEN

Definition at line 61 of file vga.h.

Referenced by vga_init().

4.45.1.13 #define VGA_COLOR_ERASE VGA_COLOR_RED

Definition at line 62 of file vga.h.

Referenced by vga_clear(), and vga_scroll().

4.45.1.14 #define VGA_COLOR_GRAY 0x08

Definition at line 53 of file vga.h.

4.45.1.15 #define VGA_COLOR_GREEN 0x02

Definition at line 47 of file vga.h.

4.45.1.16 #define VGA_COLOR_MAGENTA 0x05

Definition at line 50 of file vga.h.

4.45.1.17 #define VGA_COLOR_RED 0x04

Definition at line 49 of file vga.h.

Referenced by panic().

4.45.1.18 #define VGA_COLOR_WHITE 0x07

Definition at line 52 of file vga.h.

4.45.1.19 #define VGA_COLOR_YELLOW 0x0e

Definition at line 59 of file vga.h.

4.45.1.20 #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().

4.45.1.21 #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().

4.45.1.22 #define VGA_FB_FLAG_ACTIVE 1

Definition at line 43 of file vga.h.

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

Definition at line 68 of file vga.h.

4.45.1.24 #define VGA_LINES 25

Definition at line 64 of file vga.h.

Referenced by vga_clear(), and vga_scroll().

4.45.1.25 #define VGA_MISC_OUT_RD 0x3cc

Definition at line 39 of file vga.h.

Referenced by vga_init().

4.45.1.26 #define VGA_MISC_OUT_WR 0x3c2

Definition at line 38 of file vga.h.

Referenced by vga_init().

4.45.1.27 #define VGA_TAB_WIDTH 8

Definition at line 66 of file vga.h.

4.45.1.28 #define VGA_TEXT_VID_BASE 0xb8000

Definition at line 35 of file vga.h.

Referenced by vm_boot_init().

4.45.1.29 #define VGA_TEXT_VID_SIZE (VGA_TEXT_VID_TOP - VGA_TEXT_VID_BASE)

Definition at line 37 of file vga.h.

4.45.1.30 #define VGA_TEXT_VID_TOP 0xc0000

Definition at line 36 of file vga.h.

Referenced by vm_boot_init().

4.45.1.31 #define VGA_WIDTH 80

Definition at line 65 of file vga.h.

Referenced by vga_clear(), and vga_scroll().

4.45.2 Typedef Documentation

4.45.2.1 typedef unsigned int vga_pos_t

Definition at line 72 of file vga.h.

4.45.3 Function Documentation

4.45.3.1 void vga_clear (void)

Definition at line 71 of file vga.c.

References VGA_COLOR_ERASE, VGA_LINES, and VGA_WIDTH.

Referenced by vga_init().

```
                    {
    unsigned int idx = 0;

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

4.45.3.2 unsigned int vga_get_color (void)

Definition at line 95 of file vga.c.

Referenced by panic().

```
                                {
    return vga_text_color;
}
```

4.45.3.3 vga_pos_t vga_get_cursor_pos (void)

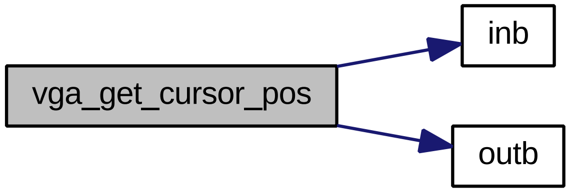
Definition at line 103 of file vga.c.

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:



4.45.3.4 void vga_init (void)

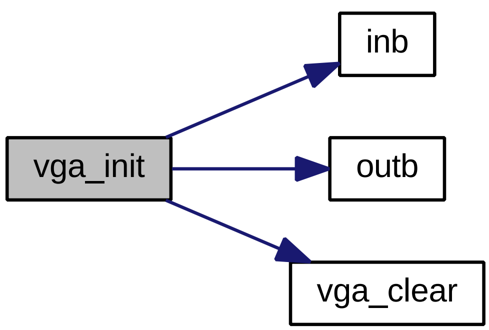
Definition at line 46 of file vga.c.

References inb(), outb(), vga_clear(), VGA_COLOR_DEFAULT, VGA_CRTC_ADDR, VGA_CRTC_DATA, VGA_MISC-_OUT_RD, and VGA_MISC_OUT_WR.

Referenced by console_init().

```
46                      {
47     unsigned char data;
48 
49     /* set text color to default */
50     vga_text_color = VGA_COLOR_DEFAULT;
51 
52     /* Set address select bit in a known state: CRTC regs at 0x3dx */
53     data = inb(VGA_MISC_OUT_RD);
54     data |= 1;
55     outb(VGA_MISC_OUT_WR, data);
56 
57     /* Move cursor to line 0 col 0 */
58     outb(VGA_CRTC_ADDR, 0x0e);
59     outb(VGA_CRTC_DATA, 0x0);
60     outb(VGA_CRTC_ADDR, 0x0f);
61     outb(VGA_CRTC_DATA, 0x0);
62 
63     /* Clear the screen */
64     vga_clear();
65 }
```

Here is the call graph for this function:



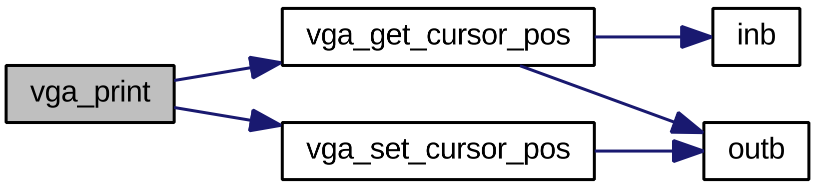
4.45.3.5 void vga_print (const char ∗ *message*)

Definition at line 125 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

```
                                    {
    unsigned short int pos = vga_get_cursor_pos();
    char c;

    while( (c = *(message++)) ) {
        pos = vga_raw_putc(c, pos);
    }

    vga_set_cursor_pos(pos);
}
```

Here is the call graph for this function:



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

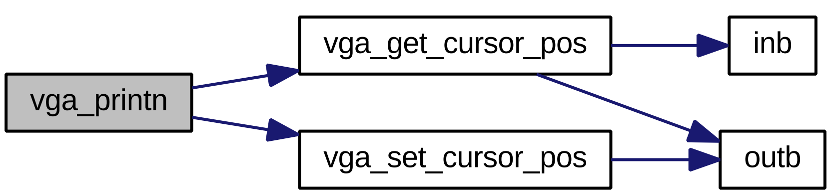
Definition at line 136 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by console_printn().

```
136                                                     {
137     vga_pos_t pos = vga_get_cursor_pos();
138     char c;
139 
140     while(n) {
141         c = *(message++);
142         pos = vga_raw_putc(c, pos);
143         --n;
144     }
145 
146     vga_set_cursor_pos(pos);
147 }
```

Here is the call graph for this function:



4.45.3.7 void vga_putc (char *c*)

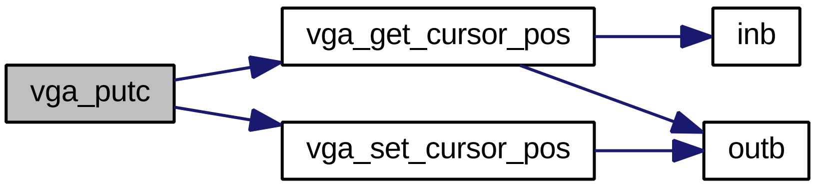
Definition at line 149 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by console_putc().

```
149                    {
150     vga_pos_t pos = vga_get_cursor_pos();
151 
152     pos = vga_raw_putc(c, pos);
153 
154     vga_set_cursor_pos(pos);
155 }
```

Here is the call graph for this function:



4.45.3.8 void vga_scroll (void)

Definition at line 80 of file vga.c.

References VGA_COLOR_ERASE, VGA_LINES, and VGA_WIDTH.

```
80                     {
81     unsigned char *di = video_base_addr;
82     unsigned char *si = video_base_addr + 2 * VGA_WIDTH;
83     unsigned int idx;
84 
85     for(idx = 0; idx < 2 * VGA_WIDTH * (VGA_LINES - 1); ++idx) {
86         *(di++) = *(si++);
87     }
88 
89     for(idx = 0; idx < VGA_WIDTH; ++idx) {
90         *(di++) = 0x20;
91         *(di++) = VGA_COLOR_ERASE;
92     }
93 }
```

4.45.3.9 void vga_set_base_addr (void * *base_addr*)

Definition at line 67 of file vga.c.

References vm_block_t::base_addr.

Referenced by vm_boot_init().

```
67                                          {
68     video_base_addr = base_addr;
69 }
```

4.45.3.10 void vga_set_color (unsigned int *color*)

Definition at line 99 of file vga.c.

Referenced by panic().

```
    {
    vga_text_color = color;
}
```

4.45.3.11 void vga_set_cursor_pos (vga_pos_t *pos*)

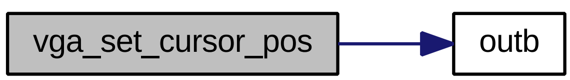
Definition at line 114 of file vga.c.

References outb(), VGA_CRTC_ADDR, and VGA_CRTC_DATA.

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

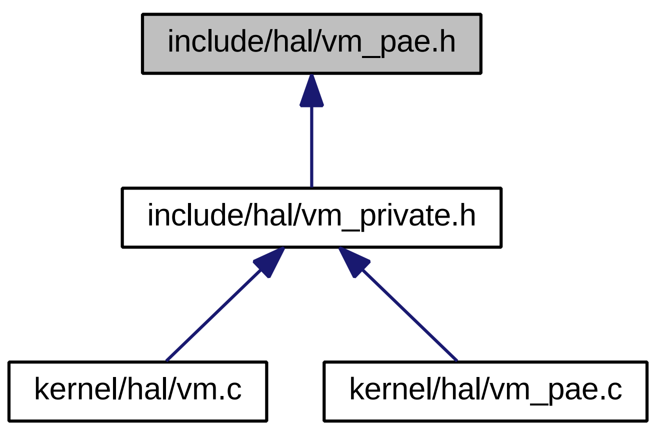
```
114                                   {
115     unsigned char h = pos >> 8;
116     unsigned char l = pos;
117
118     outb(VGA_CRTC_ADDR, 0x0e);
119     outb(VGA_CRTC_DATA, h);
120     outb(VGA_CRTC_ADDR, 0x0f);
121     outb(VGA_CRTC_DATA, l);
122 }
```

Here is the call graph for this function:



4.46 include/hal/vm_pae.h File Reference

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



Functions

- void **vm_pae_enable** (void)

    *This header file contains declarations for the PAE functions defined in **hal/vm_pae.c** (p. 342).*
- void **vm_pae_boot_init** (void)
- void **vm_pae_create_pdpt_cache** (void)

4.46.1 Function Documentation

4.46.1.1 void vm_pae_boot_init (void)

Definition at line 358 of file vm_pae.c.

References clear_pte, copy_pte, create_addr_space, create_initial_addr_space, destroy_addr_space, get_pte_flags, get_pte_pfaddr, get_pte_with_offset, lookup_page_directory, page_directory_offset_of, PAGE_TABLE_ENTRIES, page_table_entries, page_table_offset_of, set_pte, and set_pte_flags.

Referenced by vm_boot_init().

```
358                         {
359     page_table_entries          = (size_t)PAGE_TABLE_ENTRIES;
360     create_addr_space           = vm_pae_create_addr_space;
361     create_initial_addr_space   = vm_pae_create_initial_addr_space;
362     destroy_addr_space          = vm_pae_destroy_addr_space;
363     page_table_offset_of        = vm_pae_page_table_offset_of;
364     page_directory_offset_of    = vm_pae_page_directory_offset_of;
365     lookup_page_directory       = vm_pae_lookup_page_directory;
366     get_pte_with_offset         = vm_pae_get_pte_with_offset;
367     set_pte                     = vm_pae_set_pte;
368     set_pte_flags               = vm_pae_set_pte_flags;
369     get_pte_flags               = vm_pae_get_pte_flags;
370     get_pte_pfaddr              = vm_pae_get_pte_pfaddr;
371     clear_pte                   = vm_pae_clear_pte;
372     copy_pte                    = vm_pae_copy_pte;
373 }
```

4.46.1.2 void vm_pae_create_pdpt_cache (void)

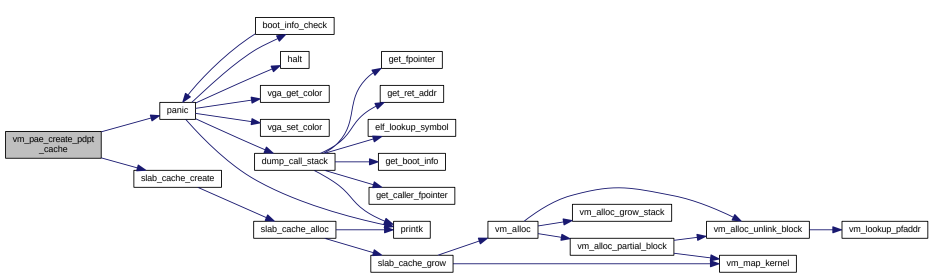
Definition at line 159 of file vm_pae.c.

References NULL, panic(), slab_cache_create(), and SLAB_DEFAULTS.

Referenced by vm_boot_init().

```
159                                 {
160     pdpt_cache = slab_cache_create(
161             "vm_pae_pdpt_cache",
162             sizeof(pdpt_t),
163             sizeof(pdpt_t),
164             NULL,
165             NULL,
166             SLAB_DEFAULTS);
167 
168     if(pdpt_cache == NULL) {
169         panic("Cannot create Page Directory Pointer Table (PDPT) slab cache.");
170     }
171 }
```

Here is the call graph for this function:



4.46.1.3 void vm_pae_enable (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. 330) and **hal/vm_pae.c** (p. 342). There should be no reason to include it anywhere else.

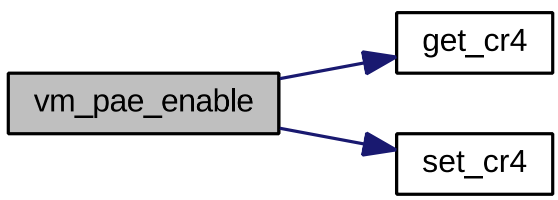
Definition at line 154 of file vm_pae.c.

References get_cr4(), set_cr4(), and X86_CR4_PAE.

Referenced by vm_boot_init().

```
154                         {
155     uint32_t temp = get_cr4();
156     set_cr4(temp | X86_CR4_PAE);
157 }
```

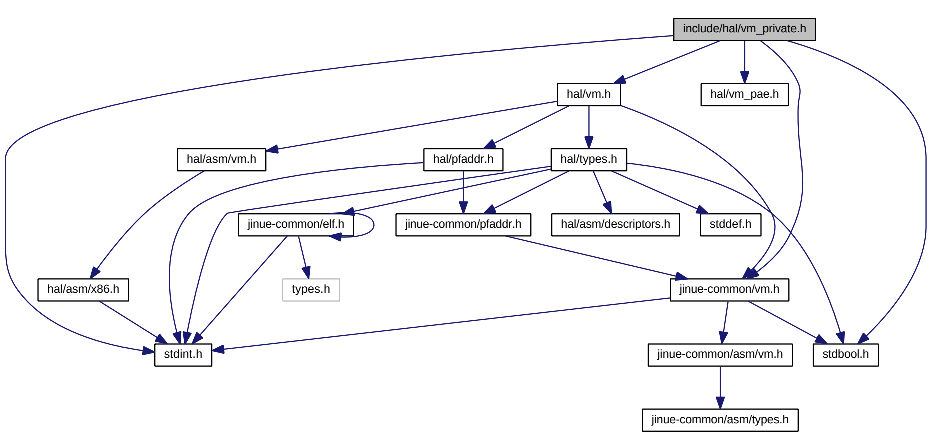
Here is the call graph for this function:



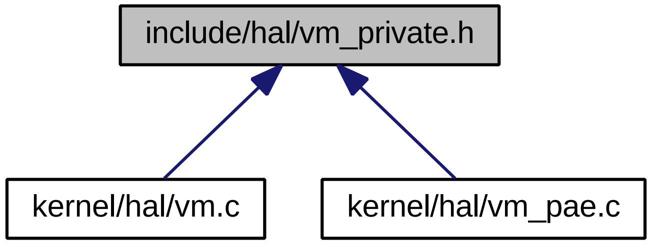
4.47 include/hal/vm_private.h File Reference

```
#include <jinue-common/vm.h>
#include <hal/vm.h>
#include <hal/vm_pae.h>
#include <stdbool.h>
#include <stdint.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. 330) and **hal/vm_pae.c** (p. 342).*
- #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

- **pfaddr_t vm_clone_page_directory** (**pfaddr_t** template_pfaddr, unsigned int start_index)
- **pte_t** ∗ **vm_allocate_page_directory** (unsigned int start_index, **bool** first_pd)
- void **vm_destroy_page_directory** (**pfaddr_t** pdpfaddr, unsigned int from_index, unsigned int to_index)

Variables

- **pte_t** ∗ **global_page_tables**
- **addr_space_t initial_addr_space**
- **size_t page_table_entries**
- **addr_space_t** ∗(∗ **create_addr_space**)(**addr_space_t** ∗)
- **addr_space_t** ∗(∗ **create_initial_addr_space**)(void)
- void(∗ **destroy_addr_space**)(**addr_space_t** ∗)
- unsigned int(∗ **page_table_offset_of**)(**addr_t**)

  page table entry offset of virtual (linear) address
- unsigned int(∗ **page_directory_offset_of**)(**addr_t**)
- **pte_t** ∗(∗ **lookup_page_directory**)(**addr_space_t** ∗, void ∗, **bool**)
- **pte_t** ∗(∗ **get_pte_with_offset**)(**pte_t** ∗, unsigned int)
- void(∗ **set_pte**)(**pte_t** ∗, **pfaddr_t**, int)
- void(∗ **set_pte_flags**)(**pte_t** ∗, int)
- int(∗ **get_pte_flags**)(**pte_t** ∗)
- **pfaddr_t**(∗ **get_pte_pfaddr**)(**pte_t** ∗)
- void(∗ **clear_pte**)(**pte_t** ∗)
- void(∗ **copy_pte**)(**pte_t** ∗, **pte_t** ∗)

4.47.1 Macro Definition Documentation

4.47.1.1 #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 54 of file vm_private.h.

4.47.1.2 #define PAGE_TABLE_ENTRIES (PAGE_SIZE / sizeof(pte_t))

This header file contains private definitions shared by **hal/vm.c** (p. 330) and **hal/vm_pae.c** (p. 342).

There should be no reason to include it anywhere else. number of entries in page table or page directory

Definition at line 45 of file vm_private.h.

Referenced by vm_pae_boot_init().

4.47.1.3 #define PAGE_TABLE_MASK (PAGE_TABLE_ENTRIES - 1)

bit mask for page table or page directory offset

Definition at line 48 of file vm_private.h.

4.47.1.4 #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 51 of file vm_private.h.

4.47.2 Function Documentation

4.47.2.1 pte_t* vm_allocate_page_directory (unsigned int *start_index*, bool *first_pd*)

Definition at line 524 of file vm.c.

References clear_pte, EARLY_PTR_TO_PFADDR, get_pte_with_offset, page_table_entries, pfalloc_early(), set_pte, and VM_FLAG_READ_WRITE.

Referenced by vm_x86_create_initial_addr_space().

```
524                                                                         {
525     unsigned int idx, idy;
526     pte_t *page_directory;
527     pte_t *page_table;
528
529     /* Allocate page directory. */
530     page_directory = (pte_t *)pfalloc_early();
531
532     /* clear user space page directory entries */
533     for(idx = 0; idx < start_index; ++idx) {
534         clear_pte( get_pte_with_offset(page_directory, idx) );
535     }
536
537     /* allocate page tables for kernel data/code region (above KLIMIT) */
538     for(idx = start_index; idx < page_table_entries; ++idx) {
539         /* allocate the page table
540          *
541          * Note that the use of pfalloc_early() here guarantees that the
542          * page table are allocated contiguously, and that they keep the
```

```
         * same address once paging is enabled. */
        page_table = (pte_t *)pfalloc_early();

        if(first_pd && 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_PFADDR(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) );
        }
    }

    return page_directory;
}
```

Here is the call graph for this function:

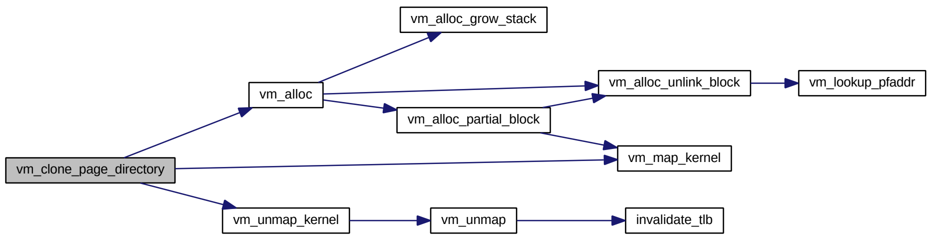
4.47.2.2 pfaddr_t vm_clone_page_directory (pfaddr_t *template_pfaddr,* unsigned int *start_index*)

Definition at line 472 of file vm.c.

References clear_pte, copy_pte, get_pte_with_offset, page_table_entries, pfalloc, vm_alloc(), VM_FLAG_READ_WRITE, vm_map_kernel(), and vm_unmap_kernel().

```
472                                                                              {
473     unsigned int idx;
474     pfaddr_t pfaddr;
475     pte_t *page_directory;
476     pte_t *template;
477 
478     /* allocate and map new page directory */
479     page_directory = (pte_t *)vm_alloc(global_page_allocator);
480     pfaddr = pfalloc();
481     vm_map_kernel((addr_t)page_directory, pfaddr, VM_FLAG_READ_WRITE);
482 
483     /* map page directory template */
484     template = (pte_t *)vm_alloc(global_page_allocator);
485     vm_map_kernel((addr_t)template, template_pfaddr, VM_FLAG_READ_WRITE);
486 
487     /* clear all entries below index start_index */
488     for(idx = 0; idx < start_index; ++idx) {
489         clear_pte( get_pte_with_offset(page_directory, idx) );
490     }
491 
492     /* copy entries from template for indexes start_index and above */
493     for(idx = start_index; idx < page_table_entries; ++idx) {
494         copy_pte(
495             get_pte_with_offset(page_directory, idx),
496             get_pte_with_offset(template, idx)
497         );
498     }
499 
500     vm_unmap_kernel((addr_t)page_directory);
501     vm_unmap_kernel((addr_t)template);
502 
503     return pfaddr;
504 }
```

Here is the call graph for this function:



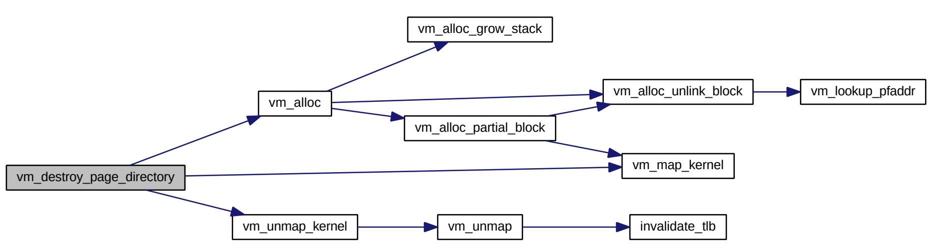
4.47.2.3 void vm_destroy_page_directory (pfaddr_t *pdpfaddr,* unsigned int *from_index,* unsigned int *to_index*)

Definition at line 581 of file vm.c.

References get_pte_flags, get_pte_pfaddr, get_pte_with_offset, pffree, vm_alloc(), VM_FLAG_READ_WRITE, vm_-map_kernel(), and vm_unmap_kernel().

```
581                                                                                          {
582     unsigned int idx;
583 
584     pte_t *page_directory = (pte_t *)vm_alloc(global_page_allocator);
585     vm_map_kernel((addr_t)page_directory, pdpfaddr, VM_FLAG_READ_WRITE);
586 
587     /* be careful not to free the kernel page tables */
588     for(idx = from_index; idx < to_index; ++idx) {
589         pte_t *pte = get_pte_with_offset(page_directory, idx);
590 
591         if(get_pte_flags(pte) & VM_FLAG_PRESENT) {
592             pffree( get_pte_pfaddr(pte) );
593         }
594     }
595 
596     vm_unmap_kernel((addr_t)page_directory);
597     pffree(pdpfaddr);
598 }
```

Here is the call graph for this function:



4.47.3 Variable Documentation

4.47.3.1 void(∗ clear_pte)(pte_t ∗)

Definition at line 703 of file vm.c.

Referenced by vm_allocate_page_directory(), vm_clone_page_directory(), vm_pae_boot_init(), and vm_unmap().

4.47.3.2 void(∗ copy_pte)(pte_t ∗, pte_t ∗)

Definition at line 705 of file vm.c.

Referenced by vm_clone_page_directory(), and vm_pae_boot_init().

4.47.3.3 addr_space_t∗(∗ create_addr_space)(addr_space_t ∗)

Definition at line 680 of file vm.c.

Referenced by vm_create_addr_space(), and vm_pae_boot_init().

4.47.3.4 addr_space_t∗(∗ create_initial_addr_space)(void)

Definition at line 682 of file vm.c.

Referenced by vm_create_initial_addr_space(), and vm_pae_boot_init().

4.47.3.5 void(∗ destroy_addr_space)(addr_space_t ∗)

Definition at line 684 of file vm.c.

Referenced by vm_destroy_addr_space(), and vm_pae_boot_init().

4.47.3.6 int(∗ get_pte_flags)(pte_t ∗)

Definition at line 699 of file vm.c.

Referenced by vm_change_flags(), vm_destroy_page_directory(), vm_lookup_pfaddr(), and vm_pae_boot_init().

4.47.3.7 pfaddr_t(∗ get_pte_pfaddr)(pte_t ∗)

Definition at line 701 of file vm.c.

Referenced by vm_destroy_page_directory(), vm_lookup_pfaddr(), and vm_pae_boot_init().

4.47.3.8 pte_t∗(∗ get_pte_with_offset)(pte_t ∗, unsigned int)

Definition at line 693 of file vm.c.

Referenced by vm_allocate_page_directory(), vm_clone_page_directory(), vm_destroy_page_directory(), vm_map_-early(), and vm_pae_boot_init().

4.47.3.9 pte_t∗ global_page_tables

Definition at line 51 of file vm.c.

4.47.3.10 addr_space_t initial_addr_space

Definition at line 53 of file vm.c.

Referenced by vm_x86_create_initial_addr_space().

4.47.3.11 pte_t∗(∗ lookup_page_directory)(addr_space_t ∗, void ∗, bool)

Definition at line 691 of file vm.c.

Referenced by vm_pae_boot_init().

4.47.3.12 unsigned int(∗ page_directory_offset_of)(addr_t)

Definition at line 689 of file vm.c.

Referenced by vm_pae_boot_init(), and vm_x86_create_initial_addr_space().

4.47.3.13 size_t page_table_entries

Definition at line 678 of file vm.c.

Referenced by vm_allocate_page_directory(), vm_clone_page_directory(), and vm_pae_boot_init().

4.47.3.14 unsigned int(∗ page_table_offset_of)(addr_t)

page table entry offset of virtual (linear) address

Definition at line 687 of file vm.c.

Referenced by vm_pae_boot_init().

4.47.3.15 void(∗ set_pte)(pte_t ∗, pfaddr_t, int)

Definition at line 695 of file vm.c.

Referenced by vm_allocate_page_directory(), vm_map_early(), and vm_pae_boot_init().

4.47.3.16 void(∗ set_pte_flags)(pte_t ∗, int)

Definition at line 697 of file vm.c.

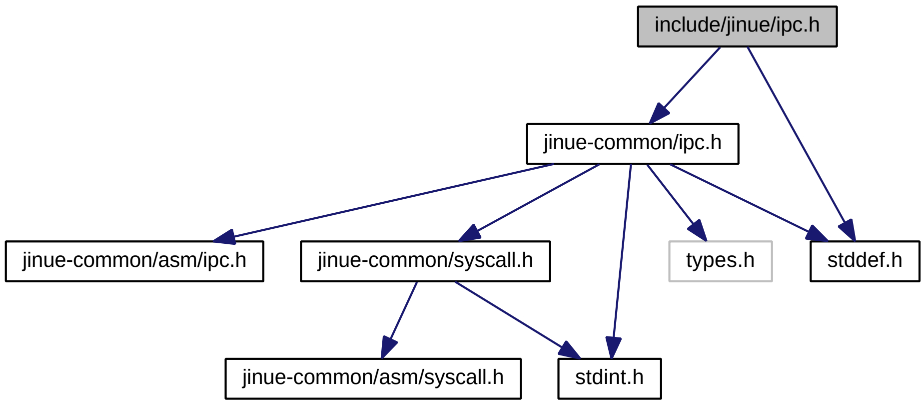
Referenced by vm_change_flags(), and vm_pae_boot_init().

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*)

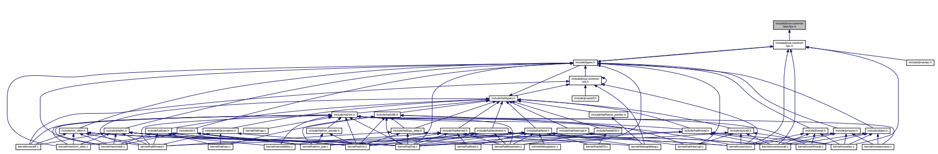
4.49.1.2 int jinue_receive (int *fd,* char ∗ *buffer,* size_t *buffer_size,* jinue_message_t ∗ *message,* int ∗ *perrno*)

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

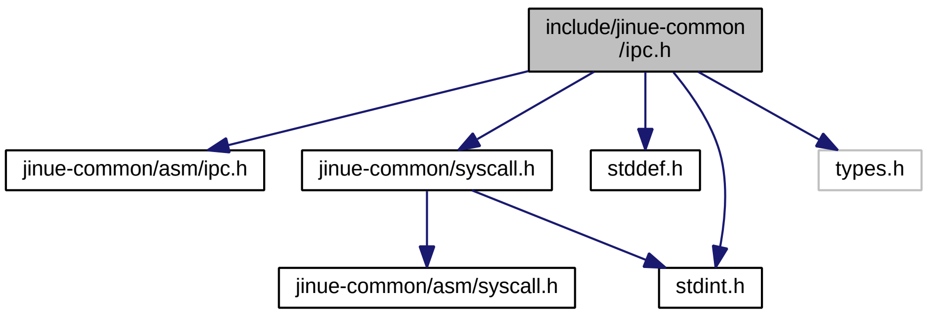
Definition at line 53 of file ipc.h.

Referenced by ipc_reply().

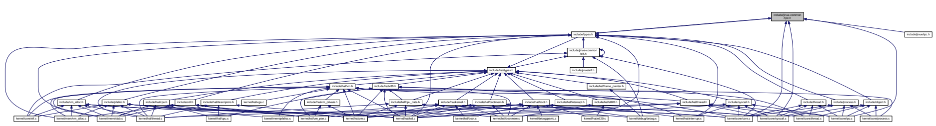
4.51 include/jinue-common/ipc.h File Reference

```
#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** (void)
- **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 (void)

Definition at line 58 of file ipc.c.

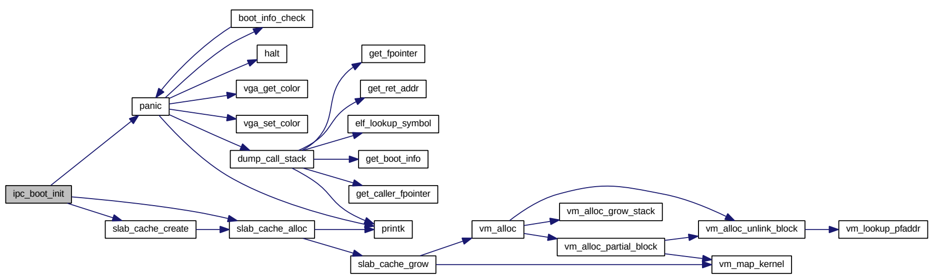
References NULL, panic(), slab_cache_alloc(), slab_cache_create(), and SLAB_DEFAULTS.

Referenced by kmain().

```
58                             {
59     ipc_object_cache = slab_cache_create(
60             "ipc_object_cache",
61             sizeof(ipc_t),
62             0,
63             ipc_object_ctor,
64             NULL,
65             SLAB_DEFAULTS );
66
```

```
67     proc_ipc = slab_cache_alloc(ipc_object_cache);
68 
69     if(proc_ipc == NULL) {
70         panic("Cannot create process manager IPC object.");
71     }
72 }
```

Here is the call graph for this function:



4.51.3.2 ipc_t∗ ipc_get_proc_object (void)

Definition at line 84 of file ipc.c.

Referenced by dispatch_syscall().

```
84                                 {
85     return proc_ipc;
86 }
```

4.51.3.3 ipc_t∗ ipc_object_create (int *flags*)

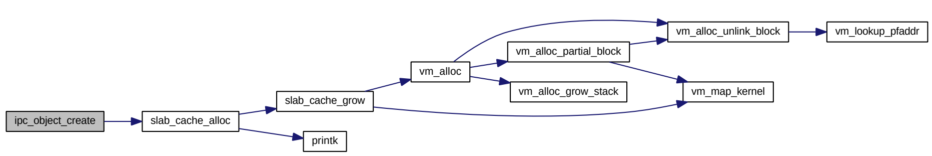
Definition at line 74 of file ipc.c.

References object_header_t::flags, ipc_t::header, NULL, and slab_cache_alloc().

Referenced by dispatch_syscall().

```
74                                    {
75     ipc_t *ipc = slab_cache_alloc(ipc_object_cache);
76 
77     if(ipc != NULL) {
78         ipc->header.flags = flags;
79     }
80 
81     return ipc;
82 }
```

Here is the call graph for this function:



4.51.3.4 void ipc_receive (jinue_syscall_args_t * *args*)

Definition at line 203 of file ipc.c.

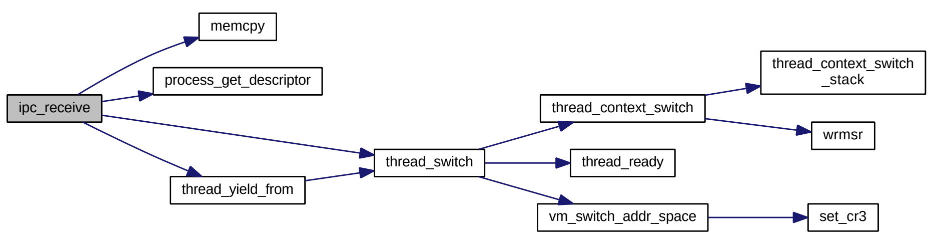
References jinue_syscall_args_t::arg0, jinue_syscall_args_t::arg1, jinue_syscall_args_t::arg2, jinue_syscall_args_t::arg3, message_info_t::data_size, object_header_t::flags, thread_t::header, JINUE_E2BIG, JINUE_EBADF, JINUE_EINVAL, JINUE_EIO, JINUE_EPERM, jinue_node_entry, memcpy(), thread_t::message_args, 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().

```
203                                         {
204     thread_t *thread = get_current_thread();
205 
206     int fd = (int)args->arg1;
207 
208     object_ref_t *ref = process_get_descriptor(thread->process, fd);
209 
210     if(! object_ref_is_valid(ref)) {
211         syscall_args_set_error(args, JINUE_EBADF);
212         return;
213     }
214 
215     if(object_ref_is_closed(ref)) {
216         syscall_args_set_error(args, JINUE_EIO);
217         return;
218     }
219 
220     if(! object_ref_is_owner(ref)) {
221         syscall_args_set_error(args, JINUE_EPERM);
222         return;
223     }
224 
225     object_header_t *header = ref->object;
226 
227     if(object_is_destroyed(header)) {
228         ref->flags |= OBJECT_REF_FLAG_CLOSED;
229         object_subref(header);
230 
231         syscall_args_set_error(args, JINUE_EIO);
232         return;
233     }
234 
235     if(header->type != OBJECT_TYPE_IPC) {
236         syscall_args_set_error(args, JINUE_EBADF);
237         return;
238     }
239 
240     ipc_t *ipc = (ipc_t *)header;
241 
242     char *user_ptr = (char *)args->arg2;
243     size_t buffer_size = jinue_args_get_buffer_size(args);
244 
245     if(! user_buffer_check(user_ptr, buffer_size)) {
246         syscall_args_set_error(args, JINUE_EINVAL);
247         return;
248     }
249 
250     thread_t *send_thread = jinue_node_entry(
251         jinue_list_dequeue(&ipc->send_list),
252         thread_t,
253         thread_list);
254 
255     if(send_thread == NULL) {
256         /* No thread is waiting to send a message, so we must wait on the receive
257          * list. */
258         jinue_list_enqueue(&ipc->recv_list, &thread->thread_list);
259 
260         thread_yield_from(
261                 thread,
262                 true,       /* make thread block */
263                 false);     /* don't destroy */
264 
265         /* set by sending thread */
```

```
266         send_thread = thread->sender;
267     }
268     else {
269         object_addref(&send_thread->header);
270         thread->sender = send_thread;
271     }
272 
273     if(send_thread->message_info.total_size > buffer_size) {
274         /* message is too big for receive buffer */
275         object_subref(&send_thread->header);
276         thread->sender = NULL;
277 
278         syscall_args_set_error(send_thread->message_args, JINUE_E2BIG);
279         syscall_args_set_error(args, JINUE_E2BIG);
280 
281         /* switch back to sender thread to return from call immediately */
282         thread_switch(
283                 thread,
284                 send_thread,
285                 false,      /* don't block (put this thread back in ready queue) */
286                 false);     /* don't destroy */
287 
288         return;
289     }
290 
291     memcpy(
292         user_ptr,
293         send_thread->message_buffer,
294         send_thread->message_info.data_size);
295 
296     args->arg0 = send_thread->message_args->arg0;
297     args->arg1 = ref->cookie;
298     /* argument 2 is left intact (buffer pointer) */
299     args->arg3 = send_thread->message_args->arg3;
300 }
```

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 302 of file ipc.c.

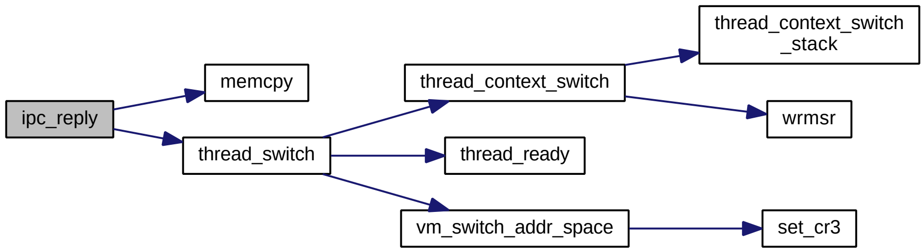
References jinue_syscall_args_t::arg2, jinue_syscall_args_t::arg3, message_info_t::buffer_size, message_info_t::data_size, message_info_t::desc_n, thread_t::header, JINUE_EINVAL, JINUE_ENOSYS, JINUE_SEND_BUFFER_SIZE_OFFSET, JINUE_SEND_MAX_N_DESC, JINUE_SEND_MAX_SIZE, JINUE_SEND_SIZE_MASK, memcpy(), thread_t::message_args, thread_t::message_buffer, thread_t::message_info, NULL, thread_t::sender, and thread_switch().

Referenced by dispatch_syscall().

```
302                                                {
```

```
    thread_t *thread        = get_current_thread();
    thread_t *send_thread   = thread->sender;

    if(send_thread == NULL) {
        syscall_args_set_error(args, JINUE_EINVAL);
        return;
    }

    size_t buffer_size   = jinue_args_get_buffer_size(args);
    size_t data_size     = jinue_args_get_data_size(args);
    size_t desc_n        = jinue_args_get_n_desc(args);
    size_t total_size    =
            data_size +
            desc_n * sizeof(jinue_ipc_descriptor_t);

    if(buffer_size > JINUE_SEND_MAX_SIZE) {
        syscall_args_set_error(args, JINUE_EINVAL);
        return;
    }

    if(total_size > buffer_size) {
        syscall_args_set_error(args, JINUE_EINVAL);
        return;
    }

    if(desc_n > JINUE_SEND_MAX_N_DESC) {
        syscall_args_set_error(args, JINUE_EINVAL);
        return;
    }

    /* the reply must fit in the sender's buffer */
    if(total_size > send_thread->message_info.buffer_size) {
        syscall_args_set_error(args, JINUE_EINVAL);
        return;
    }

    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 88 of file ipc.c.

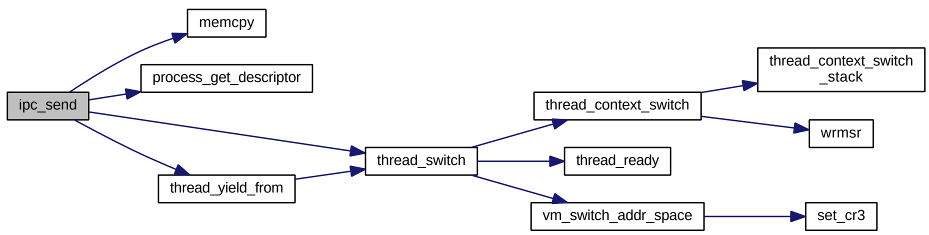
References jinue_syscall_args_t::arg0, jinue_syscall_args_t::arg1, jinue_syscall_args_t::arg2, message_info_t::buffer_size, message_info_t::cookie, message_info_t::data_size, message_info_t::desc_n, object_header_t::flags, message_info_t::function, thread_t::header, JINUE_EBADF, JINUE_EINVAL, JINUE_EIO, 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().

```
88                                          {
89      thread_t *thread = get_current_thread();
90
91      message_info_t *message_info = &thread->message_info;
92
93      message_info->function      = args->arg0;
94      message_info->buffer_size   = jinue_args_get_buffer_size(args);
95      message_info->data_size     = jinue_args_get_data_size(args);
96      message_info->desc_n        = jinue_args_get_n_desc(args);
97      message_info->total_size    =
98              message_info->data_size +
99              message_info->desc_n * sizeof(jinue_ipc_descriptor_t);
100
101      if(message_info->buffer_size > JINUE_SEND_MAX_SIZE) {
102          syscall_args_set_error(args, JINUE_EINVAL);
103          return;
104      }
105
106      if(message_info->total_size > message_info->buffer_size) {
107          syscall_args_set_error(args, JINUE_EINVAL);
108          return;
109      }
110
111      if(message_info->desc_n > JINUE_SEND_MAX_N_DESC) {
112          syscall_args_set_error(args, JINUE_EINVAL);
113          return;
114      }
115
117      if(message_info->desc_n > 0) {
118          syscall_args_set_error(args, JINUE_ENOSYS);
119          return;
120      }
121
122      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);

}
```

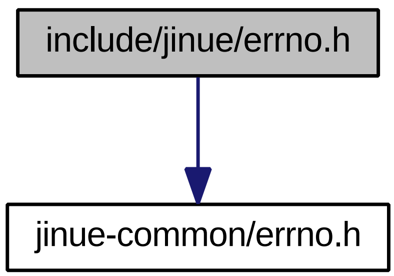
Here is the call graph for this function:



4.52 include/jinue/errno.h File Reference

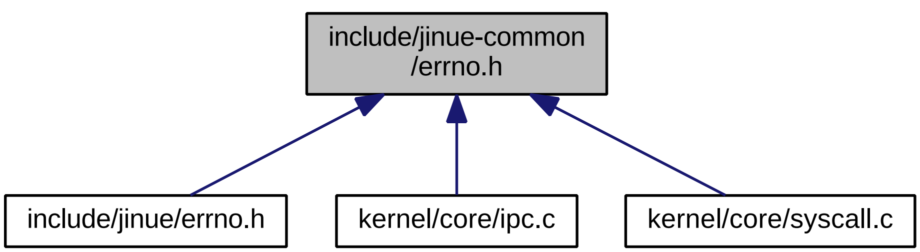
`#include <jinue-common/errno.h>`

Include dependency graph for errno.h:



4.53 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

4.53.1 Macro Definition Documentation

4.53.1.1 #define JINUE_E2BIG 9

Definition at line 51 of file errno.h.

Referenced by ipc_receive().

4.53.1.2 #define JINUE_EAGAIN 5

Definition at line 43 of file errno.h.

Referenced by dispatch_syscall().

4.53.1.3 #define JINUE_EBADF 6

Definition at line 45 of file errno.h.

Referenced by ipc_receive(), and ipc_send().

4.53.1.4 #define JINUE_EINVAL 4

Definition at line 41 of file errno.h.

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

4.53.1.5 #define JINUE_EIO 7

Definition at line 47 of file errno.h.

Referenced by ipc_receive(), and ipc_send().

4.53.1.6 #define JINUE_EMORE 1

Definition at line 35 of file errno.h.

Referenced by dispatch_syscall().

4.53.1.7 #define JINUE_ENOMEM 2

Definition at line 37 of file errno.h.

4.53.1.8 #define JINUE_ENOSYS 3

Definition at line 39 of file errno.h.

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

4.53.1.9 #define JINUE_EPERM 8

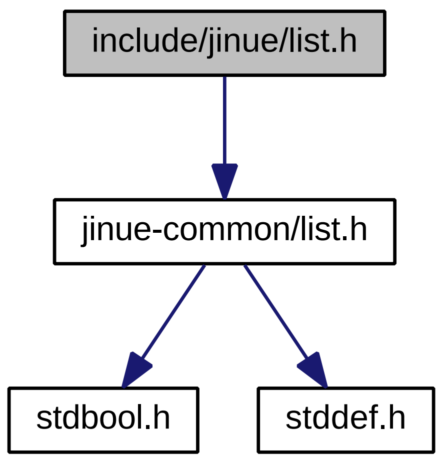
Definition at line 49 of file errno.h.

Referenced by ipc_receive().

4.54 include/jinue/list.h File Reference

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

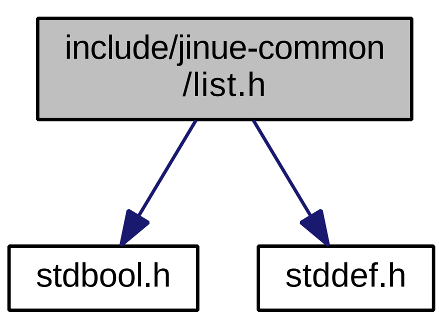
Include dependency graph for list.h:



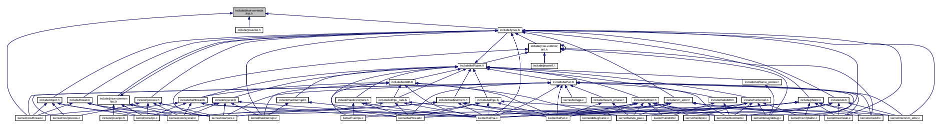
4.55 include/jinue-common/list.h File Reference

`#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**

4.55.1 Macro Definition Documentation

4.55.1.1 #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.

4.55.1.2 #define jinue_list_pop(*l*) (jinue_list_dequeue((l)))

Definition at line 121 of file list.h.

4.55.1.3 #define JINUE_LIST_STATIC {.head = NULL, .tail = NULL}

Definition at line 62 of file list.h.

4.55.1.4 #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().

4.55.1.5 #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.

4.55.2 Typedef Documentation

4.55.2.1 typedef jinue_node_t** jinue_cursor_t

Definition at line 60 of file list.h.

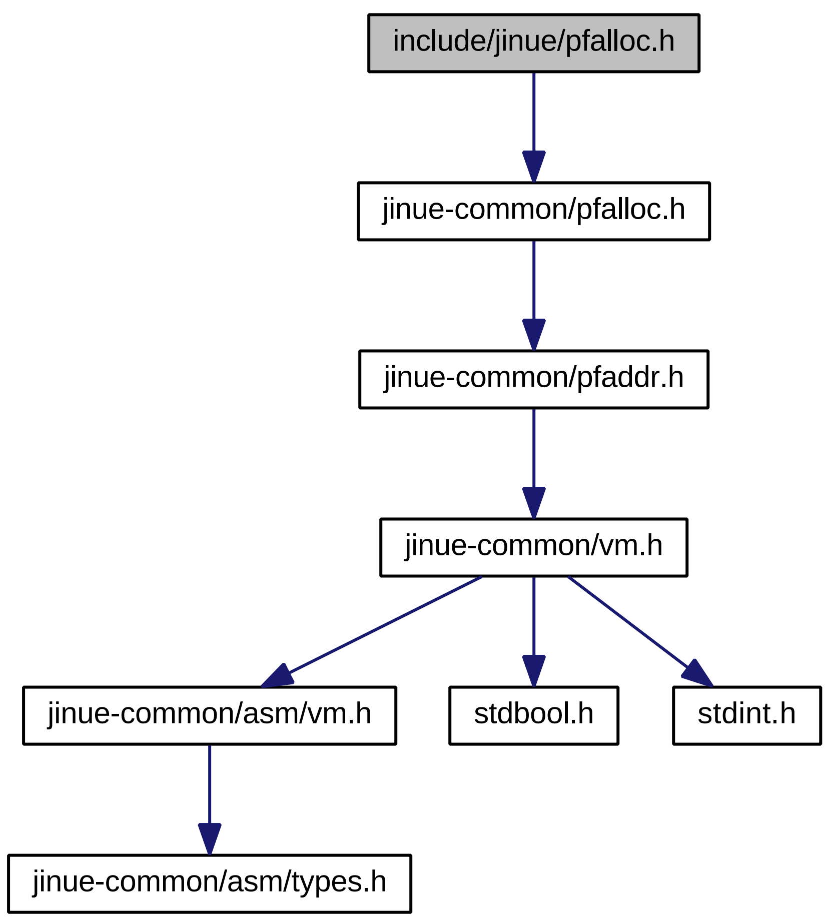
4.55.2.2 typedef struct jinue_node_t jinue_node_t

Definition at line 42 of file list.h.

4.56 include/jinue/pfalloc.h File Reference

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

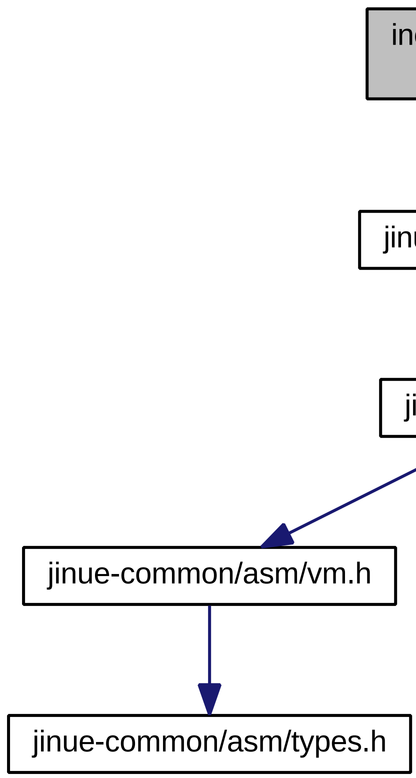
Include dependency graph for pfalloc.h:



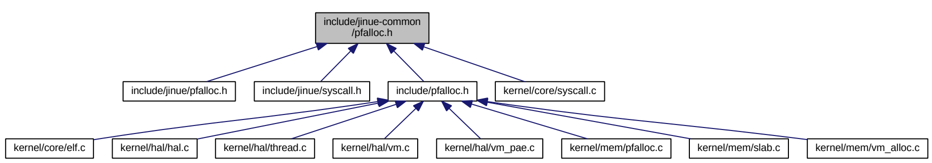
4.57 include/jinue-common/pfalloc.h File Reference

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

Include dependency graph for pfalloc.h:



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



Data Structures

- struct **memory_block_t**

Macros

- #define **KERNEL_PAGE_STACK_SIZE** 1024
- #define **KERNEL_PAGE_STACK_INIT** 128

4.57.1 Macro Definition Documentation

4.57.1.1 #define KERNEL_PAGE_STACK_INIT 128

Definition at line 39 of file pfalloc.h.

Referenced by hal_init().

4.57.1.2 #define KERNEL_PAGE_STACK_SIZE 1024

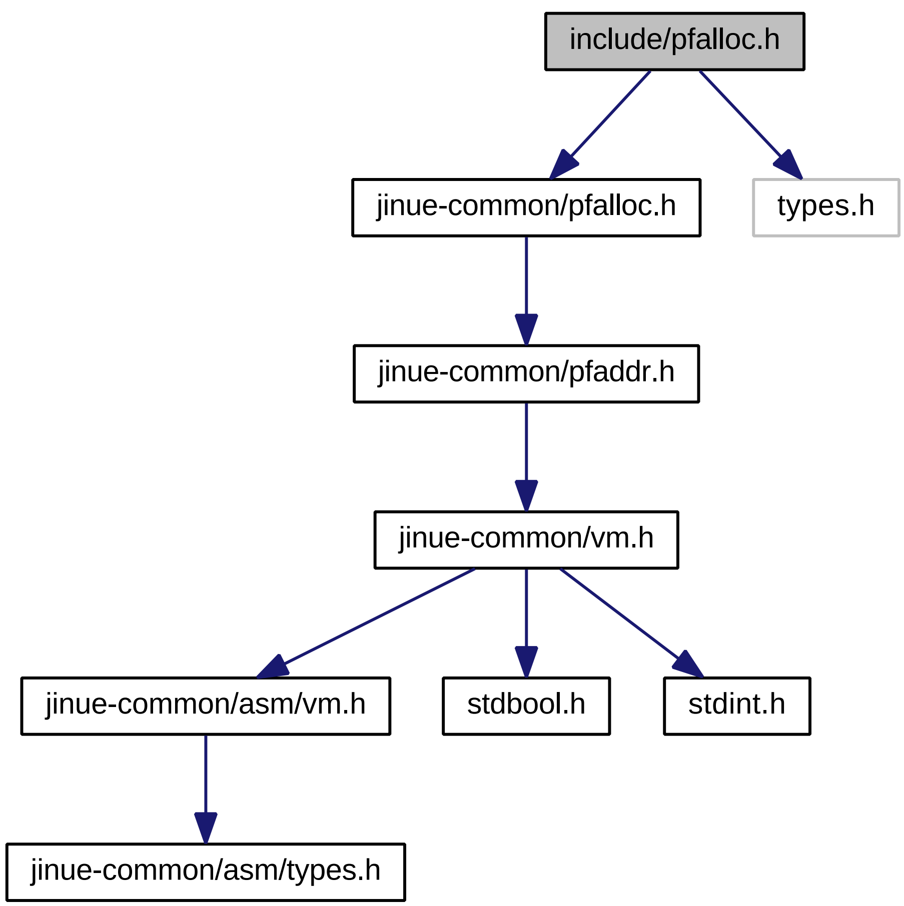
Definition at line 37 of file pfalloc.h.

Referenced by init_pfcache(), and pffree_to().

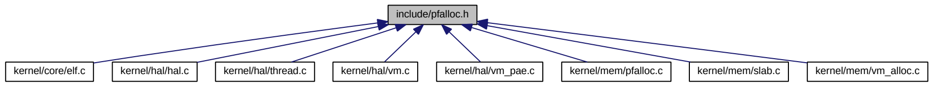
4.58 include/pfalloc.h File Reference

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

Include dependency graph for pfalloc.h:



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



Data Structures

- struct **pfcache_t**

Macros

- #define **pfalloc**() **pfalloc_from**(&**global_pfcache**)
- #define **pffree**(p) **pffree_to**(&**global_pfcache**, (p))

Functions

- **addr_t pfalloc_early** (void)
- void **init_pfcache** (**pfcache_t** ∗pfcache, **pfaddr_t** ∗stack_page)
- **pfaddr_t pfalloc_from** (**pfcache_t** ∗pfcache)
- void **pffree_to** (**pfcache_t** ∗pfcache, **pfaddr_t** pf)

Variables

- **bool use_pfalloc_early**
- **pfcache_t global_pfcache**

4.58.1 Macro Definition Documentation

4.58.1.1 #define pfalloc( ) pfalloc_from(&global_pfcache)

Definition at line 50 of file pfalloc.h.

Referenced by elf_load(), elf_setup_stack(), slab_cache_grow(), thread_page_create(), vm_alloc_init_allocator(), vm_-alloc_partial_block(), and vm_clone_page_directory().

4.58.1.2 #define pffree(*p*) pffree_to(&global_pfcache, (p))

Definition at line 52 of file pfalloc.h.

Referenced by hal_init(), thread_page_destroy(), vm_alloc_destroy(), vm_alloc_unlink_block(), and vm_destroy_page-_directory().

4.58.2 Function Documentation

4.58.2.1 void init_pfcache (pfcache_t ∗ *pfcache*, pfaddr_t ∗ *stack_page*)

Definition at line 58 of file pfalloc.c.

References pfcache_t::count, KERNEL_PAGE_STACK_SIZE, PFNULL, and pfcache_t::ptr.

Referenced by hal_init().

```
58                                                              {
59     pfaddr_t *ptr;
60     unsigned int idx;
61 
62     ptr = stack_page;
63 
64     for(idx = 0;idx < KERNEL_PAGE_STACK_SIZE; ++idx) {
65         ptr[idx] = PFNULL;
66     }
67 
68     pfcache->ptr   = stack_page;
69     pfcache->count = 0;
70 }
```

4.58.2.2 addr_t pfalloc_early (void)

ASSERTION: pfalloc_early is used early only

Definition at line 46 of file pfalloc.c.

References assert, kernel_region_top, PAGE_SIZE, and use_pfalloc_early.

Referenced by hal_init(), and vm_allocate_page_directory().

```
46                             {
47     addr_t page;
48 
50     assert(use_pfalloc_early);
51 
52     page = kernel_region_top;
53     kernel_region_top += PAGE_SIZE;
54 
55     return page;
56 }
```

4.58.2.3 pfaddr_t pfalloc_from (pfcache_t ∗ *pfcache*)

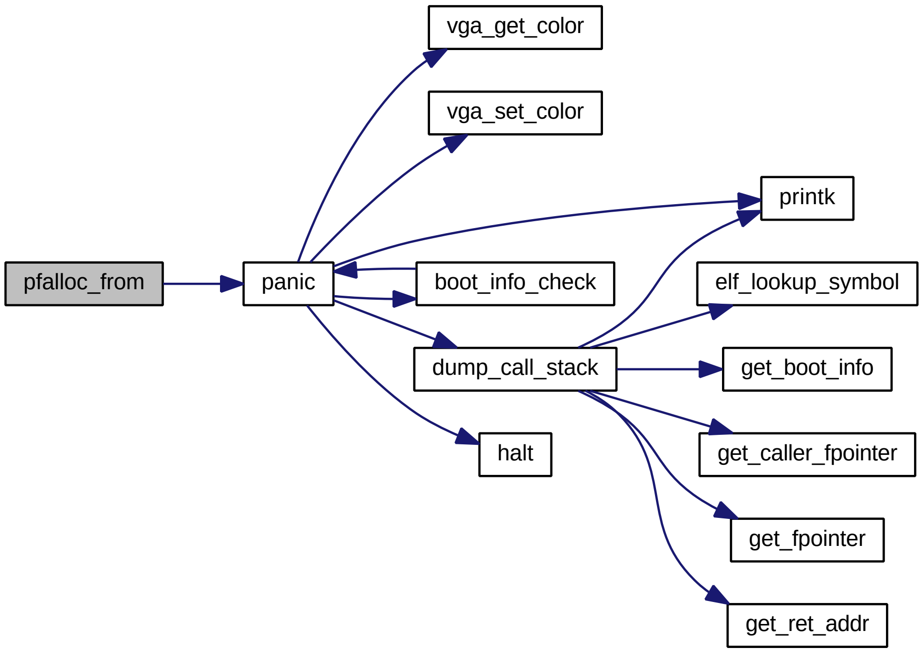
ASSERTION: pfalloc_early must be used early

Definition at line 72 of file pfalloc.c.

References assert, pfcache_t::count, panic(), pfcache_t::ptr, and use_pfalloc_early.

```
72                                             {
74     assert( ! use_pfalloc_early );
75 
76     if(pfcache->count == 0) {
77         panic("pfalloc_from(): no more pages to allocate");
78     }
79 
80     --pfcache->count;
81 
82     return *(--pfcache->ptr);
83 }
```

Here is the call graph for this function:



4.58.2.4 void pffree_to (pfcache_t * *pfcache*, pfaddr_t *pf*)

We are leaking memory here. Should we panic instead?

Definition at line 85 of file pfalloc.c.

References pfcache_t::count, KERNEL_PAGE_STACK_SIZE, and pfcache_t::ptr.

```
85                                             {
86      if(pfcache->count >= KERNEL_PAGE_STACK_SIZE) {
88          return;
89      }
90
91      ++pfcache->count;
92
93      (pfcache->ptr++)[0] = pf;
94 }
```

4.58.3 Variable Documentation

4.58.3.1 pfcache_t global_pfcache

Definition at line 43 of file pfalloc.c.

Referenced by hal_init().

4.58.3.2 bool use_pfalloc_early

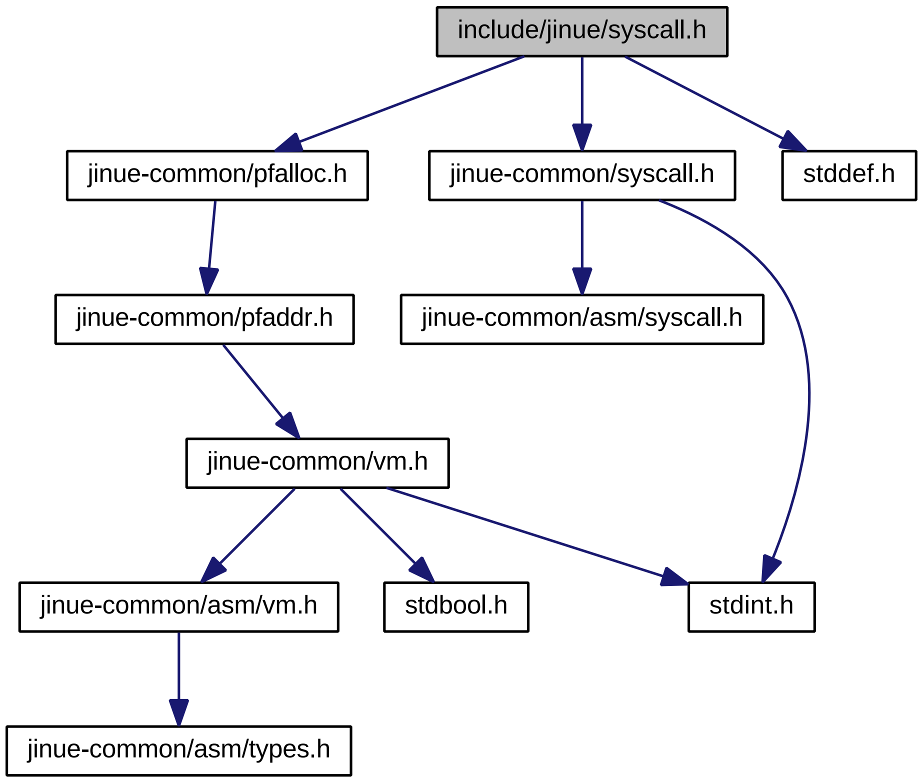
Definition at line 41 of file pfalloc.c.

Referenced by hal_init(), pfalloc_early(), pfalloc_from(), and vm_boot_init().

4.59 include/jinue/syscall.h File Reference

```
#include <jinue-common/pfalloc.h>
#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_get_free_memory** (**memory_block_t** ∗buffer, **size_t** buffer_size, int ∗perrno)
- int **jinue_thread_create** (void(∗entry)(), void ∗stack, int ∗perrno)
- int **jinue_yield** (void)
- void **jinue_thread_exit** (void)

4.59.1 Function Documentation

4.59.1.1 int jinue_call (**jinue_syscall_args_t** ∗ *args,* int ∗ *perrno*)

4.59.1.2 void jinue_call_raw (**jinue_syscall_args_t** ∗ *args*)

4.59.1.3 int jinue_get_free_memory (**memory_block_t** ∗ *buffer,* **size_t** *buffer_size,* int ∗ *perrno*)

4.59.1.4 void jinue_get_syscall_implementation (void)

4.59.1.5 const char∗ jinue_get_syscall_implementation_name (void)

4.59.1.6 void∗ jinue_get_thread_local_storage (void)

4.59.1.7 void jinue_set_thread_local_storage (void ∗ *addr,* **size_t** *size*)

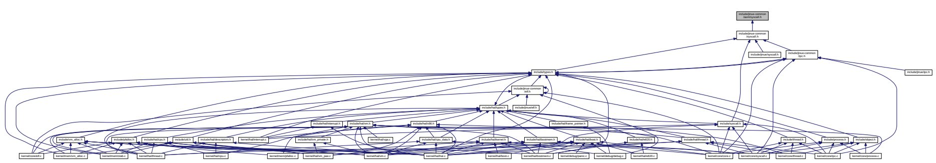
4.59.1.8 int jinue_thread_create (void(∗)() *entry,* void ∗ *stack,* int ∗ *perrno*)

4.59.1.9 void jinue_thread_exit (void)

4.59.1.10 int jinue_yield (void)

4.60 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_FREE_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

4.60.1 Macro Definition Documentation

4.60.1.1 #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().

4.60.1.2 #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().

4.60.1.3 #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().

4.60.1.4 #define SYSCALL_FUNCT_GET_FREE_MEMORY 8

get free memory block list for management by process manager

Definition at line 60 of file syscall.h.

Referenced by dispatch_syscall().

4.60.1.5 #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().

4.60.1.6 #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().

4.60.1.7 #define SYSCALL_FUNCT_RECEIVE 10

receive a message on an IPC object

Definition at line 66 of file syscall.h.

Referenced by dispatch_syscall().

4.60.1.8 #define SYSCALL_FUNCT_REPLY 11

reply to current message

Definition at line 69 of file syscall.h.

Referenced by dispatch_syscall().

4.60.1.9 #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().

4.60.1.10 #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().

4.60.1.11 #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().

4.60.1.12 #define SYSCALL_FUNCT_THREAD_CREATE 4

create a new thread

Definition at line 48 of file syscall.h.

Referenced by dispatch_syscall().

4.60.1.13 #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().

4.60.1.14 #define SYSCALL_FUNCT_USER_BASE 0x4000

start of function numbers for user IPC objects

Definition at line 78 of file syscall.h.

4.60.1.15 #define SYSCALL_IRQ 0x80

interrupt vector for system call software interrupt

Definition at line 36 of file syscall.h.

Referenced by dispatch_interrupt(), and hal_init().

4.60.1.16 #define SYSCALL_METHOD_FAST_AMD 1

AMD's fast system call method (SYSCALL/SYSLEAVE)

Definition at line 85 of file syscall.h.

Referenced by hal_init().

4.60.1.17 #define SYSCALL_METHOD_FAST_INTEL 0

Intel's fast system call method (SYSENTER/SYSEXIT)

Definition at line 82 of file syscall.h.

Referenced by hal_init().

4.60.1.18 #define SYSCALL_METHOD_INTR 2

slow/safe system call method using interrupts

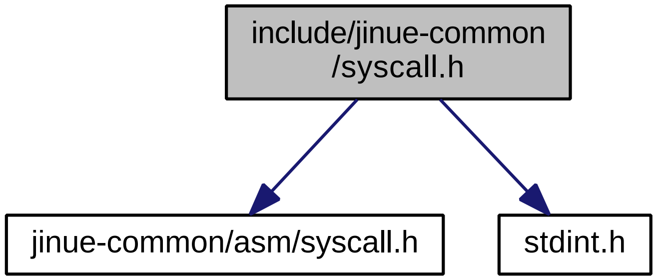
Definition at line 88 of file syscall.h.

Referenced by hal_init().

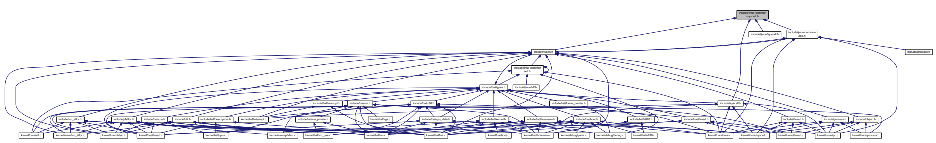
4.61 include/jinue-common/syscall.h File Reference

```
#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:



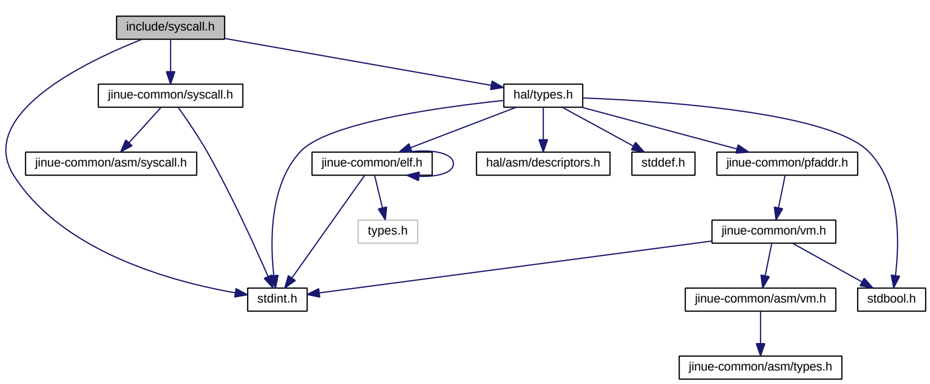
Data Structures

- struct **jinue_syscall_args_t**

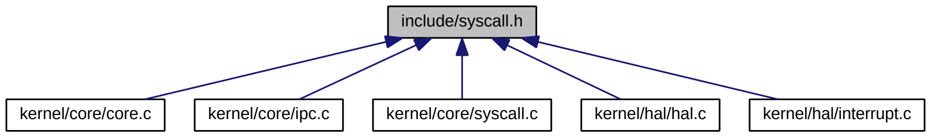
4.62 include/syscall.h File Reference

```
#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)

4.62.1 Function Documentation

4.62.1.1 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 49 of file syscall.c.

References bootmem_t::addr, memory_block_t::addr, jinue_syscall_args_t::arg0, jinue_syscall_args_t::arg1, jinue_syscall_args_t::arg2, jinue_syscall_args_t::arg3, bootmem_get_block(), bootmem_root, console_printn(), console_putc(), object_ref_t::cookie, bootmem_t::count, memory_block_t::count, object_ref_t::flags, 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_EMORE, JINUE_ENOSYS, JINUE_IPC_PROC, JINUE_IPC_SYSTEM, NULL, object_ref_t::object, OBJECT_REF_FLAG_OWNER, OBJECT_REF_FLAG_VALID, printk(), thread_t::process, process_get_descriptor(), process_unused_descriptor(), SYSCALL_FUNCT_CONSOLE_PUTC, SYSCALL_FUNCT_CONSOLE_PUTS, SYSCALL_FUNCT_CREATE_IPC, SYSCALL_FUNCT_GET_FREE_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(), and thread_yield_from().

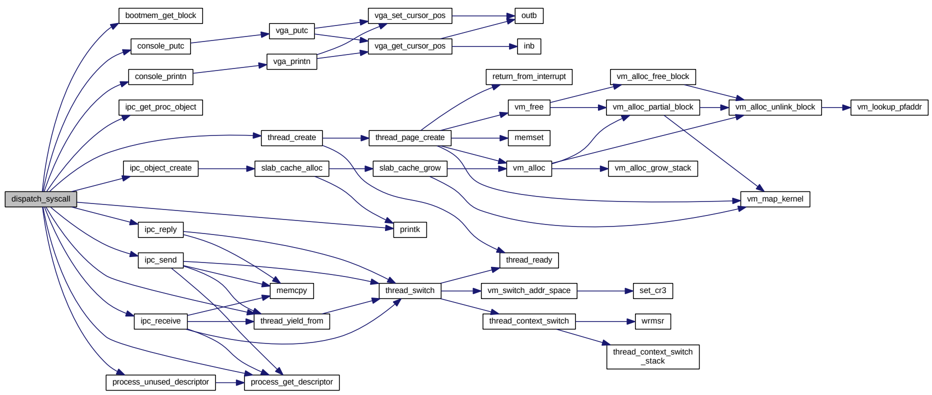
Referenced by dispatch_interrupt().

```
49                                              {
50      jinue_syscall_args_t *args = (jinue_syscall_args_t *)&trapframe->msg_arg0;
51
53      uintptr_t function_number = args->arg0;
54
55      if(function_number < SYSCALL_FUNCT_PROC_BASE) {
56          /* microkernel system calls */
57          switch(function_number) {
58
59          case SYSCALL_FUNCT_SYSCALL_METHOD:
60              syscall_args_set_return(args, syscall_method);
61              break;
62
63          case SYSCALL_FUNCT_CONSOLE_PUTC:
65              console_putc((char)args->arg1);
66              syscall_args_set_return(args, 0);
67              break;
68
69          case SYSCALL_FUNCT_CONSOLE_PUTS:
71              console_printn((char *)args->arg2, jinue_args_get_data_size(args));
72              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_FREE_MEMORY:
        {
            bootmem_t      *block;
            memory_block_t *block_dest;
            unsigned int count, count_max;

            size_t buffer_size  = jinue_args_get_buffer_size(args);
            block_dest          = (memory_block_t *)jinue_args_get_buffer_ptr(args);

            count_max = buffer_size / sizeof(memory_block_t);

            for(count = 0; count < count_max; ++count) {
                block = bootmem_get_block();

                if(block == NULL) {
                    break;
                }

                block_dest->addr  = block->addr;
                block_dest->count = block->count;

                ++block_dest;
            }

            args->arg0 = (uintptr_t)count;

            if(count == count_max && bootmem_root != NULL) {
                args->arg1  = JINUE_EMORE;
            }
            else {
                args->arg1  = 0;
            }

            args->arg2 = 0;
            args->arg3 = 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:



4.63 include/kbd.h File Reference

Functions

- void **any_key** (void)

4.63.1 Function Documentation

4.63.1.1 void any_key (void)

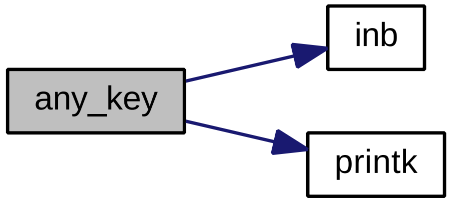
Definition at line 36 of file kbd.c.

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 );
49 
50         buffer = inb(0x60);
51 
52         if(buffer == 0x0e || buffer == 0x0f) {
53             ignore = true;
54             continue;
55         }
56 
57         if(ignore) {
58             ignore = false;
59             continue;
60         }
61 
62         if(buffer == 0x1c || buffer == 0x5a) {
63             break;
64         }
65     }
66 
67     /* advance cursor */
```

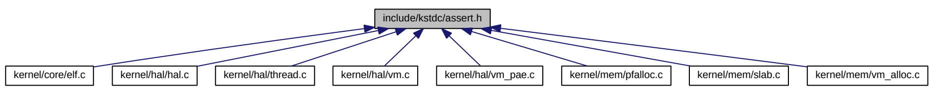
```
    printk("\n");
}
```

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(), pfalloc_early(), pfalloc_from(), slab_cache_alloc(), slab_cache_create(), slab_cache_destroy(), slab_cache_grow(), thread_context_switch(), vm_alloc(), vm_alloc_custom_block(), vm_alloc_free_block(), vm_alloc_grow_single(), vm_alloc_grow_stack(), vm_alloc_init_allocator(), vm_alloc_low_latency(), vm_alloc_partial_block(), vm_alloc_unlink_block(), vm_change_flags(), vm_destroy_addr_space(), vm_free(), vm_lookup_pfaddr(), vm_map_early(), and vm_unmap().

4.64.2 Function Documentation

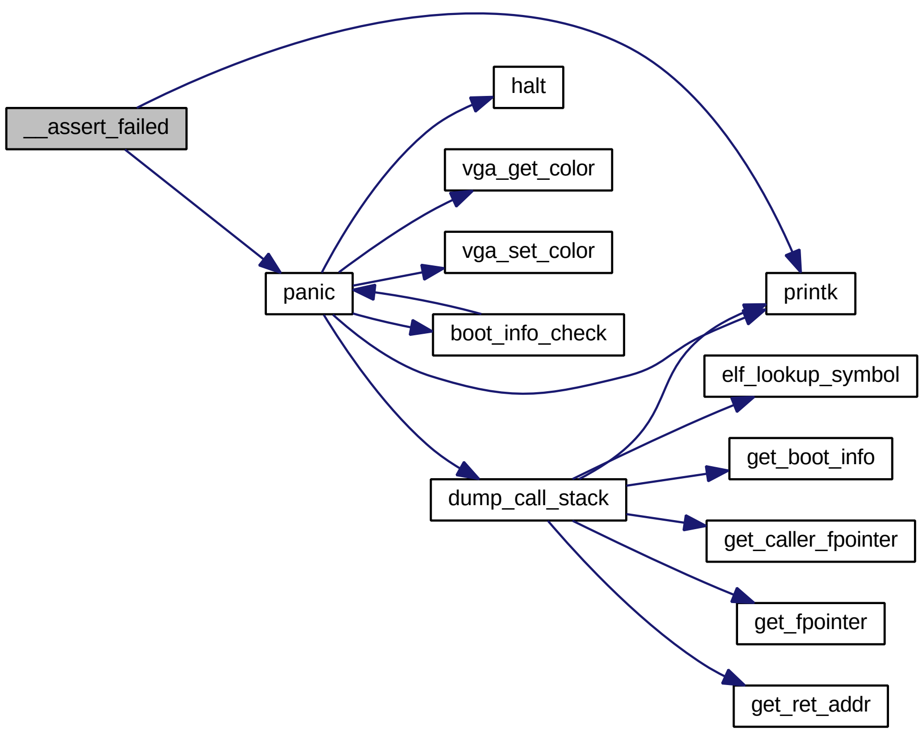
4.64.2.1 void __assert_failed (const char ∗ *expr,* const char ∗ *file,* unsigned int *line,* const char ∗ *func*)

Definition at line 36 of file 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.

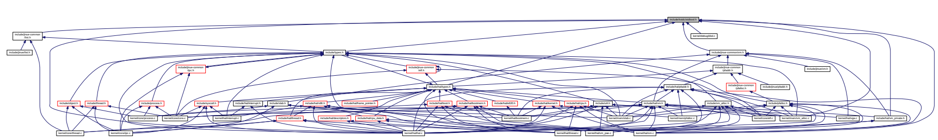
4.65.2 Typedef Documentation

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

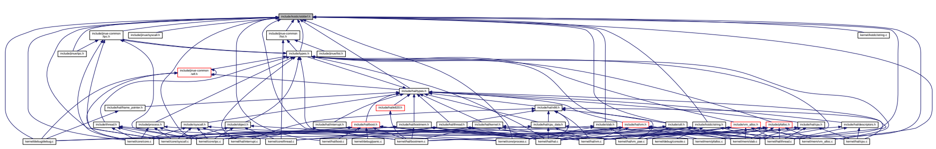
Definition at line 37 of file stdbool.h.

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 apply_mem_hole(), boot_info_check(), bootmem_get_block(), bootmem_init(), cpu_init_data(), dispatch_syscall(), dump_call_stack(), elf_lookup_symbol(), hal_init(), ipc_boot_init(), ipc_object_create(), ipc_receive(), ipc_reply(), ipc_send(), kmain(), process_boot_init(), process_create(), process_get_descriptor(), slab_cache_create(), slab_cache_grow(), thread_context_switch(), thread_create(), thread_page_create(), thread_page_destroy(), thread-_switch(), vm_alloc(), vm_alloc_custom_block(), vm_alloc_destroy(), vm_alloc_free_block(), vm_alloc_grow_single(), vm_alloc_grow_stack(), vm_alloc_init_allocator(), vm_alloc_low_latency(), vm_alloc_partial_block(), vm_alloc_unlink_-block(), vm_change_flags(), vm_destroy_addr_space(), vm_free(), vm_lookup_pfaddr(), vm_map_kernel(), vm_pae_-create_pdpt_cache(), vm_unmap(), 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

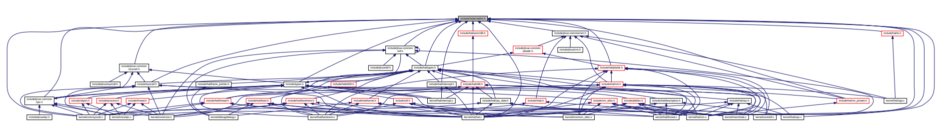
Definition at line 36 of file stddef.h.

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.

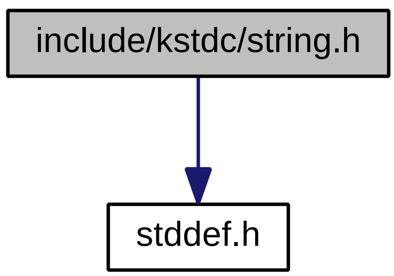
4.68.2.10 typedef unsigned int uintptr_t

Definition at line 59 of file stdint.h.

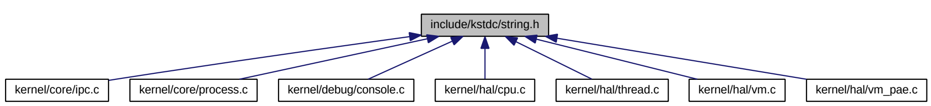
4.69 include/kstdc/string.h File Reference

```
#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)

4.69.1 Function Documentation

4.69.1.1 void∗ memcpy (void ∗ *dest,* const void ∗ *src,* size_t *n*)

Definition at line 45 of file string.c.

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

```
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 }
```

4.69.1.2 void* memset (void * *s*, int *c*, size_t *n*)

Definition at line 34 of file string.c.

Referenced by cpu_init_data(), process_create(), and thread_page_create().

```
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 }
```

4.69.1.3 size_t strlen (const char * *s*)

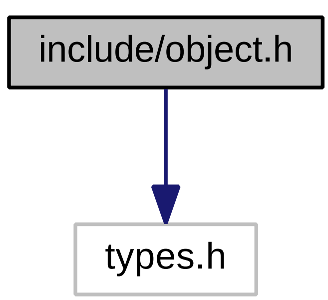
Definition at line 57 of file string.c.

Referenced by console_print().

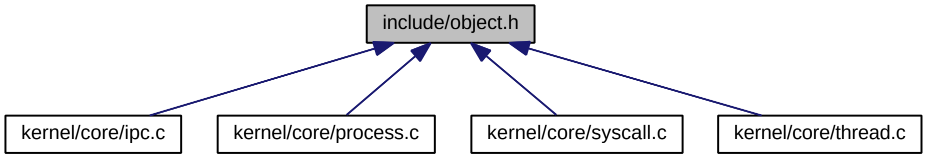
```
57                        {
58     size_t count = 0;
59
60     while(*s != 0) {
61         ++s;
62         ++count;
63     }
64
65     return count;
66 }
```

4.70 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

4.70.1 Macro Definition Documentation

4.70.1.1 #define OBJECT_FLAG_DESTROYED (1<<0)

Definition at line 42 of file object.h.

4.70.1.2 #define OBJECT_FLAG_NONE 0

Definition at line 40 of file object.h.

4.70.1.3 #define OBJECT_REF_FLAG_CLOSED (1<<1)

Definition at line 49 of file object.h.

Referenced by ipc_receive(), and ipc_send().

4.70.1.4 #define OBJECT_REF_FLAG_NONE 0

Definition at line 45 of file object.h.

4.70.1.5 #define OBJECT_REF_FLAG_OWNER (1<<2)

Definition at line 51 of file object.h.

Referenced by dispatch_syscall().

4.70.1.6 #define OBJECT_REF_FLAG_VALID (1<<0)

Definition at line 47 of file object.h.

Referenced by dispatch_syscall().

4.70.1.7 #define OBJECT_TYPE_IPC 2

Definition at line 56 of file object.h.

Referenced by ipc_receive(), and ipc_send().

4.70.1.8 #define OBJECT_TYPE_PROCESS 3

Definition at line 58 of file object.h.

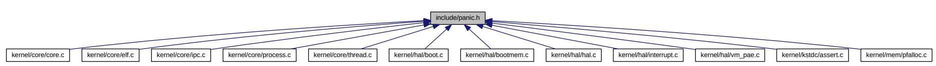
4.70.1.9 #define OBJECT_TYPE_THREAD 1

Definition at line 54 of file object.h.

Referenced by thread_create().

4.71 include/panic.h File Reference

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



Functions

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

4.71.1 Function Documentation

4.71.1.1 void panic (const char * *message*)

Definition at line 39 of file panic.c.

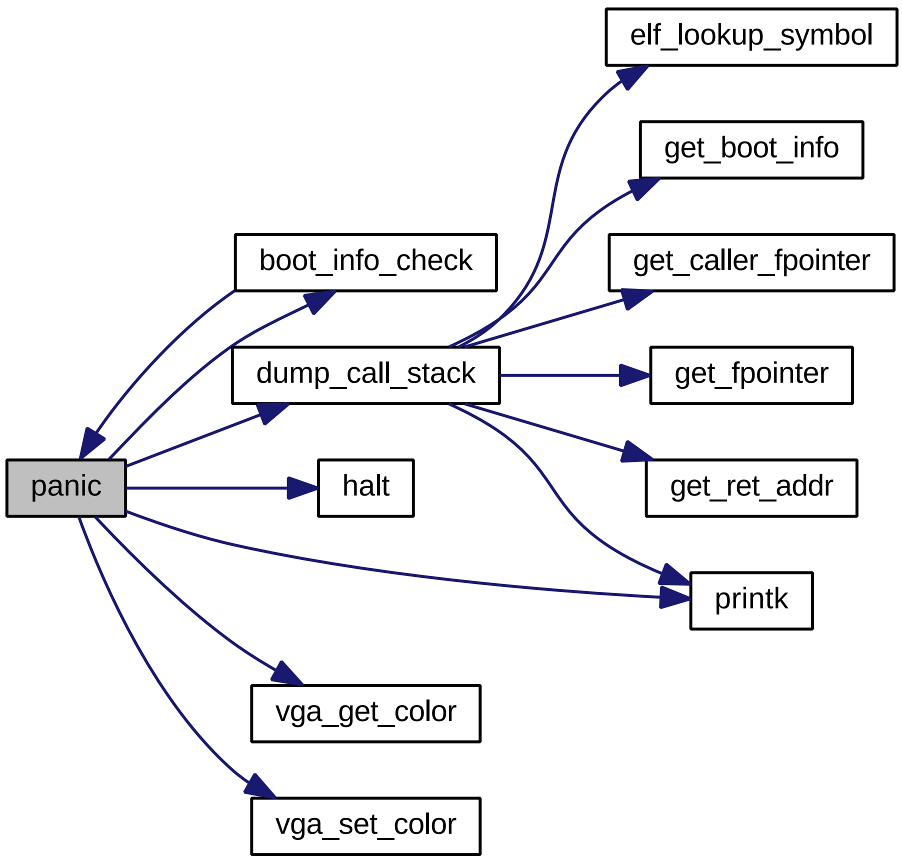
References boot_info_check(), dump_call_stack(), halt(), printk(), VGA_COLOR_RED, vga_get_color(), and vga_set_-color().

Referenced by __assert_failed(), boot_info_check(), bootmem_init(), dispatch_interrupt(), elf_check(), elf_load(), ipc_-boot_init(), kmain(), pfalloc_from(), process_boot_init(), and vm_pae_create_pdpt_cache().

```
39                                {
40     unsigned int color;
41 
42     color = vga_get_color();
43     vga_set_color(VGA_COLOR_RED);
44 
45     printk("KERNEL PANIC: %s\n", message);
```

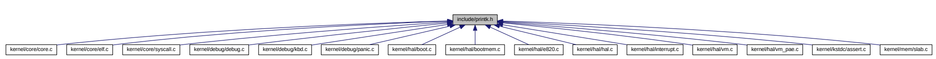
```
    vga_set_color(color);

    if( boot_info_check(false) ) {
        dump_call_stack();
    }
    else {
        printk("Cannot dump call stack because boot information structure is invalid.\n");
    }

    halt();
}
```

Here is the call graph for this function:



4.72 include/printk.h File Reference

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



Functions

- void **printk** (const char ∗format,...)

- void **print_unsigned_int** (unsigned int n)
- void **print_hex_nibble** (unsigned char byte)
- void **print_hex_b** (unsigned char byte)
- void **print_hex_w** (unsigned short word)
- void **print_hex_l** (unsigned long dword)
- void **print_hex_q** (unsigned long long qword)

4.72.1 Function Documentation

4.72.1.1 void print_hex_b (unsigned char *byte*)

4.72.1.2 void print_hex_l (unsigned long *dword*)

4.72.1.3 void print_hex_nibble (unsigned char *byte*)

4.72.1.4 void print_hex_q (unsigned long long *qword*)

4.72.1.5 void print_hex_w (unsigned short *word*)

4.72.1.6 void print_unsigned_int (unsigned int *n*)

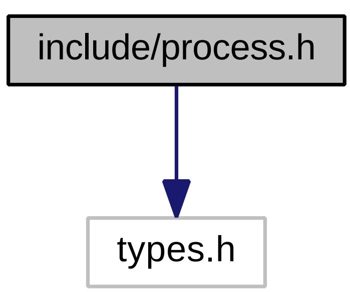
4.72.1.7 void printk (const char * *format*, ...)

Referenced by __assert_failed(), any_key(), boot_info_dump(), bootmem_init(), dispatch_interrupt(), dispatch_syscall(), dump_call_stack(), e820_dump(), elf_load(), hal_init(), kmain(), panic(), slab_cache_alloc(), slab_cache_free(), and vm_boot_init().

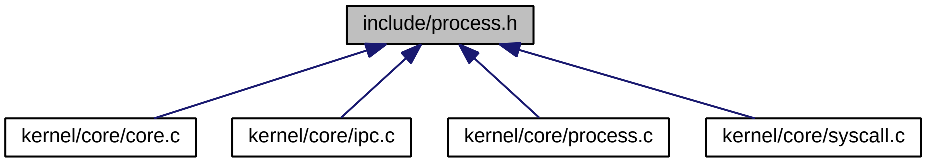
4.73 include/process.h File Reference

`#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** (void)
- **process_t** ∗ **process_create** (void)
- **object_ref_t** ∗ **process_get_descriptor** (**process_t** ∗process, int fd)
- int **process_unused_descriptor** (**process_t** ∗process)

4.73.1 Function Documentation

4.73.1.1 void process_boot_init (void)

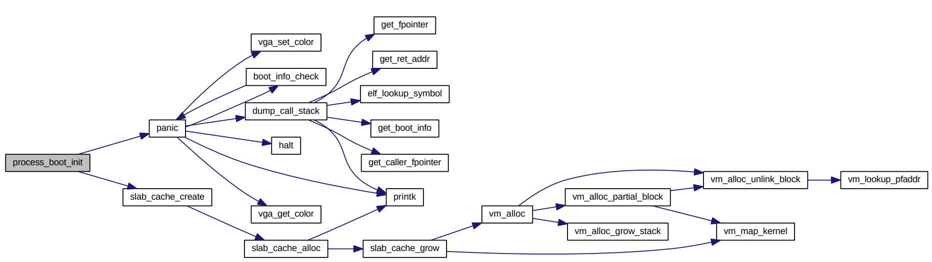
Definition at line 49 of file process.c.

References NULL, panic(), slab_cache_create(), and SLAB_DEFAULTS.

Referenced by kmain().

```
49                             {
50     process_cache = slab_cache_create(
51             "process_cache",
52             sizeof(process_t),
53             0,
54             process_ctor,
55             NULL,
56             SLAB_DEFAULTS );
57 
58     if(process_cache == NULL) {
59         panic("Cannot create process slab cache.");
60     }
61 }
```

Here is the call graph for this function:



4.73.1.2 process_t∗ process_create (void)

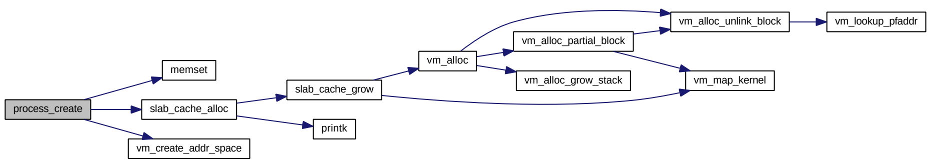
Definition at line 63 of file process.c.

References process_t::addr_space, process_t::descriptors, memset(), NULL, slab_cache_alloc(), and vm_create_addr_space().

Referenced by kmain().

```
63                              {
64     process_t *process = slab_cache_alloc(process_cache);
65
66     if(process != NULL) {
67         vm_create_addr_space(&process->addr_space);
68         memset(&process->descriptors, 0, sizeof(process->descriptors));
69     }
70
71     return process;
72 }
```

Here is the call graph for this function:



4.73.1.3 object_ref_t∗ process_get_descriptor (process_t ∗ *process,* int *fd*)

Definition at line 74 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().

```
74                                                             {
75     if(fd < 0 || fd > PROCESS_MAX_DESCRIPTORS) {
76         return NULL;
77     }
78
79     return &process->descriptors[fd];
80 }
```

4.73.1.4 int process_unused_descriptor (process_t ∗ *process*)

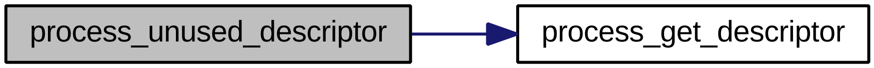
Definition at line 82 of file process.c.

References process_get_descriptor(), and PROCESS_MAX_DESCRIPTORS.

Referenced by dispatch_syscall().

```
82                                                {
83     int idx;
84
85     for(idx = 0; idx < PROCESS_MAX_DESCRIPTORS; ++idx) {
86         object_ref_t *ref = process_get_descriptor(process, idx);
87
88         if(! object_ref_is_valid(ref)) {
89             return idx;
90         }
91     }
92
93     return -1;
94 }
```

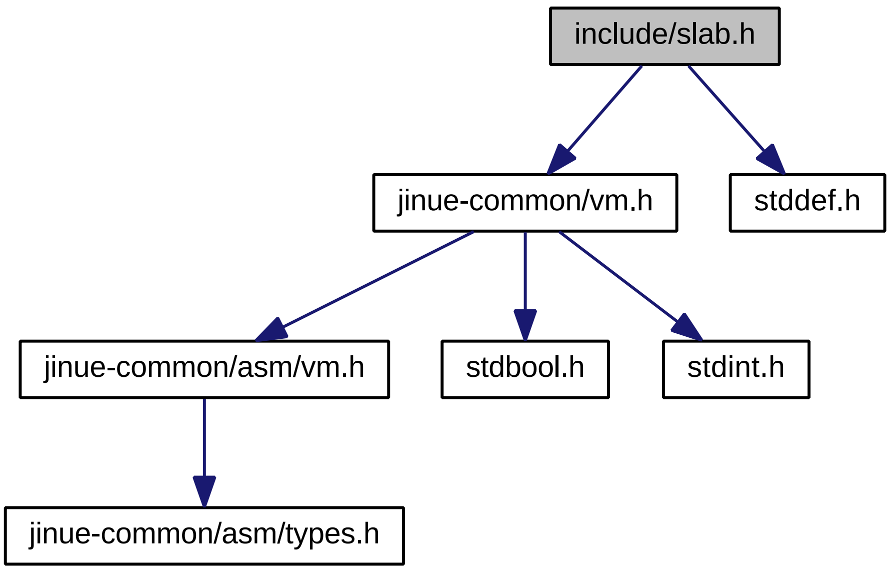
Here is the call graph for this function:



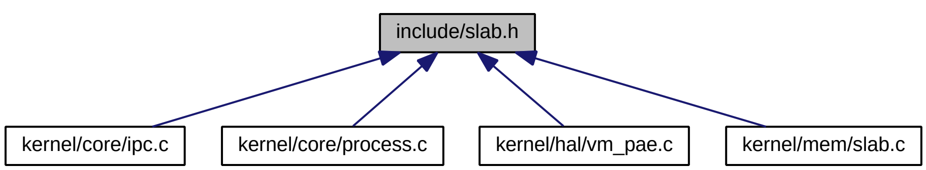
4.74 include/slab.h File Reference

```
#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

- **slab_cache_t** ∗ **slab_cache_create** (char ∗name, **size_t** size, **size_t** alignment, **slab_ctor_t** ctor, **slab_ctor_t** dtor, int flags)
- void **slab_cache_destroy** (**slab_cache_t** ∗cache)
- void ∗ **slab_cache_alloc** (**slab_cache_t** ∗cache)
- void **slab_cache_free** (void ∗buffer)
- void **slab_cache_grow** (**slab_cache_t** ∗cache)
- void **slab_cache_reap** (**slab_cache_t** ∗cache)
- void **slab_cache_set_working_set** (**slab_cache_t** ∗cache, unsigned int n)

Variables

- **slab_cache_t** ∗ **slab_cache_list**

4.74.1 Macro Definition Documentation

4.74.1.1 #define SLAB_COMPACT (1<<3)

Definition at line 57 of file slab.h.

Referenced by slab_cache_create().

4.74.1.2 #define SLAB_DEFAULT_WORKING_SET 2

Definition at line 46 of file slab.h.

Referenced by slab_cache_create().

4.74.1.3 #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().

4.74.1.4 #define SLAB_HWCACHE_ALIGN (1<<2)

Definition at line 55 of file slab.h.

Referenced by slab_cache_create().

4.74.1.5 #define SLAB_POISON (1<<1)

Definition at line 53 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_create(), slab_cache_free(), and slab_cache_grow().

4.74.1.6 #define SLAB_POISON_ALIVE_VALUE 0x0BADCAFE

Definition at line 40 of file slab.h.

Referenced by slab_cache_alloc().

4.74.1.7 #define SLAB_POISON_DEAD_VALUE 0xDEADBEEF

Definition at line 42 of file slab.h.

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

4.74.1.8 #define SLAB_RED_ZONE (1<<0)

Definition at line 51 of file slab.h.

Referenced by slab_cache_alloc(), slab_cache_create(), slab_cache_free(), and slab_cache_grow().

4.74.1.9 #define SLAB_RED_ZONE_VALUE 0x5711600D

Definition at line 44 of file slab.h.

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

4.74.1.10 #define SLAB_SIZE PAGE_SIZE

Definition at line 38 of file slab.h.

Referenced by slab_cache_create(), slab_cache_free(), and slab_cache_grow().

4.74.2 Typedef Documentation

4.74.2.1 typedef struct slab_bufctl_t slab_bufctl_t

Definition at line 90 of file slab.h.

4.74.2.2 typedef struct slab_cache_t slab_cache_t

Definition at line 84 of file slab.h.

4.74.2.3 typedef void(∗ slab_ctor_t)(void ∗, size_t)

Definition at line 60 of file slab.h.

4.74.2.4 typedef struct slab_t slab_t

Definition at line 103 of file slab.h.

4.74.3 Function Documentation

4.74.3.1 void∗ slab_cache_alloc (slab_cache_t ∗ *cache*)

ASSERTION: now that **slab_cache_grow()** (p. 246) 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 228 of file slab.c.

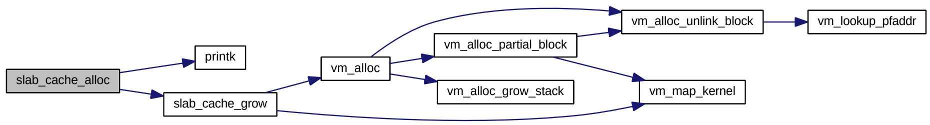
References assert, slab_cache_t::bufctl_offset, slab_cache_t::ctor, slab_cache_t::empty_count, slab_cache_t::flags, slab_t::free_list, slab_cache_t::name, slab_bufctl_t::next, slab_t::next, slab_t::obj_count, slab_cache_t::obj_size, slab_t::prev, printk(), slab_cache_grow(), SLAB_POISON, SLAB_POISON_ALIVE_VALUE, 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(), and slab_cache_create().

```
228                                                {
229     slab_t          *slab;
230     slab_bufctl_t   *bufctl;
231     uint32_t        *buffer;
232     unsigned int     idx;
233     unsigned int     dump_lines;
234 
235     if(cache->slabs_partial != NULL) {
236         slab = cache->slabs_partial;
237     }
238     else {
239         if(cache->slabs_empty == NULL) {
240             slab_cache_grow(cache);
241         }
242 
243         slab = cache->slabs_empty;
244 
246         assert(slab != NULL);
247 
257         /* We are about to allocate one object from this slab, so it will
258          *  not be empty anymore...*/
259         cache->slabs_empty      = slab->next;
260 
261         --(cache->empty_count);
262 
263         slab->next              = cache->slabs_partial;
264         if(slab->next != NULL) {
265             slab->next->prev = slab;
266         }
267         cache->slabs_partial    = slab;
268     }
269 
270     bufctl = slab->free_list;
271 
```

```
    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;
        }
    }

    buffer = (uint32_t *)( (char *)bufctl - cache->bufctl_offset );

    if(cache->flags & SLAB_POISON) {
        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;
        }

        if(cache->ctor != NULL) {
            cache->ctor((void *)buffer, cache->obj_size);
        }
    }
    else if(cache->flags & SLAB_RED_ZONE) {
        buffer[cache->obj_size / sizeof(uint32_t)] = SLAB_RED_ZONE_VALUE;
    }

    return (void *)buffer;
}
```

Here is the call graph for this function:



4.74.3.2 slab_cache_t∗ slab_cache_create (char ∗ *name,* size_t *size,* size_t *alignment,* slab_ctor_t *ctor,* slab_ctor_t *dtor,* int *flags*)

ASSERTION: ensure buffer size is at least the size of a pointer

ASSERTION: name is not NULL string

Definition at line 89 of file slab.c.

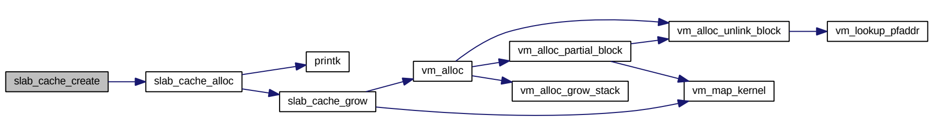
References slab_cache_t::alignment, slab_cache_t::alloc_size, assert, slab_cache_t::bufctl_offset, cpu_info, slab-_cache_t::ctor, cpu_info_t::dcache_alignment, slab_cache_t::dtor, slab_cache_t::empty_count, slab_cache_t::flags, slab_cache_t::max_colour, slab_cache_t::name, slab_cache_t::next, slab_cache_t::next_colour, NULL, slab_cache_t-::obj_size, slab_cache_t::prev, slab_cache_alloc(), slab_cache_list, 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().

```
                             {

    slab_cache_t    *cache;
    size_t           avail_space;
    size_t           wasted_space;
    unsigned int     buffers_per_slab;

    assert( size >= sizeof(void *) );

    assert(name != NULL);

    cache = slab_cache_alloc(&slab_cache_cache);

    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;

    /* add new cache to cache list */
    cache->next              = slab_cache_list;
    slab_cache_list          = cache;

    if(cache->next != NULL) {
        cache->next->prev = cache;
    }

    /* 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;
```

```
    }

    if(cache->alignment % sizeof(uint32_t) != 0) {
        cache->alignment += sizeof(uint32_t) - cache->alignment % sizeof(uint32_t);
    }

    /* reserve space for bufctl and/or redzone word */
    cache->obj_size = size;

    if(cache->obj_size % sizeof(uint32_t) != 0) {
        cache->obj_size += sizeof(uint32_t) - cache->obj_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 and/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;
    }

    avail_space = SLAB_SIZE - sizeof(slab_t);

    buffers_per_slab = avail_space / cache->alloc_size;

    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);

    return cache;
}
```

Here is the call graph for this function:



4.74.3.3 void slab_cache_destroy (slab_cache_t * *cache*)

ASSERTION: all memory has been returned to the cache

ASSERTION: empty slabs count is accurate

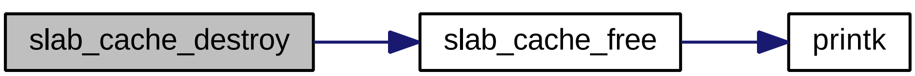
Definition at line 188 of file slab.c.

References assert, slab_cache_t::empty_count, slab_cache_t::next, slab_t::next, slab_cache_t::prev, slab_cache_free(), slab_cache_t::slabs_empty, slab_cache_t::slabs_full, and slab_cache_t::slabs_partial.

```
188 {
```

```
    slab_t       *slab;
    slab_t       *next;
    unsigned int  empty_count;

    assert(cache->slabs_full == NULL && cache->slabs_partial == NULL);

    /* remove from cache list */
    if(slab_cache_list == cache) {
        slab_cache_list = cache->next;
    }
    else {
        cache->prev->next = cache->next;
    }

    if(cache->next != NULL) {
        cache->next->prev = cache->prev;
    }

    /* release all slabs */
    slab        = cache->slabs_empty;
    empty_count = 0;

    while(slab != NULL) {
        next = slab->next;

        destroy_slab(cache, slab);

        slab = next;
        ++empty_count;
    }

    assert(cache->empty_count == empty_count);

    /* free cache structure */
    slab_cache_free(cache);
}
```

Here is the call graph for this function:



4.74.3.4 void slab_cache_free (void * *buffer*)

Definition at line 350 of file slab.c.

References ALIGN_START, 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, 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.

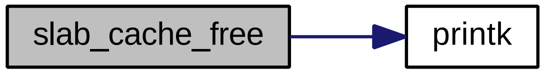
Referenced by slab_cache_destroy().

```
350                                  {
351     addr_t            slab_start;
352     slab_t           *slab;
353     slab_cache_t     *cache;
354     slab_bufctl_t    *bufctl;
355     uint32_t         *rz_word;
356     uint32_t         *buffer32;
357     unsigned int      idx;
358 
359     /* compute address of slab data structure */
360     slab_start = ALIGN_START(buffer, SLAB_SIZE);
361     slab = (slab_t *)(slab_start + SLAB_SIZE - sizeof(slab_t) );
362 
363     /* obtain address of cache and bufctl */
```

```
    cache  = slab->cache;
    bufctl = (slab_bufctl_t *)((char *)buffer + cache->bufctl_offset);

    /* If slab is on the full slabs list, move it to the partial list
     * since we are about to return a buffer to it. */
    if(slab->free_list == NULL) {
        /* 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) {
        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) {
        if(cache->dtor != NULL) {
            cache->dtor(buffer, cache->obj_size);
        }

        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;
        }

```

```
445         ++(cache->empty_count);
446     }
447 }
```

Here is the call graph for this function:



4.74.3.5 void slab_cache_grow (slab_cache_t * *cache*)

ASSERTION: slab address is not NULL

TODO: check this condition

Definition at line 449 of file slab.c.

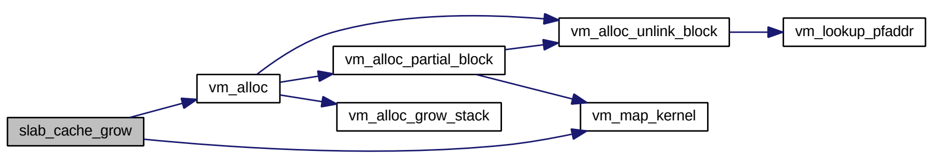
References slab_cache_t::alignment, slab_cache_t::alloc_size, assert, slab_cache_t::bufctl_offset, slab_t::cache, slab_t::colour, slab_cache_t::ctor, slab_cache_t::empty_count, slab_cache_t::flags, slab_t::free_list, global_page_allocator, slab_cache_t::max_colour, slab_bufctl_t::next, slab_t::next, slab_cache_t::next_colour, NULL, slab_t::obj_count, slab_cache_t::obj_size, pfalloc, slab_t::prev, SLAB_POISON, SLAB_POISON_DEAD_VALUE, SLAB_RED_ZONE, SLAB_RED_ZONE_VALUE, SLAB_SIZE, slab_cache_t::slabs_empty, vm_alloc(), VM_FLAG_READ_WRITE, and vm_map_kernel().

Referenced by slab_cache_alloc().

```
449                                       {
450     void             *slab_addr;
451     slab_t           *slab;
452     slab_bufctl_t    *bufctl;
453     slab_bufctl_t    *next;
454     addr_t            buffer;
455     uint32_t         *buffer_end;
456     uint32_t         *ptr;
457 
458     /* allocate new slab */
459     slab_addr = vm_alloc( global_page_allocator );
460 
462     assert(slab_addr != NULL);
463 
464     vm_map_kernel(slab_addr, pfalloc(), VM_FLAG_READ_WRITE);
465 
466     slab = (slab_t *)( (char *)slab_addr + SLAB_SIZE - sizeof(slab_t) );
467 
468     slab->cache = cache;
469 
470     /* slab is initially empty */
471     slab->obj_count = 0;
472 
473     slab->next         = cache->slabs_empty;
474     cache->slabs_empty = slab;
475 
476     if(slab->next != NULL) {
477         slab->next->prev = slab;
478     }
479 
480     ++(cache->empty_count);
481 
482     /* set slab colour and update cache next colour */
483     slab->colour = cache->next_colour;
484 
485     if(cache->next_colour < cache->max_colour) {
486         cache->next_colour += cache->alignment;
487     }
488     else {
489         cache->next_colour = 0;
490     }
```

```

    /* compute address of first bufctl */
    bufctl          = (slab_bufctl_t *)( (char *)slab_addr + slab->colour + cache->bufctl_offset );
    slab->free_list = bufctl;

    while(1) {
        buffer = (addr_t)bufctl - cache->bufctl_offset;

        if(cache->flags & SLAB_POISON) {
            buffer_end = (uint32_t *)(buffer + cache->obj_size);

            for(ptr = (uint32_t *)buffer; ptr < buffer_end; ++ptr) {
                *ptr = SLAB_POISON_DEAD_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) {
                *ptr = SLAB_RED_ZONE_VALUE;
            }
        }
        else if (cache->ctor != NULL) {
            cache->ctor((void *)buffer, cache->obj_size);
        }

        next = (slab_bufctl_t *)( (char *)bufctl + cache->alloc_size );

        if(next >= (slab_bufctl_t *)slab) {
            bufctl->next = NULL;
            break;
        }

        bufctl->next = next;
        bufctl = next;
    }
}
```

Here is the call graph for this function:



4.74.3.6 void slab_cache_reap (slab_cache_t * *cache*)

Definition at line 529 of file slab.c.

References slab_cache_t::empty_count, slab_t::next, slab_cache_t::slabs_empty, and slab_cache_t::working_set.

```
529                                           {
530     slab_t          *slab;
531
532     while(cache->empty_count > cache->working_set) {
533         /* select the first empty slab */
534         slab = cache->slabs_empty;
535
536         /* unlink it and update count */
537         cache->slabs_empty  = slab->next;
538         cache->empty_count -= 1;
539
540         /* destroy slab */
541         destroy_slab(cache, slab);
542     }
543 }
```

4.74.3.7 void slab_cache_set_working_set (**slab_cache_t** ∗ *cache,* unsigned int *n*)

Definition at line 545 of file slab.c.

References slab_cache_t::working_set.

```
545                                                             {
546     cache->working_set = n;
547 }
```

4.74.4 Variable Documentation

4.74.4.1 **slab_cache_t**∗ slab_cache_list

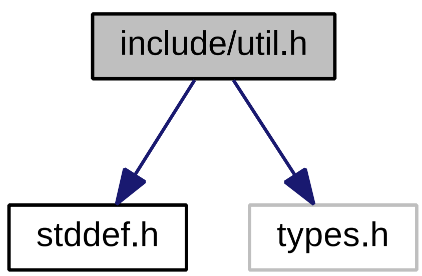
Definition at line 65 of file slab.c.

Referenced by slab_cache_create().

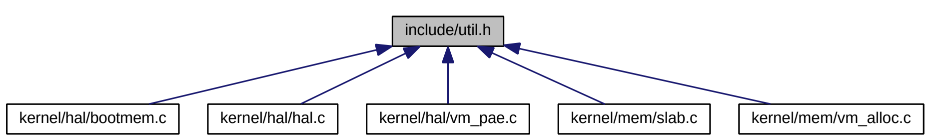
4.75 include/util.h File Reference

`#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 **OFFSET_OF**(x, s) ((**uint32_t**)(x) & ((s)-1))
- #define **ALIGN_START**(x, s) ((**addr_t**)((**uint32_t**)(x) & ∼((s)-1)))
- #define **ALIGN_END**(x, s) (**OFFSET_OF**(x, s) == 0 ? (**addr_t**)(x) : **ALIGN_START**(x, s) + 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)))

4.75.1 Macro Definition Documentation

4.75.1.1 #define ALIGN_END(x, s) (OFFSET_OF(x, s) == 0 ? (addr_t)(x) : ALIGN_START(x, s) + s)

Definition at line 43 of file util.h.

Referenced by hal_init(), vm_alloc_add_region(), and vm_alloc_init_allocator().

4.75.1.2 #define ALIGN_START(x, s) ((addr_t)((uint32_t)(x) & ∼((s)-1)))

Definition at line 41 of file util.h.

Referenced by slab_cache_free(), and vm_alloc_init_allocator().

4.75.1.3 #define alloc_backward(T, p) ((T ∗)alloc_forward_func(sizeof(T), &(p)))

Definition at line 63 of file util.h.

4.75.1.4 #define alloc_forward(T, p) ((T ∗)alloc_forward_func(sizeof(T), &(p)))

Definition at line 61 of file util.h.

4.75.1.5 #define OFFSET_OF(x, s) ((uint32_t)(x) & ((s)-1))

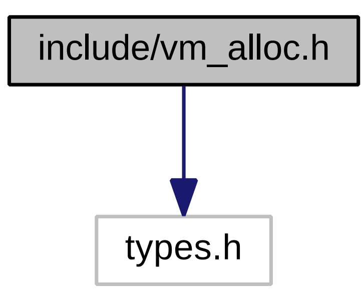
Definition at line 39 of file util.h.

Referenced by apply_mem_hole(), bootmem_init(), and vm_alloc_add_region().

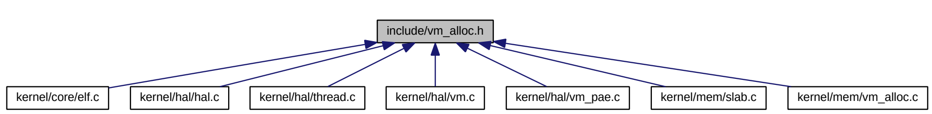
4.76 include/vm_alloc.h File Reference

`#include <types.h>`

Include dependency graph for vm_alloc.h:



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



Data Structures

- struct **vm_alloc_t**
- struct **vm_block_t**

Macros

- #define **VM_ALLOC_STACK_ENTRIES** 1024
- #define **VM_ALLOC_BLOCK_SIZE** (**VM_ALLOC_STACK_ENTRIES** ∗ **PAGE_SIZE**)
- #define **VM_ALLOC_BLOCK_MASK** (**VM_ALLOC_BLOCK_SIZE** - 1)
- #define **VM_ALLOC_WAS_FREE false**
- #define **VM_ALLOC_WAS_USED true**
- #define **VM_ALLOC_IS_FREE**(b) ((b)->next != **NULL** && (b)->stack_ptr == **NULL**)
- #define **VM_ALLOC_IS_PARTIAL**(b) ((b)->next != **NULL** && (b)->stack_ptr != **NULL**)
- #define **VM_ALLOC_IS_USED**(b) ((b)->next == **NULL**)
- #define **VM_ALLOC_EMPTY_STACK**(b) ((b)->stack_ptr >= (b)->stack_addr + **VM_ALLOC_STACK_ENTRIES**)
- #define **VM_ALLOC_FULL_STACK**(b) ((b)->stack_ptr <= (b)->stack_addr + 1)
- #define **VM_ALLOC_CANNOT_GROW**(b) ((b)->stack_next >= (b)->base_addr + **VM_ALLOC_BLOCK_SIZE**)

Typedefs

- typedef struct **vm_alloc_t vm_alloc_t**
- typedef struct **vm_block_t vm_block_t**

Functions

- **addr_t vm_alloc** (**vm_alloc_t** ∗allocator)

  Allocate a page of virtual address space.
- **addr_t vm_alloc_low_latency** (**vm_alloc_t** ∗allocator)

  Allocate a page of virtual address space for time critical code path.
- void **vm_free** (**vm_alloc_t** ∗allocator, **addr_t** page)

  Free a page of virtual address space.
- void **vm_alloc_init** (**vm_alloc_t** ∗allocator, **addr_t** start_addr, **addr_t** end_addr)
- void **vm_alloc_destroy** (**vm_alloc_t** ∗allocator)
- void **vm_alloc_init_allocator** (**vm_alloc_t** ∗allocator, **addr_t** start_addr, **addr_t** end_addr)

  Basic initialization of virtual memory allocator.
- void **vm_alloc_add_region** (**vm_alloc_t** ∗allocator, **addr_t** start_addr, **addr_t** end_addr)

  Add a contiguous region of available virtual memory to the allocator.
- void **vm_alloc_free_block** (**vm_block_t** ∗block)

  Insert block in the free list.
- void **vm_alloc_partial_block** (**vm_block_t** ∗block)

  Insert block in the partial blocks list.
- void **vm_alloc_custom_block** (**vm_block_t** ∗block, **addr_t** start_addr, **addr_t** end_addr)
- void **vm_alloc_unlink_block** (**vm_block_t** ∗block)

  Unlink memory block from free or partial block list.
- void **vm_alloc_grow_stack** (**vm_block_t** ∗block)

Initialize the stack of a partial block with all remaining pages which have not yet been allocated.

- **addr_t vm_alloc_grow_single** (**vm_block_t** *block)

  Obtain a free page from a partial block, but defer page stack initialization for the block.

Variables

- **vm_alloc_t** * **global_page_allocator**

  global page allocator (region 0..KLIMIT)

4.76.1 Macro Definition Documentation

4.76.1.1 #define VM_ALLOC_BLOCK_MASK (VM_ALLOC_BLOCK_SIZE - 1)

Definition at line 42 of file vm_alloc.h.

4.76.1.2 #define VM_ALLOC_BLOCK_SIZE (VM_ALLOC_STACK_ENTRIES * PAGE_SIZE)

Definition at line 40 of file vm_alloc.h.

Referenced by vm_alloc_add_region(), vm_alloc_custom_block(), vm_alloc_grow_stack(), vm_alloc_init_allocator(), vm_alloc_partial_block(), and vm_free().

4.76.1.3 #define VM_ALLOC_CANNOT_GROW(*b*) ((b)->stack_next >= (b)->base_addr + VM_ALLOC_BLOCK_SIZE)

Definition at line 61 of file vm_alloc.h.

Referenced by vm_alloc_grow_single().

4.76.1.4 #define VM_ALLOC_EMPTY_STACK(*b*) ((b)->stack_ptr >= (b)->stack_addr + VM_ALLOC_STACK_ENTRIES)

Definition at line 57 of file vm_alloc.h.

Referenced by vm_alloc(), and vm_alloc_low_latency().

4.76.1.5 #define VM_ALLOC_FULL_STACK(*b*) ((b)->stack_ptr <= (b)->stack_addr + 1)

Definition at line 59 of file vm_alloc.h.

Referenced by vm_alloc_custom_block(), vm_alloc_grow_stack(), and vm_free().

4.76.1.6 #define VM_ALLOC_IS_FREE(*b*) ((b)->next != NULL && (b)->stack_ptr == NULL)

Definition at line 50 of file vm_alloc.h.

Referenced by vm_alloc_custom_block().

4.76.1.7 #define VM_ALLOC_IS_PARTIAL(*b*) ((b)->next != NULL && (b)->stack_ptr != NULL)

Definition at line 52 of file vm_alloc.h.

Referenced by vm_alloc_custom_block(), and vm_free().

4.76.1.8 #define VM_ALLOC_IS_USED(*b*) ((b)->next == NULL)

Definition at line 54 of file vm_alloc.h.

Referenced by vm_alloc_custom_block(), and vm_free().

4.76.1.9 #define VM_ALLOC_STACK_ENTRIES 1024

Definition at line 38 of file vm_alloc.h.

4.76.1.10 #define VM_ALLOC_WAS_FREE false

Definition at line 45 of file vm_alloc.h.

4.76.1.11 #define VM_ALLOC_WAS_USED true

Definition at line 47 of file vm_alloc.h.

4.76.2 Typedef Documentation

4.76.2.1 typedef struct vm_alloc_t vm_alloc_t

Definition at line 114 of file vm_alloc.h.

4.76.2.2 typedef struct vm_block_t vm_block_t

Definition at line 116 of file vm_alloc.h.

4.76.3 Function Documentation

4.76.3.1 addr_t vm_alloc (vm_alloc_t * *allocator*)

Allocate a page of virtual address space.

Parameters

allocator	allocator which manages the memory region from which we wish to obtain a page

ASSERTION: allocator is not null

ASSERTION: since block is expected to be partial, its stack pointer should not be null

ASSERTION: at this point, the page stack should not be empty (stack underflow check)

Definition at line 87 of file vm_alloc.c.

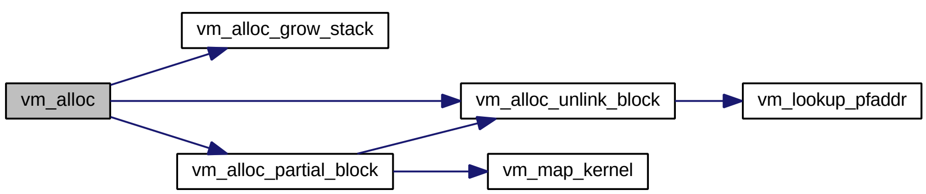
References assert, vm_alloc_t::free_list, NULL, vm_alloc_t::partial_list, vm_block_t::stack_ptr, VM_ALLOC_EMPTY_-STACK, vm_alloc_grow_stack(), vm_alloc_partial_block(), and vm_alloc_unlink_block().

Referenced by elf_load(), elf_setup_stack(), slab_cache_grow(), thread_page_create(), vm_clone_page_directory(), and vm_destroy_page_directory().

```
87                                    {
88     vm_block_t *block;
```

```
    addr_t      page;

    assert(allocator != NULL);

    block = allocator->partial_list;

    if(block == NULL) {
        block = allocator->free_list;

        if(block == NULL) {
             return (addr_t)NULL;
         }

         vm_alloc_partial_block(block);
     }

     assert(block->stack_ptr != NULL);

     /* if the page stack is empty, perform deferred page stack initialization */
     if( VM_ALLOC_EMPTY_STACK(block) ) {
         vm_alloc_grow_stack(block);
     }

     assert( ! VM_ALLOC_EMPTY_STACK(block) );

     page = *(block->stack_ptr++);

     /* if we just emptied the stack, mark the block as used */
     if( VM_ALLOC_EMPTY_STACK(block) ) {
         vm_alloc_unlink_block(block);
     }

     return page;
}
```

Here is the call graph for this function:



4.76.3.2 void vm_alloc_add_region (vm_alloc_t * *allocator,* addr_t *start_addr,* addr_t *end_addr*)

Add a contiguous region of available virtual memory to the allocator.

Parameters

allocator	**vm_alloc_t** (p. 52) structure for a virtual memory allocator
start_addr	start address of the region
end_addr	end address of the region (first unavailable page)

Definition at line 344 of file vm_alloc.c.

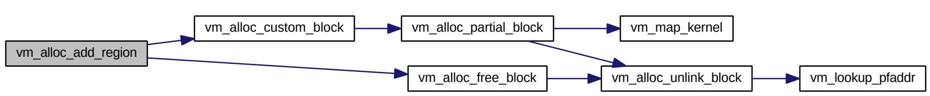
References ALIGN_END, vm_alloc_t::base_addr, vm_block_t::base_addr, vm_alloc_t::block_array, OFFSET_OF, vm_-alloc_t::start_addr, VM_ALLOC_BLOCK_SIZE, vm_alloc_custom_block(), and vm_alloc_free_block().

Referenced by vm_alloc_init(), and vm_boot_init().

```
344                                                                                   {
345     addr_t       start_addr_adjusted;
346     unsigned int start;
347     unsigned int end;
348     unsigned int end_full;
```

```
    unsigned int idx;
    addr_t       limit;

    /* skip the block array */
    if(start_addr >= allocator->start_addr) {
        start_addr_adjusted = start_addr;
    }
    else {
        start_addr_adjusted = allocator->start_addr;
    }

    /* start and end block indices */
    start = ((unsigned int)start_addr_adjusted - (unsigned int)allocator->
  base_addr) / VM_ALLOC_BLOCK_SIZE;
    end   = ((unsigned int)end_addr            - (unsigned int)allocator->
  base_addr) / VM_ALLOC_BLOCK_SIZE;

    /* check and remember whether last block is partial (last_full < end) or
     * completely free (last_full == end) */
    if( OFFSET_OF(end_addr, VM_ALLOC_BLOCK_SIZE) == 0) {
        end_full = end;
    }
    else {
        end_full = end + 1;
    }

    /* array initialization -- first block (if partial) */
    idx = start;

    if( OFFSET_OF(start_addr_adjusted, VM_ALLOC_BLOCK_SIZE) != 0 ) {
        limit = ALIGN_END(start_addr_adjusted, VM_ALLOC_BLOCK_SIZE);

        if(end_addr < limit) {
            limit = end_addr;
        }

        vm_alloc_custom_block(&allocator->block_array[idx], start_addr_adjusted, limit);

        ++idx;
    }

    /* array initialization -- free blocks */
    for(; idx < end; ++idx) {
        vm_alloc_free_block(&allocator->block_array[idx]);
    }

    /* array initialization -- last block (if partial) */
    if(idx < end_full) {
        vm_alloc_custom_block(&allocator->block_array[idx], allocator->
  block_array[idx].base_addr, end_addr);
    }
}
```

Here is the call graph for this function:



4.76.3.3 void vm_alloc_custom_block (vm_block_t ∗ *block*, addr_t *start_addr*, addr_t *end_addr*)

ASSERTION: block is not null

ASSERTION: start and end addresses must be page aligned

ASSERTION: start and end addr are inside block, address range is non-empty

ASSERTION: block is not free

ASSERTION: block is partial at this point

ASSERTION: page stack overflow check

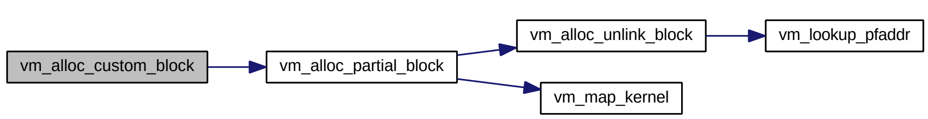
Definition at line 561 of file vm_alloc.c.

References assert, vm_block_t::base_addr, NULL, page_offset_of, PAGE_SIZE, vm_block_t::stack_addr, vm_block_t-::stack_ptr, VM_ALLOC_BLOCK_SIZE, VM_ALLOC_FULL_STACK, VM_ALLOC_IS_FREE, VM_ALLOC_IS_PARTIA-L, VM_ALLOC_IS_USED, and vm_alloc_partial_block().

Referenced by vm_alloc_add_region().

```
561                                                                                   {
562 #ifndef NDEBUG
563     addr_t      limit;
564 #endif
565     addr_t      page;
566     addr_t      adjusted_start;
567 
569     assert(block != NULL);
570 
572     assert(page_offset_of(start_addr) == 0 && page_offset_of(end_addr) == 0);
573 
574 #ifndef NDEBUG
575     limit = block->base_addr + VM_ALLOC_BLOCK_SIZE;
576 #endif
577 
579     assert(start_addr >= block->base_addr && end_addr <= limit && start_addr < end_addr );
580 
582     assert( ! VM_ALLOC_IS_FREE(block) );
583 
584     adjusted_start = start_addr;
585 
586     if( VM_ALLOC_IS_USED(block) ) {
587         /* if no stack address is specified at this point, use the first page
588          * of the address range for this purpose */
589         if( block->stack_addr == NULL ) {
590             block->stack_addr = (addr_t *)start_addr;
591             adjusted_start    = start_addr + PAGE_SIZE;
592 
593             /* if the address range contained only a single page, there is
594              * nothing left to do here */
595             if(adjusted_start >= end_addr) {
596                 return;
597             }
598         }
599 
600         vm_alloc_partial_block(block);
601     }
602 
604     assert( VM_ALLOC_IS_PARTIAL(block) );
605 
606     /* initialize stack */
607     page = adjusted_start;
608     while(page < end_addr) {
610         assert( ! VM_ALLOC_FULL_STACK(block) );
611 
612         *(--block->stack_ptr) = page;
613         page += PAGE_SIZE;
614     }
615 }
```

Here is the call graph for this function:



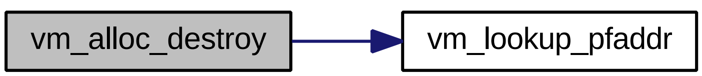
4.76.3.4 void vm_alloc_destroy (vm_alloc_t * *allocator*)

Definition at line 218 of file vm_alloc.c.

References vm_alloc_t::block_array, vm_block_t::next, NULL, PAGE_SIZE, vm_alloc_t::partial_list, pffree, vm_block_t::stack_addr, and vm_lookup_pfaddr().

```
218                                          {
219     vm_block_t   *head;
220     vm_block_t   *block;
221     pfaddr_t      paddr;
222     addr_t        addr;
223     unsigned int  idx;
224
225     /* de-allocate page stacks */
226     head  = allocator->partial_list;
227     block = head;
228
229     if(block != NULL) {
230         do {
231             paddr = vm_lookup_pfaddr(NULL, (addr_t)block->stack_addr);
232             pffree(paddr);
233
234             block = block->next;
235         } while(block != head);
236     }
237
238     /* de-allocate block array pages */
239     addr = (addr_t)allocator->block_array;
240     for(idx = 0; idx < allocator->array_pages; ++idx) {
241         paddr = vm_lookup_pfaddr(NULL, addr);
242         pffree(paddr);
243
244         addr += PAGE_SIZE;
245     }
246 }
```

Here is the call graph for this function:



4.76.3.5 void vm_alloc_free_block (vm_block_t * *block*)

Insert block in the free list.

This is typically done when the block was a partial one, and the last page has just been returned to it.

Parameters

block	block to insert in the free list

ASSERTION: block is not null

ASSERTION: block->allocator should not be NULL

Definition at line 407 of file vm_alloc.c.

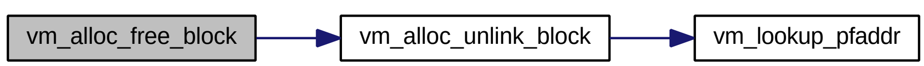
References vm_block_t::allocator, assert, vm_alloc_t::free_list, vm_block_t::next, NULL, vm_block_t::prev, vm_block_t::stack_ptr, and vm_alloc_unlink_block().

Referenced by vm_alloc_add_region(), and vm_free().

```
407                                         {
408     vm_block_t      *prev;
409     vm_block_t      *next;
410
412     assert(block != NULL);
413
414     /* unlink from partial list if necessary */
415     vm_alloc_unlink_block(block);
```

```
    assert(block->allocator != NULL);

    /* link block to the free list */
    if(block->allocator->free_list == NULL) {
        /* special case: free list is empty */
        block->allocator->free_list = block;

        block->next = block;
        block->prev = block;
    }
    else {
        /* block will be at the end of the free list */
        next = block->allocator->free_list;
        prev = next->prev;

        /* re-link block */
        block->prev = prev;
        block->next = next;

        prev->next = block;
        next->prev = block;
    }

    /* set the stack pointer to null to indicate this is a free block */
    block->stack_ptr = NULL;
}
```

Here is the call graph for this function:



4.76.3.6 addr_t vm_alloc_grow_single (vm_block_t ∗ *block*)

Obtain a free page from a partial block, but defer page stack initialization for the block.

This function must only be called on a partial block, and only after checking first that the page stack is empty. This function takes care of unlinking the block from the partial list if the last page is allocated.

Parameters

block	block from which to allocate the page

ASSERTION: block is not null

ASSERTION: block is linked (it should be in the partial list)

ASSERTION: block actually has a stack

ASSERTION: region can still grow

Definition at line 734 of file vm_alloc.c.

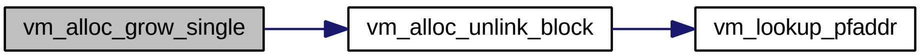
References assert, vm_block_t::next, NULL, PAGE_SIZE, vm_block_t::prev, vm_block_t::stack_next, vm_block_t::stack_ptr, VM_ALLOC_CANNOT_GROW, and vm_alloc_unlink_block().

Referenced by vm_alloc_low_latency().

```
734                                            {
735     addr_t page;
736
738     assert(block != NULL);
739
741     assert(block->next != NULL && block->prev != NULL);
742
744     assert(block->stack_ptr != NULL);
745
```

```
    assert( ! VM_ALLOC_CANNOT_GROW(block) );

    page              = block->stack_next;
    block->stack_next = page + PAGE_SIZE;

    if( VM_ALLOC_CANNOT_GROW(block) ) {
        /* block is now used up, remove it from the partial list */
        vm_alloc_unlink_block(block);
    }

    return page;
}
```

Here is the call graph for this function:



4.76.3.7 void vm_alloc_grow_stack (vm_block_t * *block*)

Initialize the stack of a partial block with all remaining pages which have not yet been allocated.

Parameters

block	block which will have its stack initialized

ASSERTION: block is not null

ASSERTION: block is linked (it should be in the partial list)

ASSERTION: block actually has a stack

ASSERTION: stack underflow check

Definition at line 695 of file vm_alloc.c.

References assert, vm_block_t::base_addr, vm_block_t::next, NULL, PAGE_SIZE, vm_block_t::prev, vm_block_t::stack_next, vm_block_t::stack_ptr, VM_ALLOC_BLOCK_SIZE, and VM_ALLOC_FULL_STACK.

Referenced by vm_alloc().

```
695                                             {
696     addr_t  limit;
697     addr_t  page;
698     addr_t *stack_ptr;
699
701     assert(block != NULL);
702
704     assert(block->next != NULL && block->prev != NULL);
705
707     assert(block->stack_ptr != NULL);
708
709     stack_ptr = block->stack_ptr;
710     page      = block->stack_next;
711     limit     = block->base_addr + VM_ALLOC_BLOCK_SIZE;
712
713     while(page < limit) {
715         assert( ! VM_ALLOC_FULL_STACK(block) );
716
717         *(--stack_ptr) = page;
718
719         page += PAGE_SIZE;
720     }
721
722     block->stack_ptr  = stack_ptr;
723     block->stack_next = limit;
724 }
```

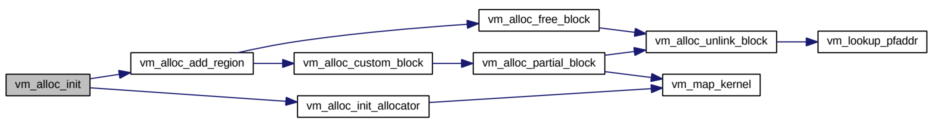
4.76.3.8 void vm_alloc_init (**vm_alloc_t** * *allocator,* **addr_t** *start_addr,* **addr_t** *end_addr*)

Definition at line 213 of file vm_alloc.c.

References vm_alloc_add_region(), and vm_alloc_init_allocator().

```
213                                                                    {
214     vm_alloc_init_allocator( allocator, start_addr, end_addr);
215     vm_alloc_add_region(     allocator, start_addr, end_addr);
216 }
```

Here is the call graph for this function:



4.76.3.9 void vm_alloc_init_allocator (**vm_alloc_t** * *allocator,* **addr_t** *start_addr,* **addr_t** *end_addr*)

Basic initialization of virtual memory allocator.

Parameters

allocator	**vm_alloc_t** (p. 52) structure for a virtual memory allocator
start_addr	start address of the region managed by the allocator
size	size of the region managed by the allocator

ASSERTION: allocator structure pointer must not be null

ASSERTION: start and end addresses must be multiples of page size (page-aligned memory region)

ASSERTION: once all the array pages are allocated, we should have reached the allocatable pages region

Definition at line 254 of file vm_alloc.c.

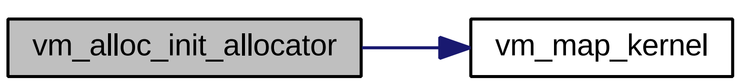
References ALIGN_END, ALIGN_START, vm_block_t::allocator, vm_alloc_t::array_pages, assert, vm_alloc_t::base-_addr, vm_block_t::base_addr, vm_alloc_t::block_array, vm_alloc_t::block_count, vm_alloc_t::end_addr, vm_alloc_t-::free_list, vm_block_t::next, NULL, page_offset_of, PAGE_SIZE, vm_alloc_t::partial_list, pfalloc, vm_block_t::stack_-addr, vm_alloc_t::start_addr, VM_ALLOC_BLOCK_SIZE, VM_FLAG_READ_WRITE, and vm_map_kernel().

Referenced by vm_alloc_init(), and vm_boot_init().

```
254                                                                                   {
255     addr_t        base_addr;        /* block-aligned start address */
256     addr_t        aligned_end;        /* block-aligned end address */
257     addr_t        adjusted_start;    /* actual start of available memory, block array skipped */
258 
259     vm_block_t   *block_array;        /* start of array */
260     unsigned int  block_count;        /* array size, in blocks (entries) */
261     size_t        array_size;        /* array size, in bytes */
262     unsigned int  array_page_count;     /* array size, in pages */
263 
264     addr_t        addr;                /* some virtual address */
265     pfaddr_t      paddr;            /* some page frame address */
266     unsigned int  idx;                /* an array index */
267 
268 
270     assert(allocator != NULL);
271 
273     assert( page_offset_of(start_addr) == 0 && page_offset_of(end_addr) == 0 );
274 
275 
276     /* align base and end addresses to block size */
```

```
    base_addr   = (addr_t)ALIGN_START(start_addr, VM_ALLOC_BLOCK_SIZE);
    aligned_end = (addr_t)ALIGN_END(end_addr, VM_ALLOC_BLOCK_SIZE);

    /* calculate number of memory blocks managed by this allocator */
    block_count = ( (char *)aligned_end - (char *)base_addr ) /
  VM_ALLOC_BLOCK_SIZE;

    /* calculate the number of pages required to store the memory block
     * descriptor array */
    array_size = block_count * sizeof(vm_block_t);
    array_page_count = array_size / PAGE_SIZE;
    if(array_size % PAGE_SIZE != 0) {
        ++array_page_count;
    }

    /* address of the block array */
    block_array = (vm_block_t *)start_addr;

    /* adjust base address to skip block descriptor array */
    adjusted_start = start_addr + array_page_count * PAGE_SIZE;

    /* initialize allocator struct */
    allocator->start_addr   = adjusted_start;
    allocator->end_addr     = end_addr;
    allocator->base_addr    = base_addr;
    allocator->block_count  = block_count;
    allocator->block_array  = block_array;
    allocator->array_pages  = array_page_count;
    allocator->free_list    = NULL;
    allocator->partial_list = NULL;

    /* allocate block descriptor array pages */
    addr = (addr_t)block_array;
    for(idx = 0; idx < array_page_count; ++idx) {
        /* allocate and map page */
        paddr = pfalloc();
        vm_map_kernel(addr, paddr, VM_FLAG_READ_WRITE);

        /* calculate address of next page */
        addr += PAGE_SIZE;
    }

    assert(addr == adjusted_start);

    /* basic initialization of array (all blocks unlinked/used) */
    addr = base_addr;
    for(idx = 0; idx < block_count; ++idx) {
        block_array[idx].base_addr  = addr;
        block_array[idx].allocator  = allocator;

        /* mark block as unlinked for now */
        block_array[idx].next       = NULL;

        /* a null stack base indicates the block is uninitialized */
        block_array[idx].stack_addr = NULL;

        /* calculate address of next block */
        addr += VM_ALLOC_BLOCK_SIZE;
    }
}
```

Here is the call graph for this function:



4.76.3.10 addr_t vm_alloc_low_latency (vm_alloc_t * *allocator*)

Allocate a page of virtual address space for time critical code path.

Same as **vm_alloc()** (p. 252), but some time consuming housekeeping steps are deferred.

Parameters

allocator	allocator which manages the memory region from which we wish to obtain a page

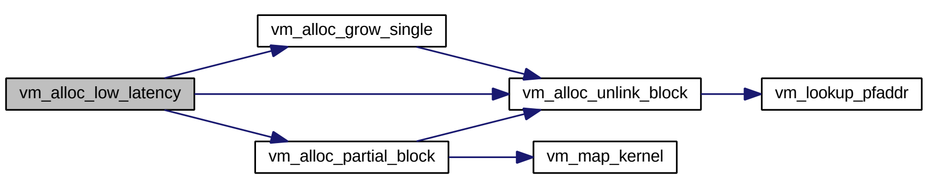
ASSERTION: allocator is not null

Definition at line 133 of file vm_alloc.c.

References assert, vm_alloc_t::free_list, NULL, vm_alloc_t::partial_list, vm_block_t::stack_ptr, VM_ALLOC_EMPTY_-STACK, vm_alloc_grow_single(), vm_alloc_partial_block(), and vm_alloc_unlink_block().

```
133                                                {
134     vm_block_t *block;
135     addr_t      page;
136 
138     assert(allocator != NULL);
139 
140     block = allocator->partial_list;
141 
142     if(block == NULL) {
143         block = allocator->free_list;
144 
145         if(block == NULL) {
146             return (addr_t)NULL;
147         }
148 
149         vm_alloc_partial_block(block);
150     }
151 
152     /* if the page stack is empty, allocate sequentially from the start of the
153      * block and continue to defer page stack initialization */
154     if( VM_ALLOC_EMPTY_STACK(block) ) {
155         return vm_alloc_grow_single(block);
156     }
157 
158     page = *(block->stack_ptr++);
159 
160     if( VM_ALLOC_EMPTY_STACK(block) ) {
161         /* block is now used up, remove it from the partial blocks list */
162         vm_alloc_unlink_block(block);
163     }
164 
165     return page;
166 }
```

Here is the call graph for this function:



4.76.3.11 void vm_alloc_partial_block (vm_block_t * *block*)

Insert block in the partial blocks list.

This is typically done when the block is a free one from which we intend to allocate pages, or when the block is used (unlinked) and we intend to return pages to it. The stack is initialized empty, but the deferred stack initialization mechanism is enabled if the block is free on function entry.

Parameters

block	block to insert in the partial list

ASSERTION: block is not null

ASSERTION: block stack address is not null

ASSERTION: block->allocator should not be NULL

Definition at line 454 of file vm_alloc.c.

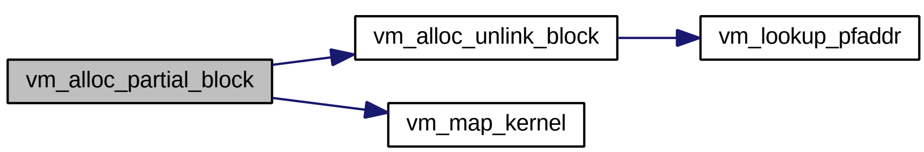
References vm_block_t::allocator, assert, vm_block_t::base_addr, vm_block_t::next, NULL, PAGE_SIZE, vm_alloc_t::partial_list, pfalloc, vm_block_t::prev, vm_block_t::stack_addr, vm_block_t::stack_next, vm_block_t::stack_ptr, VM_ALLOC_BLOCK_SIZE, vm_alloc_unlink_block(), VM_FLAG_READ_WRITE, and vm_map_kernel().

Referenced by vm_alloc(), vm_alloc_custom_block(), vm_alloc_low_latency(), and vm_free().

```
454                                               {
455     vm_block_t      *prev;
456     vm_block_t      *next;
457     addr_t          *stack_addr;
458     pfaddr_t         paddr;
459     bool             was_free;
460
461
463     assert(block != NULL);
464
465     /* To keep in mind...
466      *
467      * When the allocator is initialized, some blocks may be created partial
468      * (typical for the first and the last block of the region). If there is a
469      * hole at the start of the block, the page stack will be at the first
470      * available page, not at the start of the block. Since these blocks have
471      * holes, they will never be in the free state.
472      *
473      * So, when a block is free on function entry, we ensure the stack is placed
474      * at the start of the block so that all the remaining pages can be
475      * allocated sequentially (see deferred stack intialization below). However,
476      * if the block is in the used state on function entry, we leave the stack
477      * at its previous location since the first page of the block might not be
478      * available. */
479
480     if(block->next == NULL) {
482         assert(block->stack_addr != NULL);
483
484         /* block was used on function entry */
485         was_free = false;
486     }
487     else {
488         if(block->stack_ptr != NULL) {
489             /* block is already partial, leave it untouched */
490             return;
491         }
492
493         /* block was free on function entry */
494         was_free = true;
495
496         /* unlink from free list */
497         vm_alloc_unlink_block(block);
498
499         /* use first page of block for the stack */
500         block->stack_addr = (addr_t *)block->base_addr;
501     }
502
503     /* allocate the page stack */
504     stack_addr  = block->stack_addr;
505     paddr       = pfalloc();
506     vm_map_kernel((addr_t)stack_addr, paddr, VM_FLAG_READ_WRITE);
507
509     assert(block->allocator != NULL);
510
511     /* link block to the partial list */
512     if(block->allocator->partial_list == NULL) {
513         /* special case: partial list is empty */
514         block->allocator->partial_list = block;
515
516         block->next = block;
```

```
        block->prev = block;
    }
    else {
        /* block will be at to the end of the partial block list */
        next = block->allocator->partial_list;
        prev = next->prev;

        /* re-link block */
        block->prev = prev;
        block->next = next;

        prev->next = block;
        next->prev = block;
    }

    /* Ok, here's the deal (deferred stack intialization)...
     *
     * We do not want to initialize the page stack right now because this is
     * a time consuming operation, and we might be in time-critical code
     * (interrupt handling code for example). Instead, the stack initialization
     * is deferred until the next page allocations. The first non-time critical
     * allocation which encounters an empty stack will initialize the whole
     * stack. In the meantime, time critical ones will just allocate pages
     * sequentially from the start of the block.
     *
     * The stack_next pointer in the vm_block_t structure points to the next
     * page available for sequential allocation. The memory block is actually
     * used up (no more pages available) when the page stack is empty AND the
     * stack_next pointer has reached the end of the block. */

    /* initialize the stack as empty */
    block->stack_ptr = (addr_t *)( (char *)stack_addr + PAGE_SIZE );

    if(was_free) {
        /* free block: we skip the first page as it was allocated for the
         * stack itself */
        block->stack_next = block->base_addr + PAGE_SIZE;
    }
    else {
        /* used block: sequential allocation no longer possible */
        block->stack_next = block->base_addr + VM_ALLOC_BLOCK_SIZE;
    }
}
```

Here is the call graph for this function:



4.76.3.12 void vm_alloc_unlink_block (vm_block_t * *block*)

Unlink memory block from free or partial block list.

It is not an error if block is not linked to either list. On exit of this funtion, the block is in the used state.

Parameters

block	block to unlink from list

ASSERTION: block is not null

ASSERTION: block is either properly linked (no null pointers) or not at all (next is null)

ASSERTION: block->allocator should not be NULL

ASSERTION: block should not be the head of both free and partial lists

ASSERTION: if block is alone in its list, the previous node pointer should point to self

ASSERTION: if block is alone in its list, we expect it to be the head of either the free or the partial list

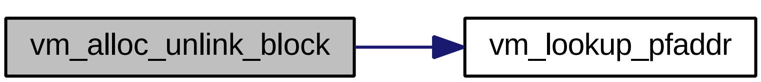
Definition at line 623 of file vm_alloc.c.

References vm_block_t::allocator, assert, vm_alloc_t::free_list, vm_block_t::next, NULL, vm_alloc_t::partial_list, pffree, vm_block_t::prev, vm_block_t::stack_addr, vm_block_t::stack_ptr, and vm_lookup_pfaddr().

Referenced by vm_alloc(), vm_alloc_free_block(), vm_alloc_grow_single(), vm_alloc_low_latency(), and vm_alloc_partial_block().

```
623                                              {
624     vm_alloc_t *allocator;
625     pfaddr_t    paddr;
626 
628     assert(block != NULL);
629 
632     assert(block->prev != NULL || block->next == NULL);
633 
635     assert(block->allocator != NULL);
636 
637     /* get allocator for block (required for next assert as well as subsequent code) */
638     allocator = block->allocator;
639 
641     assert(allocator->free_list != block || allocator->partial_list != block);
642 
643     /* if block is already unlinked, we have nothing to do here */
644     if(block->next == NULL) {
645         return;
646     }
647 
648     /* if block has a stack, discard it */
649     if(block->stack_ptr != NULL) {
650         paddr = vm_lookup_pfaddr(NULL, (addr_t)block->stack_addr);
651         pffree(paddr);
652     }
653 
654     /* special case: block is alone in its list */
655     if(block->next == block) {
658         assert(block->prev == block);
659 
662         assert(allocator->free_list == block || allocator->partial_list == block);
663 
664         if(allocator->free_list == block) {
665             allocator->free_list = NULL;
666         }
667 
668         if(allocator->partial_list == block) {
669             allocator->partial_list = NULL;
670         }
671     }
672     else {
673         if(allocator->free_list == block) {
674             allocator->free_list = block->next;
675         }
676 
677         if(allocator->partial_list == block) {
678             allocator->partial_list = block->next;
679         }
680 
681         /* unlink block */
682         block->next->prev = block->prev;
683         block->prev->next = block->next;
684     }
685 
686     /* set next pointer to null to indicate block is unlinked */
687     block->next = NULL;
688 }
```

Here is the call graph for this function:



4.76.3.13 void vm_free (vm_alloc_t * *allocator*, addr_t *page*)

Free a page of virtual address space.

Parameters

allocator	allocator which manages the memory region to which the page is freed

ASSERTION: allocator is not null

ASSERTION: ensure we are freeing to the proper allocator/region

ASSERTION: ensure address is page aligned

ASSERTION: block should now be partial

ASSERTION: stack overflow check

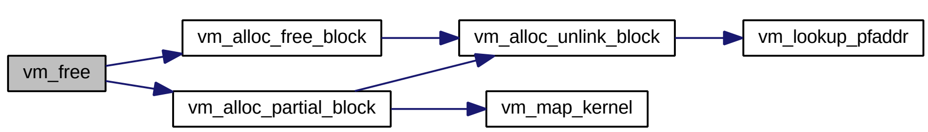
Definition at line 172 of file vm_alloc.c.

References assert, vm_alloc_t::base_addr, vm_alloc_t::block_array, NULL, page_offset_of, vm_block_t::stack_addr, vm_block_t::stack_ptr, vm_alloc_t::start_addr, VM_ALLOC_BLOCK_SIZE, vm_alloc_free_block(), VM_ALLOC_FULL_STACK, VM_ALLOC_IS_PARTIAL, VM_ALLOC_IS_USED, and vm_alloc_partial_block().

Referenced by elf_load(), elf_setup_stack(), thread_page_create(), and thread_page_destroy().

```
172                                                 {
173     vm_block_t   *block;
174     unsigned int  idx;
175 
177     assert(allocator != NULL);
178 
180     assert(page >= allocator->start_addr && page < allocator->end_addr);
181 
183     assert(page_offset_of(page) == 0);
184 
185     /* find the block to which the free page belong */
186     idx = ( (unsigned int)page - (unsigned int)allocator->base_addr ) /
      VM_ALLOC_BLOCK_SIZE;
187     block = &allocator->block_array[idx];
188 
189     /* if the block was a used block, make it a partial block */
190     if( VM_ALLOC_IS_USED(block) ) {
191         if(block->stack_addr == NULL) {
192             block->stack_addr = (addr_t *)page;
193             return;
194         }
195 
196         vm_alloc_partial_block(block);
197     }
198 
200     assert( VM_ALLOC_IS_PARTIAL(block) );
201 
203     assert( ! VM_ALLOC_FULL_STACK(block) );
204 
205     *(--block->stack_ptr) = page;
206 
207     /* check if we just freed the whole block */
208     if( VM_ALLOC_FULL_STACK(block) ) {
209         vm_alloc_free_block(block);
210     }
211 }
```

Here is the call graph for this function:



4.76.4 Variable Documentation

4.76.4.1 vm_alloc_t* global_page_allocator

global page allocator (region 0..KLIMIT)

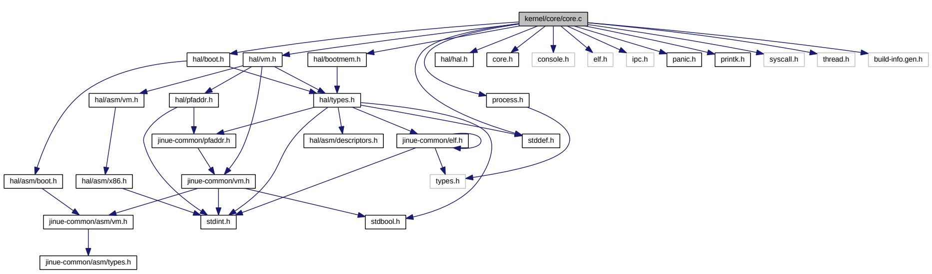
Definition at line 58 of file vm.c.

Referenced by elf_load(), elf_setup_stack(), slab_cache_grow(), thread_page_create(), and thread_page_destroy().

4.77 kernel/core/core.c File Reference

```
#include <hal/boot.h>
#include <hal/bootmem.h>
#include <hal/hal.h>
#include <hal/vm.h>
#include <core.h>
#include <console.h>
#include <elf.h>
#include <ipc.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 core.c:



Functions

- void **kmain** (void)

4.77.1 Function Documentation

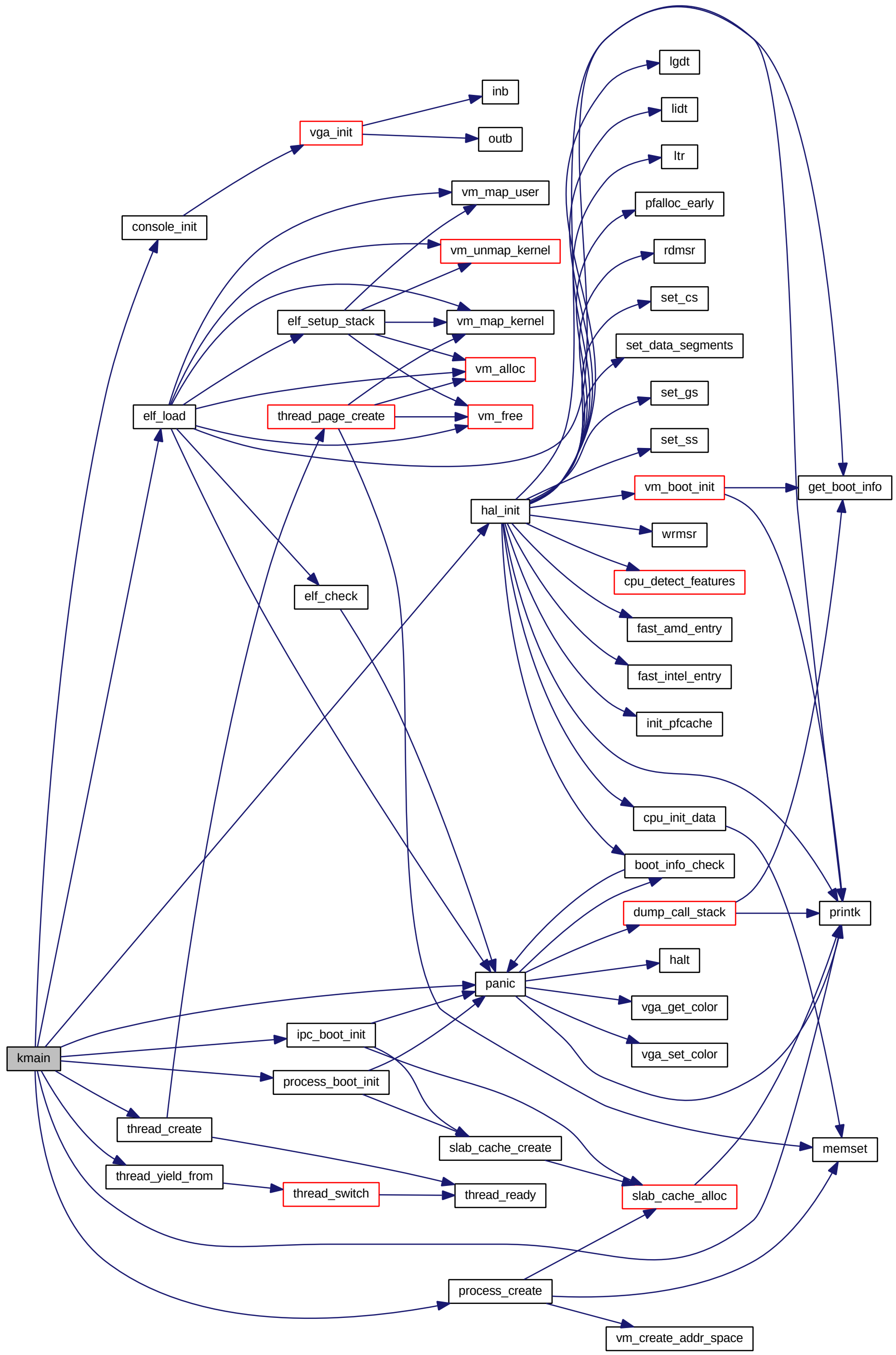
4.77.1.1 void kmain (void)

Definition at line 66 of file core.c.

References process_t::addr_space, console_init(), elf_load(), elf_info_t::entry, hal_init(), ipc_boot_init(), NULL, panic(), printk(), process_boot_init(), process_create(), elf_info_t::stack_addr, thread_create(), and thread_yield_from().

```
{
    elf_info_t elf_info;

    /* initialize console and say hello */
    console_init();

    printk("Kernel revision " GIT_REVISION " built " BUILD_TIME " on " BUILD_HOST "\n");

    /* initialize hardware abstraction layer */
    hal_init();

    /* initialize caches */
    ipc_boot_init();
    process_boot_init();

    /* create process for process manager */
    process_t *process = process_create();

    /* load process manager binary */
    Elf32_Ehdr *elf = find_process_manager();
    elf_load(&elf_info, elf, &process->addr_space);

    /* create initial thread */
    thread_t *thread = thread_create(
            process,
            elf_info.entry,
            elf_info.stack_addr);

    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:

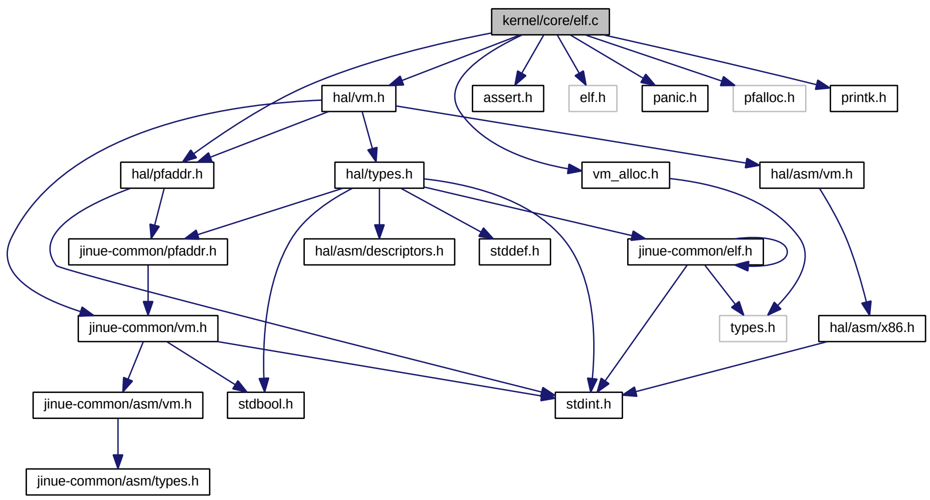


4.78 kernel/core/elf.c File Reference

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

```
#include <hal/vm.h>
#include <assert.h>
#include <elf.h>
#include <panic.h>
#include <pfalloc.h>
#include <printk.h>
#include <vm_alloc.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)
- void **elf_setup_stack** (**elf_info_t** ∗info)
- int **elf_lookup_symbol** (const **Elf32_Ehdr** ∗elf_header, **Elf32_Addr** addr, int type, **elf_symbol_t** ∗result)

4.78.1 Function Documentation

4.78.1.1 void elf_check (Elf32_Ehdr ∗ *elf*)

Definition at line 42 of file elf.c.

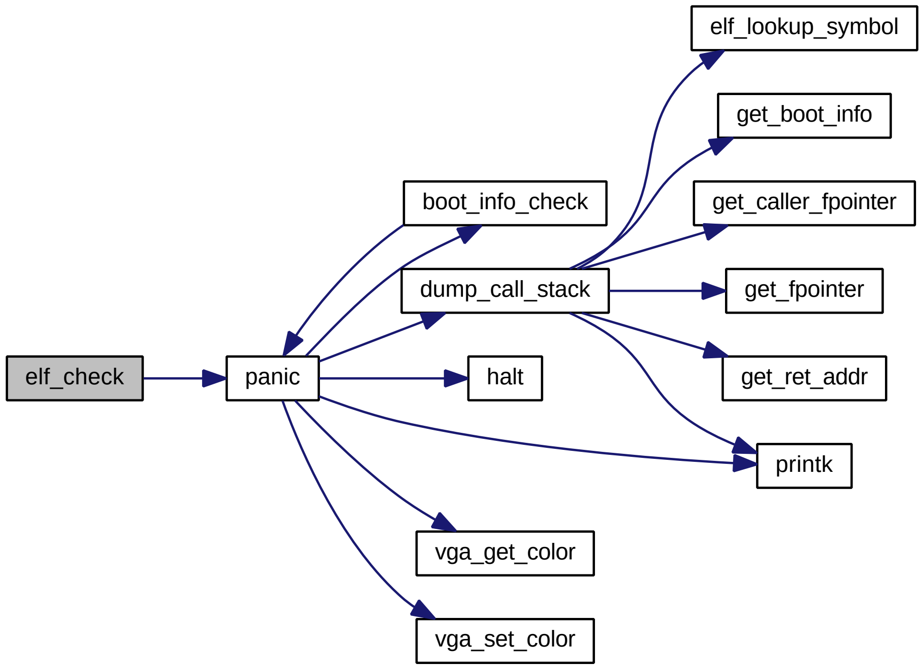
References Elf32_Ehdr::e_entry, Elf32_Ehdr::e_flags, Elf32_Ehdr::e_ident, Elf32_Ehdr::e_machine, Elf32_Ehdr::e_phentsize, Elf32_Ehdr::e_phnum, Elf32_Ehdr::e_phoff, Elf32_Ehdr::e_type, Elf32_Ehdr::e_version, EI_CLASS, EI_DATA, EI_MAG0, EI_MAG1, EI_MAG2, EI_MAG3, EI_VERSION, ELF_MAGIC0, ELF_MAGIC1, ELF_MAGIC2, ELF_MAGIC3, ELFCLASS32, ELFDATA2LSB, EM_386, ET_EXEC, and panic().

Referenced by elf_load().

```
42                              {
43      /* check: valid ELF binary magic number */
44      if(     elf->e_ident[EI_MAG0] != ELF_MAGIC0 ||
45              elf->e_ident[EI_MAG1] != ELF_MAGIC1 ||
46              elf->e_ident[EI_MAG2] != ELF_MAGIC2 ||
47              elf->e_ident[EI_MAG3] != ELF_MAGIC3 ) {
48          panic("Not an ELF binary");
49      }
50
51      /* check: 32-bit objects */
```

```
    if(elf->e_ident[EI_CLASS] != ELFCLASS32) {
        panic("Bad file class");
    }

    /* check: endianess */
    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:



4.78.1.2 void elf_load (**elf_info_t** ∗ *info,* **Elf32_Ehdr** ∗ *elf,* **addr_space_t** ∗ *addr_space*)

TODO: add exec flag once PAE is enabled

TODO: add exec flag once PAE is enabled

Definition at line 97 of file elf.c.

References elf_info_t::addr_space, elf_info_t::at_phdr, elf_info_t::at_phent, elf_info_t::at_phnum, Elf32_Ehdr::e_entry, Elf32_Ehdr::e_phentsize, Elf32_Ehdr::e_phnum, Elf32_Ehdr::e_phoff, EARLY_PTR_TO_PFADDR, elf_check(), elf-_setup_stack(), elf_info_t::entry, global_page_allocator, Elf32_Phdr::p_filesz, Elf32_Phdr::p_memsz, PAGE_MASK, page_offset_of, PAGE_SIZE, panic(), PF_W, pfalloc, printk(), PT_LOAD, vm_alloc(), VM_FLAG_READ_ONLY, VM_-FLAG_READ_WRITE, vm_free(), vm_map_kernel(), vm_map_user(), and vm_unmap_kernel().

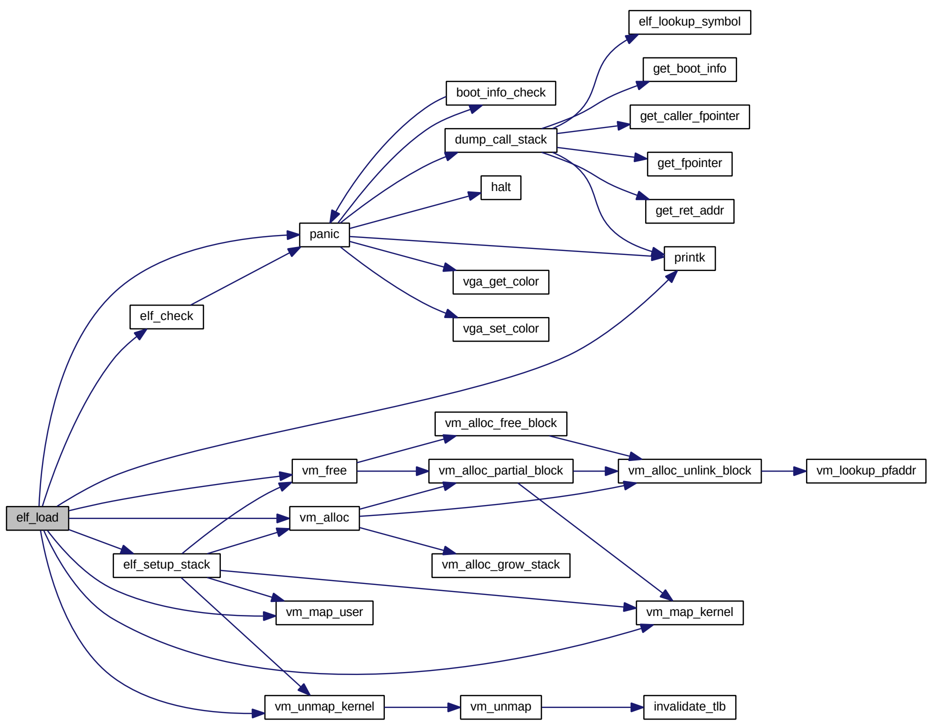
Referenced by kmain().

```
97                                                                           {
98      Elf32_Phdr *phdr;
99      pfaddr_t page;
100      addr_t vpage;
101      char *vptr, *vend, *vfend, *vnext;
102      char *file_ptr;
103      char *stop;
104      char *dest, *dest_page;
105      unsigned int idx;
106      unsigned long flags;
107
108
109      /* check that ELF binary is valid */
110      elf_check(elf);
111
112      /* get the program header table */
113      phdr = (Elf32_Phdr *)((char *)elf + elf->e_phoff);
114
115      info->at_phdr       = (addr_t)phdr;
116      info->at_phnum      = elf->e_phnum;
```

```
    info->at_phent      = elf->e_phentsize;
    info->addr_space    = addr_space;
    info->entry         = (addr_t)elf->e_entry;

    /* temporary page for copies */
    dest_page = (char *)vm_alloc(global_page_allocator);

    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 */
        file_ptr = (char *)elf + phdr[idx].p_offset;
        vptr     = (char *)phdr[idx].p_vaddr;
        vend     = vptr  + phdr[idx].p_memsz;  /* limit for padding */
        vfend    = vptr  + phdr[idx].p_filesz; /* limit for copy */

        /* align on page boundaries, be inclusive,
           note that vfend is not aligned        */
        file_ptr = (char *) ( (uintptr_t)file_ptr & ~PAGE_MASK );
        vptr     = (char *) ( (uintptr_t)vptr  & ~PAGE_MASK );

        if(page_offset_of(vend) != 0) {
            vend  = (char *) ( (uintptr_t)vend  & ~PAGE_MASK );
            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) {
                /* start of this page and next page */
                vpage = (addr_t)vptr;
                vnext = vptr + PAGE_SIZE;

                /* allocate and map the new page */
                page = pfalloc();
                vm_map_kernel((addr_t)dest_page, page, VM_FLAG_READ_WRITE);

                dest = dest_page;

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

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

                /* pad */
                while(vptr < vnext) {
                    *(dest++) = 0;
                    ++vptr;
                }

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

                /* undo temporary mapping and map page in proper address
                 * space */
                vm_unmap_kernel((addr_t)dest_page);
                vm_map_user(addr_space, (addr_t)vpage, page, flags);
            }
        }
        else {
            while(vptr < vend) {
                /* perform mapping */
```

```
                vm_map_user(addr_space, (addr_t)vptr, EARLY_PTR_TO_PFADDR(file_ptr),
      VM_FLAG_READ_ONLY);

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

    vm_free(global_page_allocator, (addr_t)dest_page);

    elf_setup_stack(info);

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

Here is the call graph for this function:



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

Definition at line 284 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;
}
```

4.78.1.4 void elf_setup_stack (elf_info_t * *info*)

TODO: check for overlap of stack with loaded segments

Definition at line 214 of file elf.c.

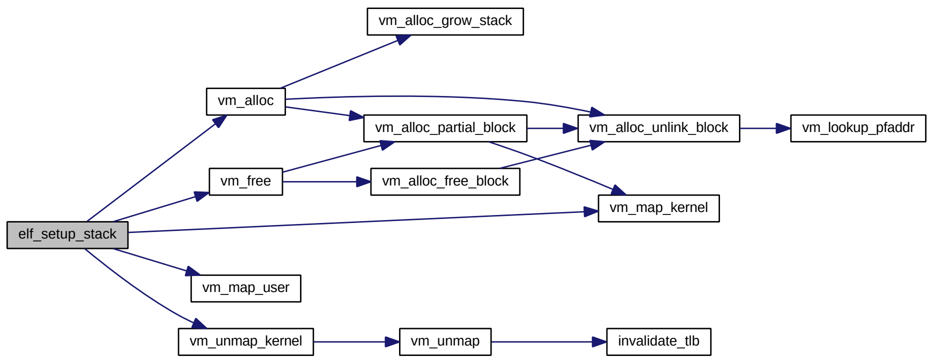
References Elf32_auxv_t::a_type, Elf32_auxv_t::a_un, Elf32_auxv_t::a_val, elf_info_t::addr_space, AT_ENTRY, AT_NULL, AT_PAGESZ, elf_info_t::at_phdr, AT_PHDR, elf_info_t::at_phent, AT_PHENT, elf_info_t::at_phnum, AT_PHNUM, AT_STACKBASE, elf_info_t::entry, global_page_allocator, PAGE_SIZE, pfalloc, elf_info_t::stack_addr, STACK_BASE, STACK_START, vm_alloc(), VM_FLAG_READ_WRITE, vm_free(), vm_map_kernel(), vm_map_user(), and vm_unmap_kernel().

Referenced by elf_load().

```
214                                         {
215     pfaddr_t page;
216     addr_t vpage;
217 
220     /* initial stack allocation */
221     for(vpage = (addr_t)STACK_START; vpage < (addr_t)STACK_BASE; vpage +=
      PAGE_SIZE) {
222         page  = pfalloc();
223         vm_map_user(info->addr_space, vpage, page, VM_FLAG_READ_WRITE);
```

```
    }

    /* 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 = vm_alloc(global_page_allocator);
    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);
    vm_free(global_page_allocator, top_page);
}
```

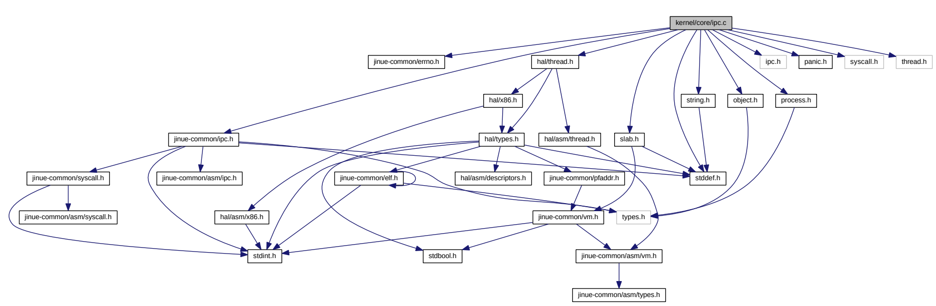
Here is the call graph for this function:



4.79 kernel/core/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** (void)
- **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)

4.79.1 Function Documentation

4.79.1.1 void ipc_boot_init (void)

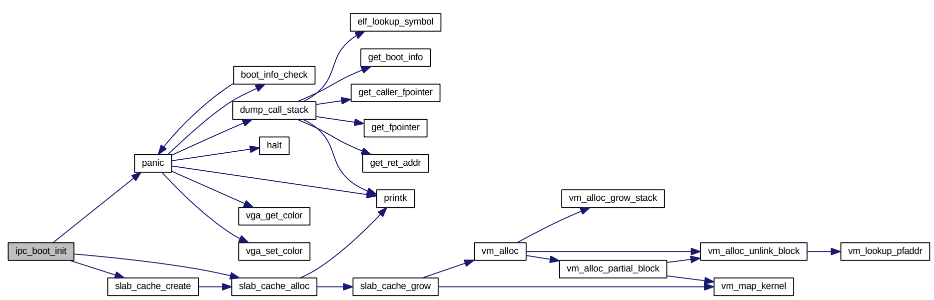
Definition at line 58 of file ipc.c.

References NULL, panic(), slab_cache_alloc(), slab_cache_create(), and SLAB_DEFAULTS.

Referenced by kmain().

```
58                          {
59     ipc_object_cache = slab_cache_create(
60             "ipc_object_cache",
61             sizeof(ipc_t),
62             0,
63             ipc_object_ctor,
64             NULL,
65             SLAB_DEFAULTS );
66 
67     proc_ipc = slab_cache_alloc(ipc_object_cache);
68 
69     if(proc_ipc == NULL) {
70         panic("Cannot create process manager IPC object.");
71     }
72 }
```

Here is the call graph for this function:



4.79.1.2 ipc_t∗ ipc_get_proc_object (void)

Definition at line 84 of file ipc.c.

Referenced by dispatch_syscall().

```
84                                 {
85     return proc_ipc;
86 }
```

4.79.1.3 ipc_t∗ ipc_object_create (int *flags*)

Definition at line 74 of file ipc.c.

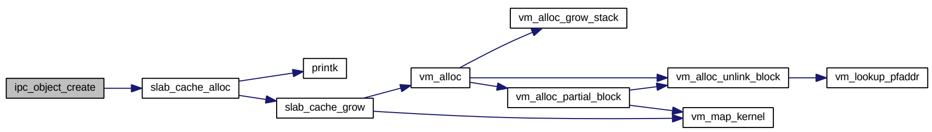
References object_header_t::flags, ipc_t::header, NULL, and slab_cache_alloc().

Referenced by dispatch_syscall().

```
74                                    {
75     ipc_t *ipc = slab_cache_alloc(ipc_object_cache);
```

```
76
77     if(ipc != NULL) {
78         ipc->header.flags = flags;
79     }
80
81     return ipc;
82 }
```

Here is the call graph for this function:



4.79.1.4 void ipc_receive (jinue_syscall_args_t ∗ *args*)

Definition at line 203 of file ipc.c.

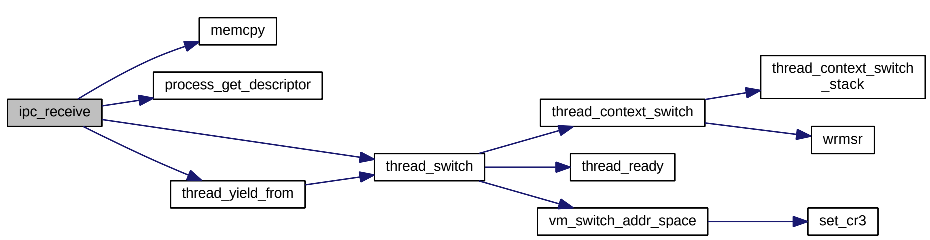
References jinue_syscall_args_t::arg0, jinue_syscall_args_t::arg1, jinue_syscall_args_t::arg2, jinue_syscall_args_t::arg3, message_info_t::data_size, object_header_t::flags, thread_t::header, JINUE_E2BIG, JINUE_EBADF, JINUE_EINVAL, JINUE_EIO, JINUE_EPERM, jinue_node_entry, memcpy(), thread_t::message_args, 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().

```
203                                         {
204     thread_t *thread = get_current_thread();
205
206     int fd = (int)args->arg1;
207
208     object_ref_t *ref = process_get_descriptor(thread->process, fd);
209
210     if(! object_ref_is_valid(ref)) {
211         syscall_args_set_error(args, JINUE_EBADF);
212         return;
213     }
214
215     if(object_ref_is_closed(ref)) {
216         syscall_args_set_error(args, JINUE_EIO);
217         return;
218     }
219
220     if(! object_ref_is_owner(ref)) {
221         syscall_args_set_error(args, JINUE_EPERM);
222         return;
223     }
224
225     object_header_t *header = ref->object;
226
227     if(object_is_destroyed(header)) {
228         ref->flags |= OBJECT_REF_FLAG_CLOSED;
229         object_subref(header);
230
231         syscall_args_set_error(args, JINUE_EIO);
232         return;
233     }
234
235     if(header->type != OBJECT_TYPE_IPC) {
236         syscall_args_set_error(args, JINUE_EBADF);
237         return;
238     }
239
240     ipc_t *ipc = (ipc_t *)header;
241
```

```
    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:



4.79.1.5 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 302 of file ipc.c.

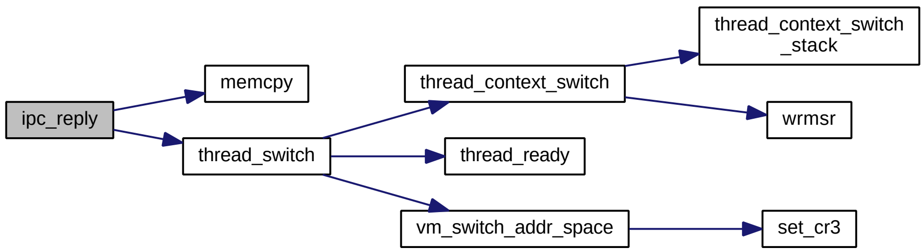
References jinue_syscall_args_t::arg2, jinue_syscall_args_t::arg3, message_info_t::buffer_size, message_info_t::data_size, message_info_t::desc_n, thread_t::header, JINUE_EINVAL, JINUE_ENOSYS, JINUE_SEND_BUFFER_SIZE_OFFSET, JINUE_SEND_MAX_N_DESC, JINUE_SEND_MAX_SIZE, JINUE_SEND_SIZE_MASK, memcpy(), thread_t::message_args, thread_t::message_buffer, thread_t::message_info, NULL, thread_t::sender, and thread_switch().

Referenced by dispatch_syscall().

```
302                                             {
303     thread_t *thread        = get_current_thread();
304     thread_t *send_thread   = thread->sender;
305 
306     if(send_thread == NULL) {
308         syscall_args_set_error(args, JINUE_EINVAL);
309         return;
310     }
311 
312     size_t buffer_size   = jinue_args_get_buffer_size(args);
313     size_t data_size     = jinue_args_get_data_size(args);
314     size_t desc_n        = jinue_args_get_n_desc(args);
315     size_t total_size    =
316             data_size +
317             desc_n * sizeof(jinue_ipc_descriptor_t);
318 
319     if(buffer_size > JINUE_SEND_MAX_SIZE) {
320         syscall_args_set_error(args, JINUE_EINVAL);
321         return;
322     }
323 
324     if(total_size > buffer_size) {
325         syscall_args_set_error(args, JINUE_EINVAL);
326         return;
327     }
328 
329     if(desc_n > JINUE_SEND_MAX_N_DESC) {
330         syscall_args_set_error(args, JINUE_EINVAL);
331         return;
332     }
333 
334     /* the reply must fit in the sender's buffer */
335     if(total_size > send_thread->message_info.buffer_size) {
336         syscall_args_set_error(args, JINUE_EINVAL);
337         return;
338     }
339 
341     if(desc_n > 0) {
342         syscall_args_set_error(args, JINUE_ENOSYS);
343         return;
344     }
345 
346     const char *user_ptr = (const char *)args->arg2;
347 
348     if(! user_buffer_check(user_ptr, buffer_size)) {
349         syscall_args_set_error(args, JINUE_EINVAL);
350         return;
351     }
352 
353     memcpy(&send_thread->message_buffer, user_ptr, data_size);
354 
358     syscall_args_set_return(send_thread->message_args, 0);
359     send_thread->message_args->arg3 =
360             args->arg3 & ~(JINUE_SEND_SIZE_MASK << JINUE_SEND_BUFFER_SIZE_OFFSET);
361 
362     send_thread->message_info.data_size = data_size;
363     send_thread->message_info.desc_n    = desc_n;
364 
```

```
    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:



4.79.1.6 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 88 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.

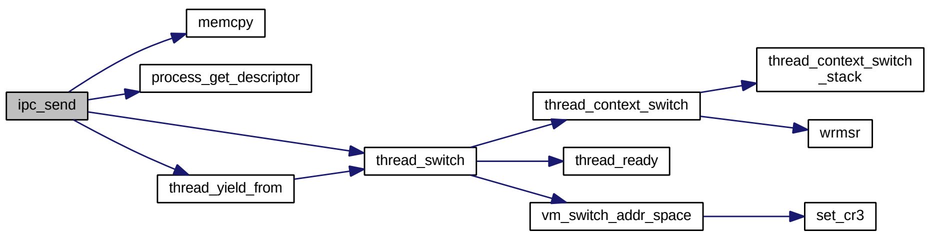
Referenced by dispatch_syscall().

```
88                                       {
89      thread_t *thread = get_current_thread();
90
91      message_info_t *message_info = &thread->message_info;
92
93      message_info->function      = args->arg0;
94      message_info->buffer_size   = jinue_args_get_buffer_size(args);
95      message_info->data_size     = jinue_args_get_data_size(args);
96      message_info->desc_n        = jinue_args_get_n_desc(args);
97      message_info->total_size    =
98              message_info->data_size +
99              message_info->desc_n * sizeof(jinue_ipc_descriptor_t);
100
101     if(message_info->buffer_size > JINUE_SEND_MAX_SIZE) {
102         syscall_args_set_error(args, JINUE_EINVAL);
103         return;
104     }
105
106     if(message_info->total_size > message_info->buffer_size) {
107         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);

}
```

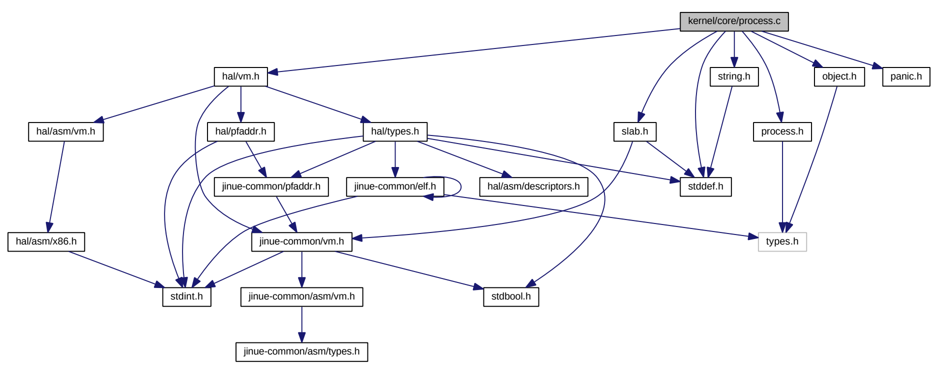
Here is the call graph for this function:



4.80 kernel/core/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** (void)
- **process_t** ∗ **process_create** (void)
- **object_ref_t** ∗ **process_get_descriptor** (**process_t** ∗process, int fd)
- int **process_unused_descriptor** (**process_t** ∗process)

4.80.1 Function Documentation

4.80.1.1 void process_boot_init (void)

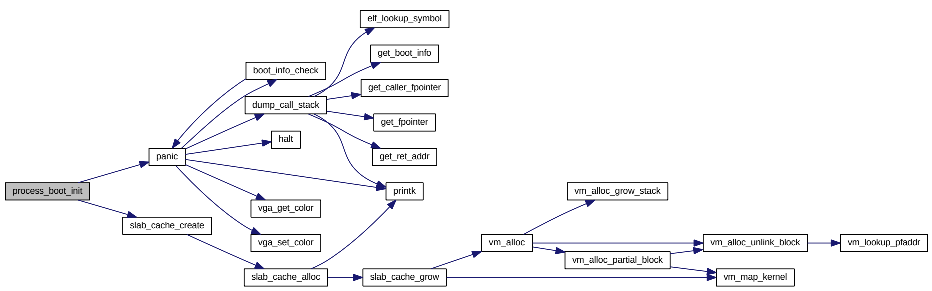
Definition at line 49 of file process.c.

References NULL, panic(), slab_cache_create(), and SLAB_DEFAULTS.

Referenced by kmain().

```
49                          {
50     process_cache = slab_cache_create(
51             "process_cache",
52             sizeof(process_t),
53             0,
54             process_ctor,
55             NULL,
56             SLAB_DEFAULTS );
57 
58     if(process_cache == NULL) {
59         panic("Cannot create process slab cache.");
60     }
61 }
```

Here is the call graph for this function:



4.80.1.2 process_t∗ process_create (void)

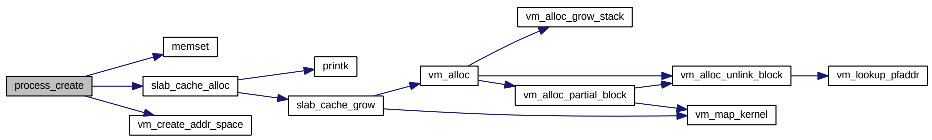
Definition at line 63 of file process.c.

References process_t::addr_space, process_t::descriptors, memset(), NULL, slab_cache_alloc(), and vm_create_addr_space().

Referenced by kmain().

```
63                              {
64     process_t *process = slab_cache_alloc(process_cache);
65 
66     if(process != NULL) {
67         vm_create_addr_space(&process->addr_space);
68         memset(&process->descriptors, 0, sizeof(process->descriptors));
69     }
70 
71     return process;
72 }
```

Here is the call graph for this function:



4.80.1.3 object_ref_t* process_get_descriptor (process_t * *process,* int *fd*)

Definition at line 74 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().

```
74                                                                   {
75      if(fd < 0 || fd > PROCESS_MAX_DESCRIPTORS) {
76          return NULL;
77      }
78
79      return &process->descriptors[fd];
80 }
```

4.80.1.4 int process_unused_descriptor (process_t * *process*)

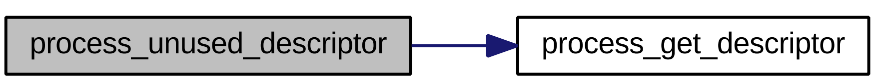
Definition at line 82 of file process.c.

References process_get_descriptor(), and PROCESS_MAX_DESCRIPTORS.

Referenced by dispatch_syscall().

```
82                                                     {
83      int idx;
84
85      for(idx = 0; idx < PROCESS_MAX_DESCRIPTORS; ++idx) {
86          object_ref_t *ref = process_get_descriptor(process, idx);
87
88          if(! object_ref_is_valid(ref)) {
89              return idx;
90          }
91      }
92
93      return -1;
94 }
```

Here is the call graph for this function:

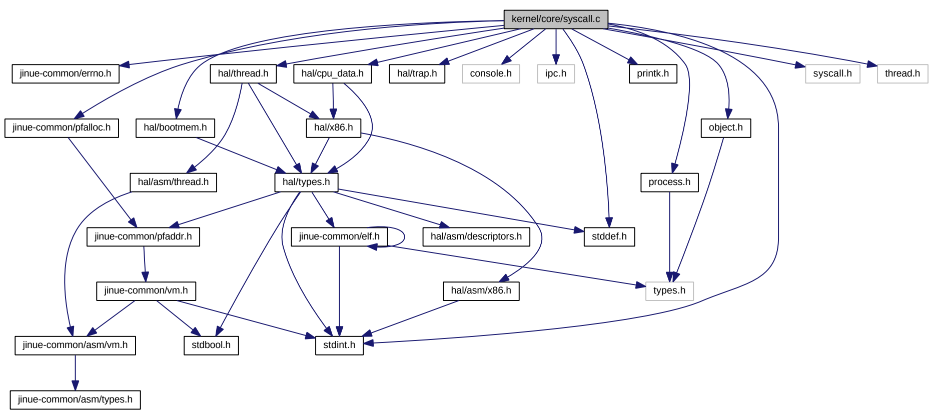


4.81 kernel/core/syscall.c File Reference

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

```
#include <jinue-common/pfalloc.h>
#include <hal/bootmem.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)

4.81.1 Function Documentation

4.81.1.1 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 49 of file syscall.c.

References bootmem_t::addr, memory_block_t::addr, jinue_syscall_args_t::arg0, jinue_syscall_args_t::arg1, jinue_-syscall_args_t::arg2, jinue_syscall_args_t::arg3, bootmem_get_block(), bootmem_root, console_printn(), console_-putc(), object_ref_t::cookie, bootmem_t::count, memory_block_t::count, object_ref_t::flags, ipc_t::header, IPC_FLAG-_NONE, IPC_FLAG_SYSTEM, ipc_get_proc_object(), ipc_object_create(), ipc_receive(), ipc_reply(), ipc_send(), JINU-E_EAGAIN, JINUE_EMORE, JINUE_ENOSYS, JINUE_IPC_PROC, JINUE_IPC_SYSTEM, NULL, object_ref_t::object, OBJECT_REF_FLAG_OWNER, OBJECT_REF_FLAG_VALID, printk(), thread_t::process, process_get_descriptor(), process_unused_descriptor(), SYSCALL_FUNCT_CONSOLE_PUTC, SYSCALL_FUNCT_CONSOLE_PUTS, SYSCALL_FUNCT_CREATE_IPC, SYSCALL_FUNCT_GET_FREE_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(), and thread_yield_from().

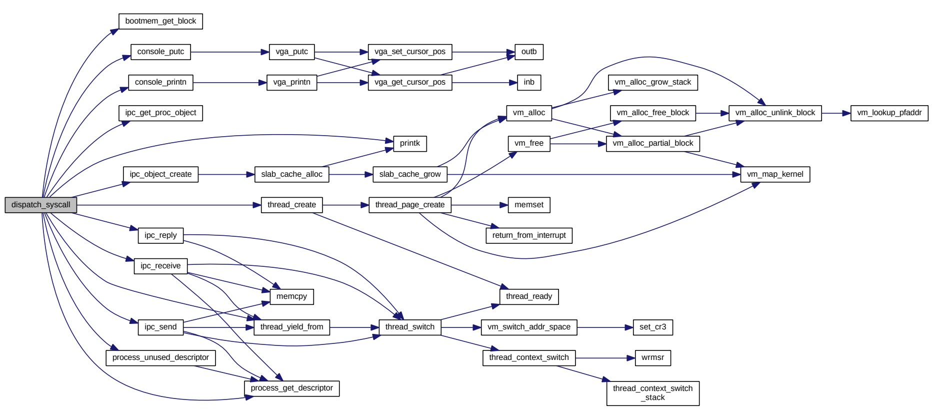
Referenced by dispatch_interrupt().

```
                                          {
    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);
            syscall_args_set_return(args, 0);
            break;

        case SYSCALL_FUNCT_CONSOLE_PUTS:
            console_printn((char *)args->arg2, jinue_args_get_data_size(args));
            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_FREE_MEMORY:
        {
            bootmem_t      *block;
            memory_block_t *block_dest;
            unsigned int count, count_max;

```

```
            size_t buffer_size  = jinue_args_get_buffer_size(args);
            block_dest          = (memory_block_t *)jinue_args_get_buffer_ptr(args);

            count_max = buffer_size / sizeof(memory_block_t);

            for(count = 0; count < count_max; ++count) {
                block = bootmem_get_block();

                if(block == NULL) {
                    break;
                }

                block_dest->addr  = block->addr;
                block_dest->count = block->count;

                ++block_dest;
            }

            args->arg0 = (uintptr_t)count;

            if(count == count_max && bootmem_root != NULL) {
                args->arg1  = JINUE_EMORE;
            }
            else {
                args->arg1  = 0;
            }

            args->arg2 = 0;
            args->arg3 = 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);
    }
}
```

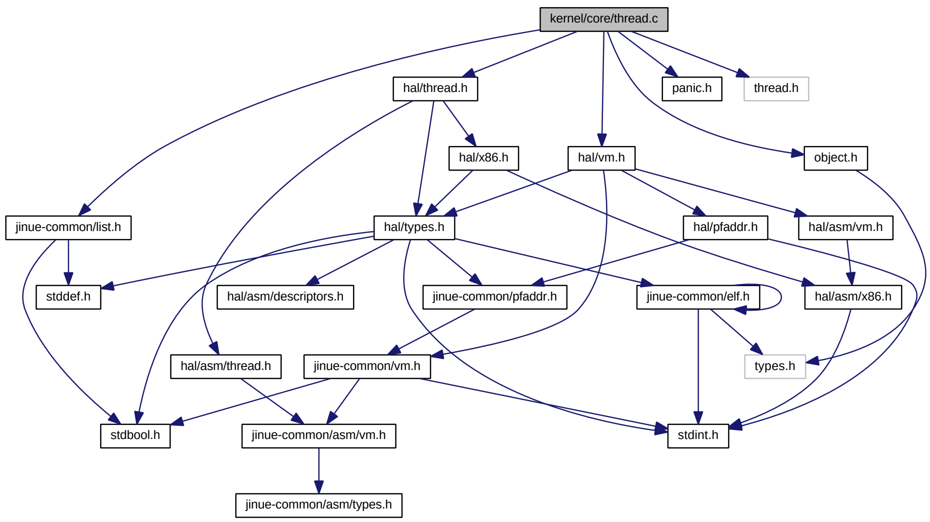
Here is the call graph for this function:



4.82 kernel/core/thread.c File Reference

```
#include <jinue-common/list.h>
#include <hal/thread.h>
#include <hal/vm.h>
#include <object.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)
- 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)

4.82.1 Function Documentation

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

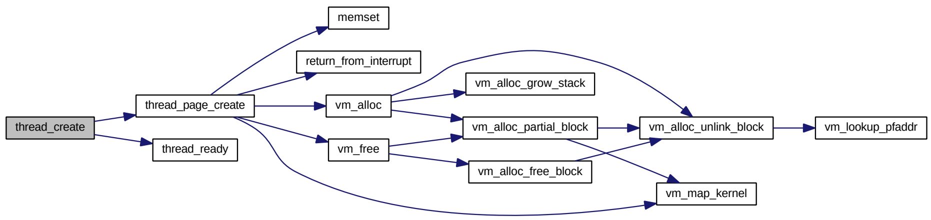
Definition at line 42 of file thread.c.

References thread_t::header, NULL, OBJECT_TYPE_THREAD, thread_t::process, thread_t::sender, thread_t::thread_list, thread_page_create(), and thread_ready().

Referenced by dispatch_syscall(), and kmain().

```
                                      {

    thread_t *thread = thread_page_create(entry, user_stack);

    if(thread != NULL) {
        object_header_init(&thread->header, OBJECT_TYPE_THREAD);

        jinue_node_init(&thread->thread_list);

        thread->process     = process;
        thread->sender      = NULL;

        thread_ready(thread);
    }

    return thread;
}
```

Here is the call graph for this function:



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

Definition at line 63 of file thread.c.

References thread_t::thread_list.

Referenced by thread_create(), and thread_switch().

```
63                                        {
64      /* add thread to the tail of the ready list to give other threads a chance
65       * to run */
66      jinue_list_enqueue(&ready_list, &thread->thread_list);
67 }
```

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

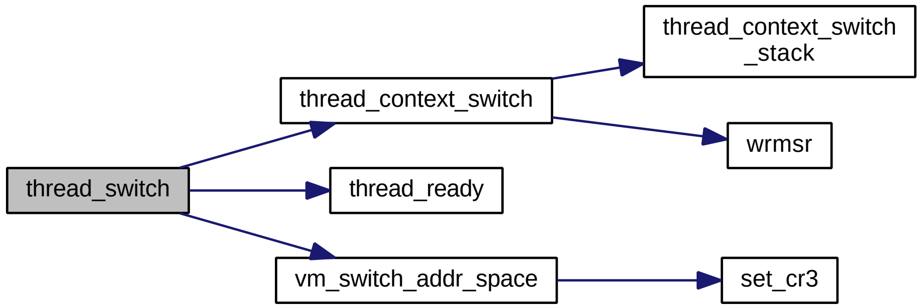
Definition at line 69 of file thread.c.

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().

```
73                            {
74
75      if(to_thread != from_thread) {
76          thread_context_t    *from_context;
77          process_t           *from_process;
78
79          if(from_thread == NULL) {
80              from_context = NULL;
81              from_process = NULL;
82          }
83          else {
84              from_context = &from_thread->thread_ctx;
85              from_process = from_thread->process;
86
87              /* Put the the thread we are switching away from (the current thread)
88               * back into the ready list, unless it just blocked or it is being
89               * destroyed. */
90              if(! (do_destroy || blocked)) {
91                  thread_ready(from_thread);
92              }
93          }
94
95          if(from_process != to_thread->process) {
96              vm_switch_addr_space(&to_thread->process->addr_space);
97          }
98
99          thread_context_switch(
100              from_context,
101              &to_thread->thread_ctx,
102              do_destroy);
103     }
104 }
```

Here is the call graph for this function:



4.82.1.4 void thread_yield_from (**thread_t** ∗ *from_thread,* **bool** *blocked,* **bool** *do_destroy*)

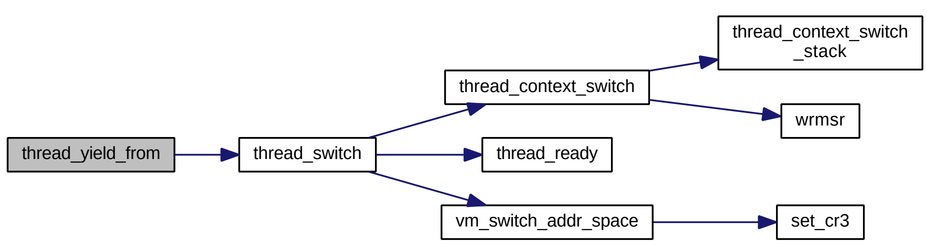
Definition at line 130 of file thread.c.

References thread_switch().

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

```
130                                                                                  {
131     bool from_can_run = ! (blocked || do_destroy);
132
133     thread_switch(
134             from_thread,
135             reschedule(from_thread, from_can_run),
136             blocked,
137             do_destroy);
138 }
```

Here is the call graph for this function:

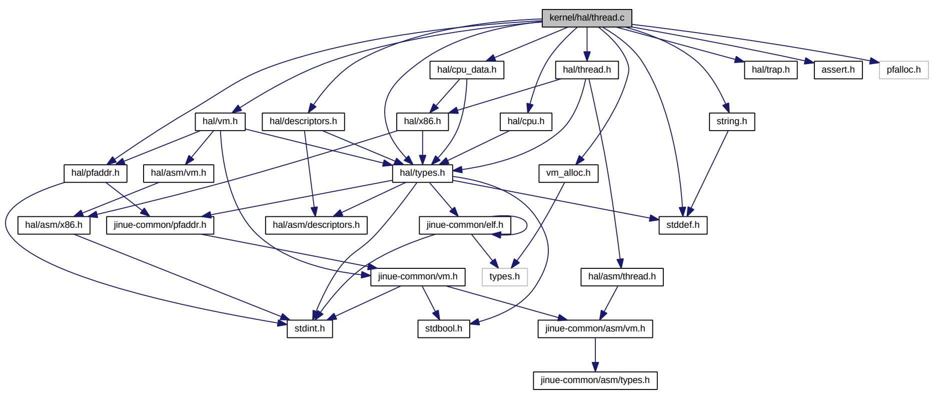


4.83 kernel/hal/thread.c File Reference

`#include <hal/cpu.h>`

```
#include <hal/cpu_data.h>
#include <hal/descriptors.h>
#include <hal/pfaddr.h>
#include <hal/thread.h>
#include <hal/trap.h>
#include <hal/types.h>
#include <hal/vm.h>
#include <assert.h>
#include <pfalloc.h>
#include <stddef.h>
#include <string.h>
#include <vm_alloc.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_create** (**addr_t** entry, **addr_t** user_stack)
- void **thread_page_destroy** (**thread_t** ∗thread)
- void **thread_context_switch** (**thread_context_t** ∗from_ctx, **thread_context_t** ∗to_ctx, **bool** destroy_from)

4.83.1 Function Documentation

4.83.1.1 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 145 of file thread.c.

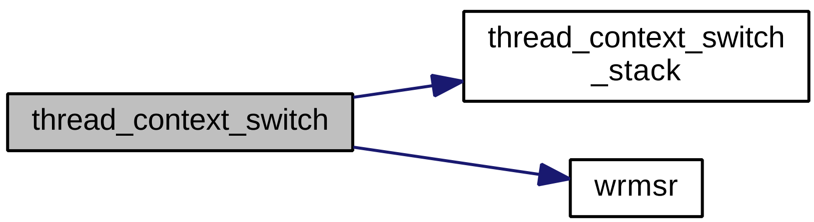
References assert, CPU_FEATURE_SYSENTER, tss_t::esp0, tss_t::esp1, tss_t::esp2, MSR_IA32_SYSENTER_ESP, NULL, thread_context_switch_stack(), and wrmsr().

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:



4.83.1.2 void thread_context_switch_stack (thread_context_t ∗ *from_ctx*, thread_context_t ∗ *to_ctx*, bool *destroy_from*)

Referenced by thread_context_switch().

4.83.1.3 thread_t∗ thread_page_create (addr_t *entry*, addr_t *user_stack*)

Definition at line 85 of file thread.c.

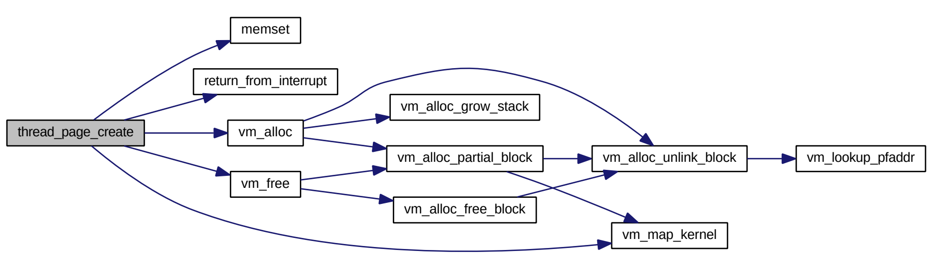
References trapframe_t::cs, trapframe_t::ds, trapframe_t::eflags, trapframe_t::eip, kernel_context_t::eip, trapframe_t::es, trapframe_t::esp, trapframe_t::fs, GDT_USER_CODE, GDT_USER_DATA, global_page_allocator, trapframe_t::gs, thread_context_t::local_storage_addr, memset(), NULL, pfalloc, PFNULL, return_from_interrupt(), RPL_USER, thread_context_t::saved_stack_pointer, SEG_SELECTOR, trapframe_t::ss, vm_alloc(), VM_FLAG_READ_WRITE, vm_free(), and vm_map_kernel().

Referenced by thread_create().

```
                                        {

    /* allocate thread context */
    thread_t *thread = (thread_t *)vm_alloc( global_page_allocator );

    if(thread != NULL) {
        pfaddr_t pf = pfalloc();

        if(pf == PFNULL) {
            vm_free(global_page_allocator, (addr_t)thread);
            return NULL;
        }

        vm_map_kernel((addr_t)thread, pf, VM_FLAG_READ_WRITE);

```

```
        /* initialize fields */
        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:



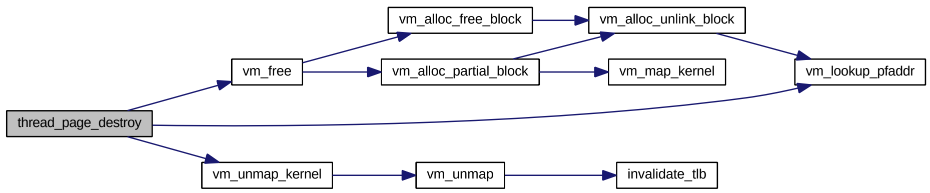
4.83.1.4 void thread_page_destroy (thread_t * *thread*)

Definition at line 138 of file thread.c.

References global_page_allocator, NULL, pffree, vm_free(), vm_lookup_pfaddr(), and vm_unmap_kernel().

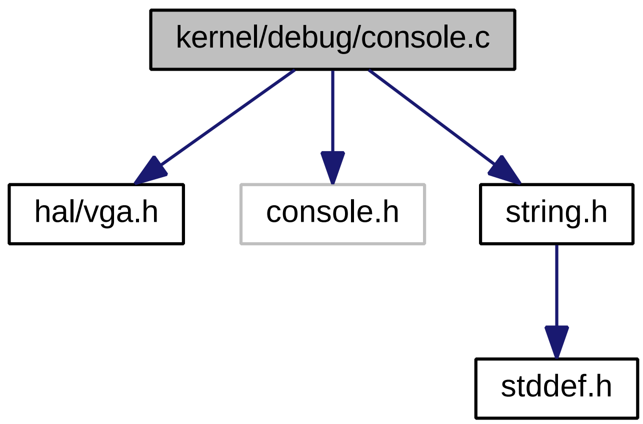
```
138                                     {
139     pfaddr_t pfaddr = vm_lookup_pfaddr(NULL, (addr_t)thread);
140     vm_unmap_kernel((addr_t)thread);
141     vm_free(global_page_allocator, (addr_t)thread);
142     pffree(pfaddr);
143 }
```

Here is the call graph for this function:



4.84 kernel/debug/console.c File Reference

#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)
- void **console_putc** (char c)
- void **console_print** (const char ∗message)

4.84.1 Function Documentation

4.84.1.1 void console_init (void)

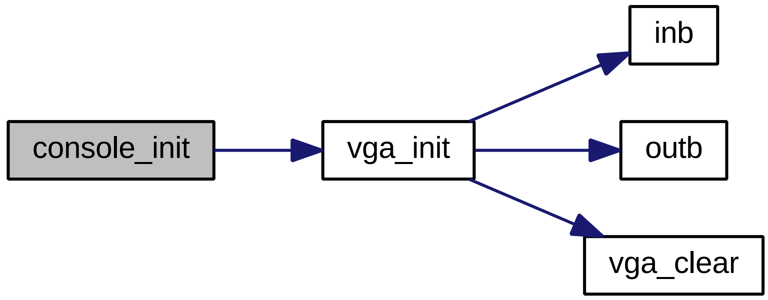
Definition at line 37 of file console.c.

References vga_init().

Referenced by kmain().

```
37                              {
38     vga_init();
39 }
```

Here is the call graph for this function:



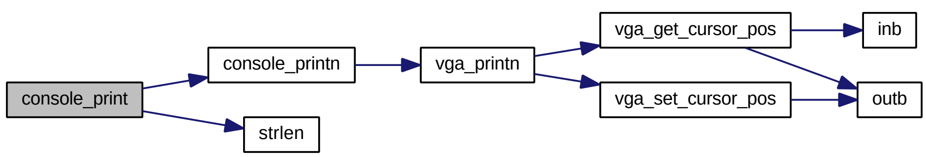
4.84.1.2 void console_print (const char ∗ *message*)

Definition at line 49 of file console.c.

References console_printn(), and strlen().

```
49                                      {
50      console_printn(message, strlen(message));
51 }
```

Here is the call graph for this function:



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

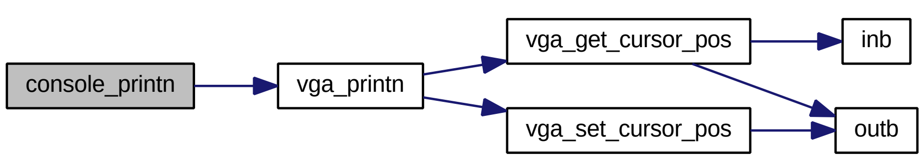
Definition at line 41 of file console.c.

References vga_printn().

Referenced by console_print(), and dispatch_syscall().

```
41                                                         {
42      vga_printn(message, n);
43 }
```

Here is the call graph for this function:



4.84.1.4 void console_putc (char *c*)

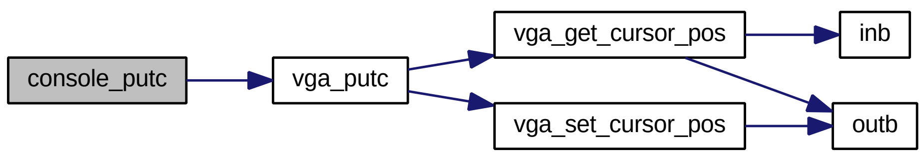
Definition at line 45 of file console.c.

References vga_putc().

Referenced by dispatch_syscall().

```
45                          {
46     vga_putc(c);
47 }
```

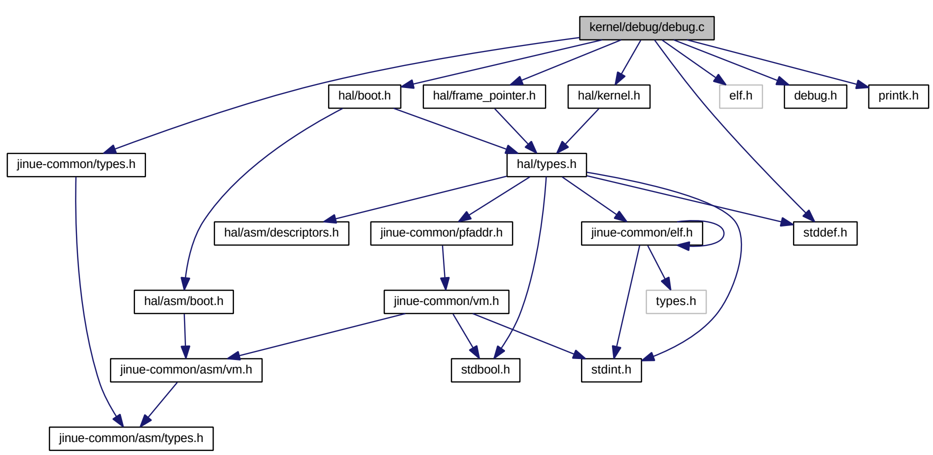
Here is the call graph for this function:



4.85 kernel/debug/debug.c File Reference

```
#include <jinue-common/types.h>
#include <hal/boot.h>
#include <hal/frame_pointer.h>
#include <hal/kernel.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)

4.85.1 Function Documentation

4.85.1.1 void dump_call_stack (void)

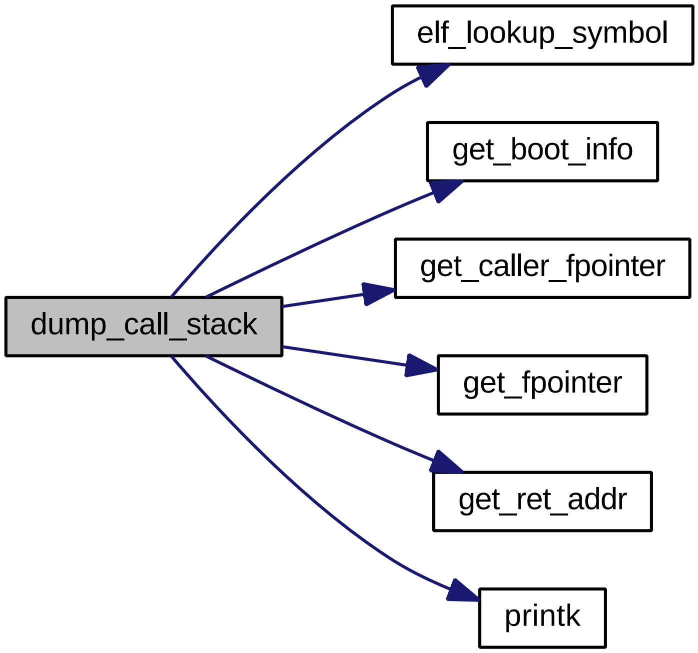
Definition at line 42 of file debug.c.

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().

```
42                          {
43     addr_t               fptr;
44 
45     const boot_info_t *boot_info = get_boot_info();
46 
47     printk("Call stack dump:\n");
48 
49     fptr = get_fpointer();
50 
51     while(fptr != NULL) {
52         addr_t return_addr = get_ret_addr(fptr);
53         if(return_addr == NULL) {
54             break;
55         }
56 
57         /* assume e8 xx xx xx xx for call instruction encoding */
58         return_addr -= 5;
59 
60         elf_symbol_t symbol;
61         int retval = elf_lookup_symbol(
62                 boot_info->kernel_start,
63                 (Elf32_Addr)return_addr,
64                 STT_FUNCTION,
65                 &symbol);
66 
67         if(retval < 0) {
68             printk("\t0x%x (unknown)\n", return_addr);
69         }
70         else {
71             const char *name = symbol.name;
72 
73             if(name == NULL) {
74                 name = "[unknown]";
75             }
76 
77             printk(
78                     "\t0x%x (%s+%u)\n",
79                     return_addr,
80                     name,
81                     return_addr - symbol.addr);
82         }
83 
84         fptr = get_caller_fpointer(fptr);
85     }
86 }
```

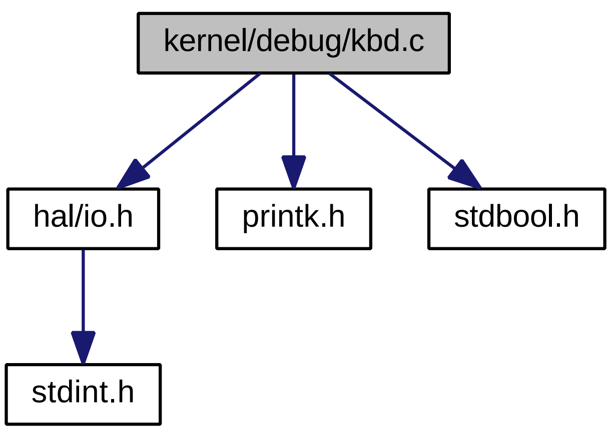
Here is the call graph for this function:



4.86 kernel/debug/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)

4.86.1 Function Documentation

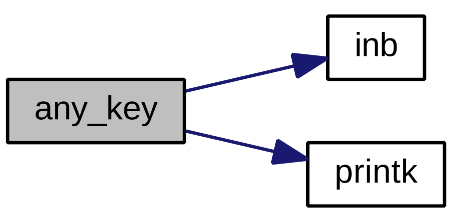
4.86.1.1 void any_key (void)

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 );
49
50         buffer = inb(0x60);
51
52         if(buffer == 0x0e || buffer == 0x0f) {
53             ignore = true;
54             continue;
55         }
56
57         if(ignore) {
58             ignore = false;
59             continue;
60         }
61
62         if(buffer == 0x1c || buffer == 0x5a) {
63             break;
64         }
65     }
66
67     /* advance cursor */
68     printk("\n");
69 }
```

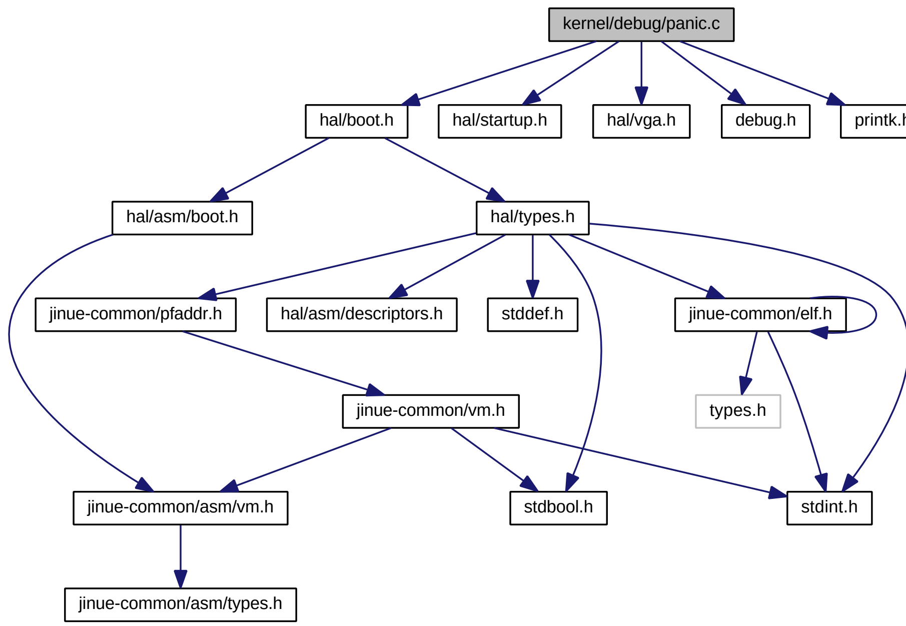
Here is the call graph for this function:



4.87 kernel/debug/panic.c File Reference

```
#include <hal/boot.h>
#include <hal/startup.h>
#include <hal/vga.h>
#include <debug.h>
#include <printk.h>
```

Include dependency graph for panic.c:



Functions

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

4.87.1 Function Documentation

4.87.1.1 void panic (const char ∗ *message*)

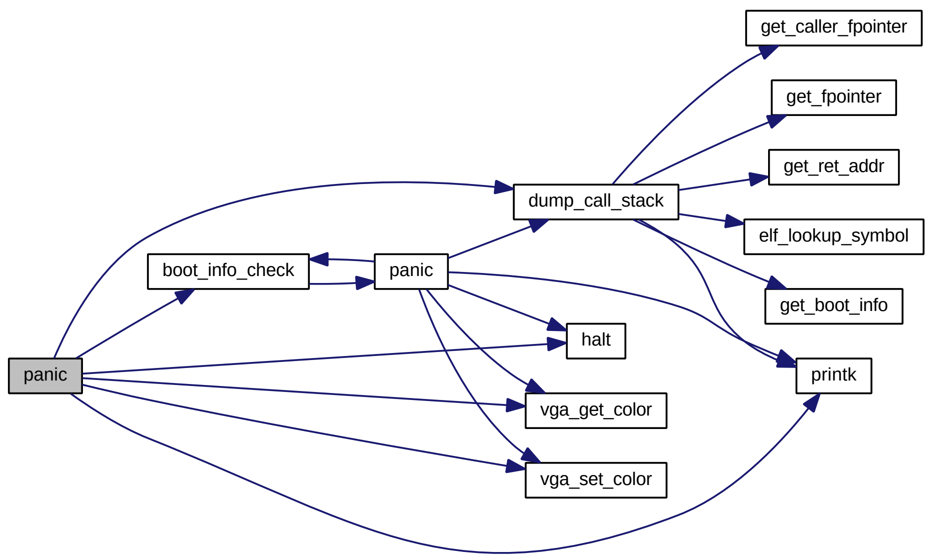
Definition at line 39 of file panic.c.

References boot_info_check(), dump_call_stack(), halt(), printk(), VGA_COLOR_RED, vga_get_color(), and vga_set_-color().

Referenced by __assert_failed(), boot_info_check(), bootmem_init(), dispatch_interrupt(), elf_check(), elf_load(), ipc_-boot_init(), kmain(), pfalloc_from(), process_boot_init(), and vm_pae_create_pdpt_cache().

```
39                              {
40     unsigned int color;
41 
42     color = vga_get_color();
43     vga_set_color(VGA_COLOR_RED);
44 
45     printk("KERNEL PANIC: %s\n", message);
46 
47     vga_set_color(color);
48 
49     if( boot_info_check(false) ) {
50         dump_call_stack();
51     }
52     else {
53         printk("Cannot dump call stack because boot information structure is invalid.\n");
54     }
55 
56     halt();
57 }
```

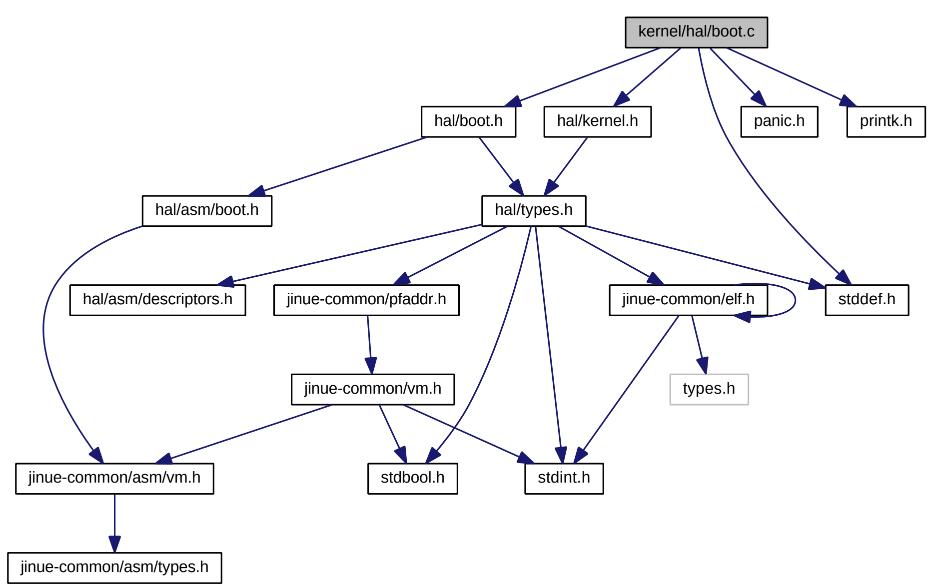
Here is the call graph for this function:



4.88 kernel/hal/boot.c File Reference

```
#include <hal/boot.h>
#include <hal/kernel.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**

4.88.1 Function Documentation

4.88.1.1 bool boot_info_check (bool *panic_on_failure*)

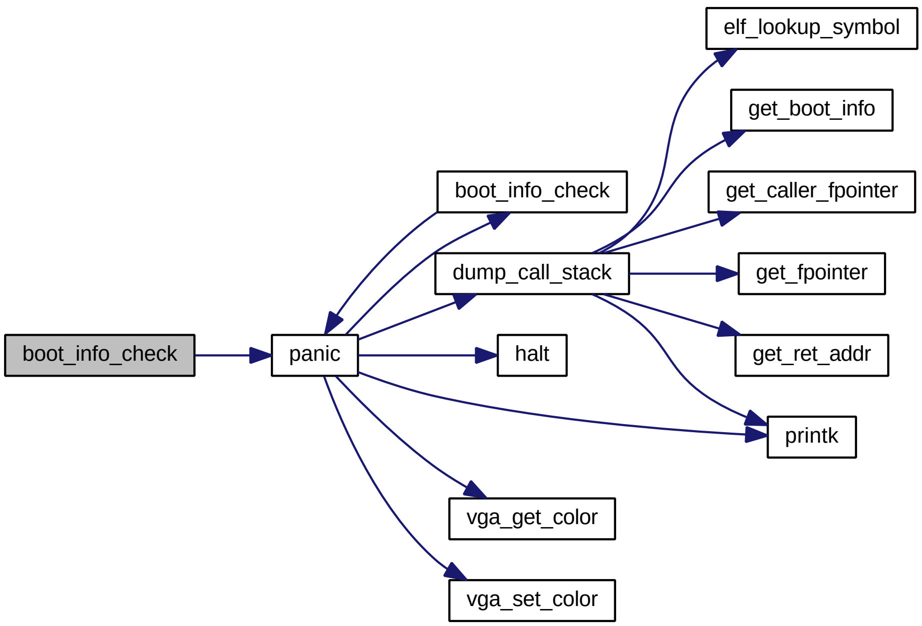
Definition at line 41 of file boot.c.

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

Referenced by hal_init(), and panic().

```
41                                             {
42     /* This data structure is accessed early during the boot process, before
43      * paging is enabled. What this means is that, if boot_info is NULL and we
44      * dereference it, it does *not* cause a page fault or any other CPU
45      * exception. */
46     if(boot_info == NULL) {
47         if(panic_on_failure) {
48             panic("Boot information structure pointer is NULL.");
49         }
50
51         return false;
52     }
53
54     if(boot_info->setup_signature != BOOT_SETUP_MAGIC) {
55         if(panic_on_failure) {
56             panic("Bad setup header signature.");
57         }
58
59         return false;
60     }
61
62     return true;
63 }
```

Here is the call graph for this function:



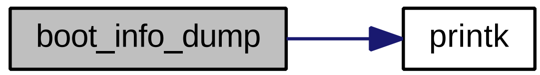
4.88.1.2 void boot_info_dump (void)

Definition at line 69 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.

```
69                          {
70     printk("Boot information structure:\n");
71     printk("    kernel_start    %x  %u\n", boot_info->kernel_start    , boot_info->kernel_start    );
72     printk("    kernel_size     %x  %u\n", boot_info->kernel_size     , boot_info->kernel_size     );
73     printk("    proc_start      %x  %u\n", boot_info->proc_start      , boot_info->proc_start      );
74     printk("    proc_size       %x  %u\n", boot_info->proc_size       , boot_info->proc_size       );
75     printk("    image_start     %x  %u\n", boot_info->image_start     , boot_info->image_start     );
76     printk("    image_top       %x  %u\n", boot_info->image_top       , boot_info->image_top       );
77     printk("    e820_entries    %x  %u\n", boot_info->e820_entries    , boot_info->e820_entries    );
78     printk("    e820_map        %x  %u\n", boot_info->e820_map        , boot_info->e820_map        );
79     printk("    boot_heap       %x  %u\n", boot_info->boot_heap       , boot_info->boot_heap       );
80     printk("    boot_end        %x  %u\n", boot_info->boot_end        , boot_info->boot_end        );
81     printk("    page_table      %x  %u\n", boot_info->page_table      , boot_info->page_table      );
82     printk("    page_directory  %x  %u\n", boot_info->page_directory  , boot_info->page_directory  );
83     printk("    setup_signature %x  %u\n", boot_info->setup_signature, boot_info->setup_signature );
84 }
```

Here is the call graph for this function:



4.88.1.3 const boot_info_t* get_boot_info (void)

Definition at line 65 of file boot.c.

References boot_info.

Referenced by bootmem_init(), dump_call_stack(), e820_dump(), hal_init(), and vm_boot_init().

```
65                                  {
66     return boot_info;
67 }
```

4.88.2 Variable Documentation

4.88.2.1 const boot_info_t* boot_info

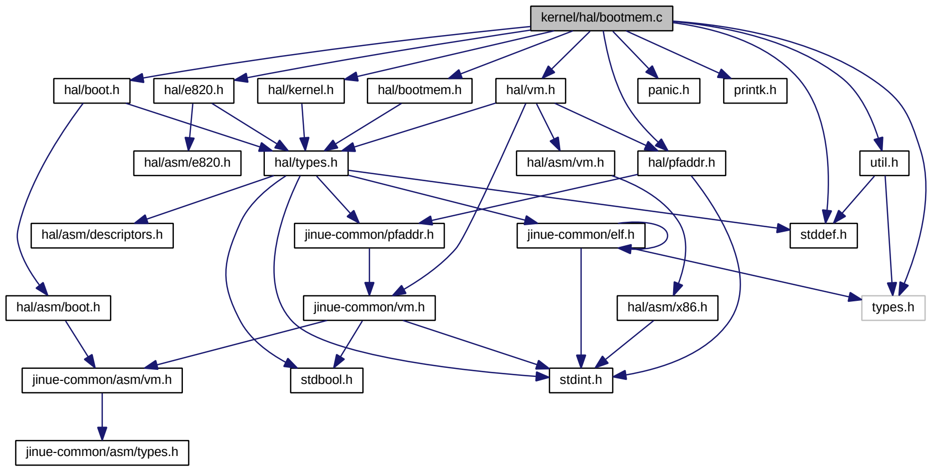
Definition at line 39 of file boot.c.

Referenced by bootmem_init(), dump_call_stack(), e820_dump(), get_boot_info(), hal_init(), and vm_boot_init().

4.89 kernel/hal/bootmem.c File Reference

```
#include <hal/boot.h>
#include <hal/bootmem.h>
#include <hal/e820.h>
#include <hal/kernel.h>
#include <hal/pfaddr.h>
#include <hal/vm.h>
#include <panic.h>
#include <printk.h>
#include <stddef.h>
#include <types.h>
#include <util.h>
```

Include dependency graph for bootmem.c:



Functions

- void **new_ram_map_entry** (**pfaddr_t** addr, **uint32_t** count, **bootmem_t** ∗∗head)
- void **apply_mem_hole** (**e820_addr_t** hole_start, **e820_addr_t** hole_end, **bootmem_t** ∗∗head)
- void **bootmem_init** (**bool** use_pae)
- **bootmem_t** ∗ **bootmem_get_block** (void)

Variables

- **bootmem_t** ∗ **ram_map**

  kernel memory map
- **bootmem_t** ∗ **bootmem_root**

  available memory map (allocator)
- void ∗ **boot_heap**

  current top of boot heap

4.89.1 Function Documentation

4.89.1.1 void apply_mem_hole (e820_addr_t *hole_start,* e820_addr_t *hole_end,* bootmem_t ∗∗ *head*)

Definition at line 68 of file bootmem.c.

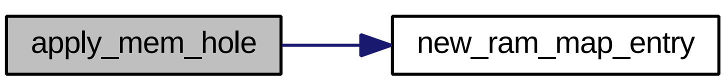
References bootmem_t::addr, bootmem_t::count, new_ram_map_entry(), bootmem_t::next, NULL, OFFSET_OF, PAGE_MASK, PAGE_SIZE, and PFADDR_SHIFT.

Referenced by bootmem_init().

```
68                                                                                    {
69     bootmem_t *ptr, **dptr;
70     pfaddr_t addr, top;
71     pfaddr_t hole_addr, hole_top;
72
73     hole_addr = hole_start >> PFADDR_SHIFT;
74     hole_top  = hole_end   >> PFADDR_SHIFT;
75
```

```
    /* align on page boundaries */
    if( OFFSET_OF(hole_start, PAGE_SIZE) != 0 ) {
        hole_addr = (hole_addr & (e820_addr_t)~(PAGE_MASK >> PFADDR_SHIFT));
    }

    if( OFFSET_OF(hole_end, PAGE_SIZE) != 0 ) {
        hole_top = (hole_top & (e820_addr_t)~(PAGE_MASK >> PFADDR_SHIFT)) + (PAGE_SIZE >> PFADDR_SHIFT);
    }

    /* apply hole to all available memory blocks */
    for(dptr = head, ptr = *head; ptr != NULL; dptr = &ptr->next, ptr = ptr->next) {
        addr  = ptr->addr;
        top   = addr + ptr->count * (PAGE_SIZE >> PFADDR_SHIFT);

        /* case where the block is completely inside the hole */
        if(addr >= hole_addr && top <= hole_top) {
            /* remove this block */
            *dptr = ptr->next;

            return;
        }

        /* case where the block must be split in two because the hole is
         * inside it */
         if(addr < hole_addr && top > hole_top) {
             /* first block: below the hole */
             ptr->count = (hole_addr - addr) / (PAGE_SIZE >> PFADDR_SHIFT);

             /* second block: above the hole */
             new_ram_map_entry(hole_top, (top - hole_top) / (PAGE_SIZE >> PFADDR_SHIFT), head);

             return;
         }

         /* fix size or addr if block overlaps hole */
         if(addr >= hole_addr && addr < hole_top) {
             ptr->addr = hole_top;
             ptr->count = (top - hole_top) / (PAGE_SIZE >> PFADDR_SHIFT);

             return;
         }

         if(top > hole_addr && top <= hole_top) {
             ptr->count = (hole_addr - addr) / (PAGE_SIZE >> PFADDR_SHIFT);
         }
     }
}
```

Here is the call graph for this function:



4.89.1.2 bootmem_t* bootmem_get_block (void)

Definition at line 244 of file bootmem.c.

References bootmem_root, bootmem_t::next, and NULL.

Referenced by dispatch_syscall().

```
244                                    {
245     bootmem_t *block;
246 
247     block = bootmem_root;
248 
249     if(block != NULL) {
250         bootmem_root = block->next;
```

```
    }

    return block;
}
```

4.89.1.3 void bootmem_init (bool *use_pae*)

TODO check for available regions overlap

TODO this won't work for available memory > 4GB

Definition at line 124 of file bootmem.c.

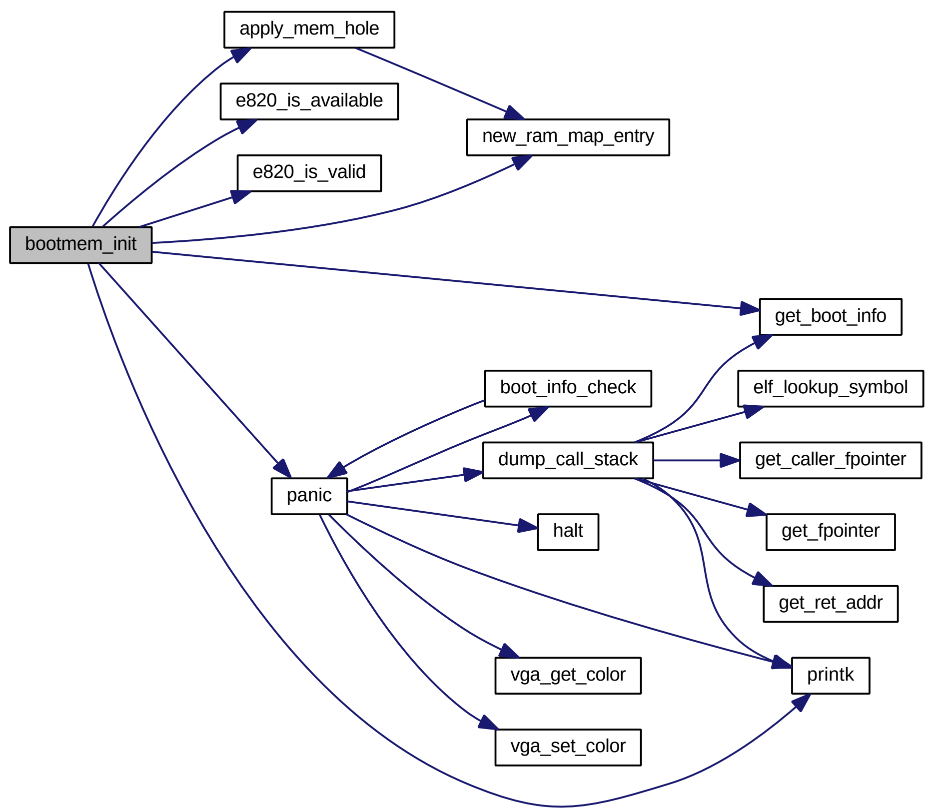
References bootmem_t::addr, e820_t::addr, ADDR_4GB, ADDR_TO_PFADDR, apply_mem_hole(), boot_heap, boot_info, bootmem_t::count, boot_info_t::e820_entries, e820_is_available(), e820_is_valid(), boot_info_t::e820_map, get_boot_info(), boot_info_t::image_start, KB, kernel_region_top, new_ram_map_entry(), bootmem_t::next, NULL, OFFSET_OF, PAGE_MASK, PAGE_SIZE, panic(), printk(), ram_map, and e820_t::size.

Referenced by vm_boot_init().

```
124                                   {
125     const addr_t initial_boot_heap = boot_heap;
126 
127     bootmem_t *ptr;
128     bootmem_t *temp_root;
129     unsigned int idx;
130 
131     const boot_info_t *boot_info = get_boot_info();
132 
135     /* copy the available ram entries from the e820 map and insert them
136      * in a linked list */
137     ram_map = NULL;
138 
139     for(idx = 0; idx < boot_info->e820_entries; ++idx) {
140         const e820_t *e820_entry = &boot_info->e820_map[idx];
141 
142         if(! e820_is_valid(e820_entry)) {
143             continue;
144         }
145 
146         if( e820_is_available(e820_entry) ) {
147             /* get memory entry start and end addresses */
148             e820_addr_t start = e820_entry->addr;
149             e820_addr_t end   = start + e820_entry->size;
150 
151             /* align on page boundaries */
152             if( OFFSET_OF(start, PAGE_SIZE) != 0 ) {
153                 start = (start & (e820_addr_t)~PAGE_MASK) + PAGE_SIZE;
154             }
155 
156             if( OFFSET_OF(end, PAGE_SIZE) != 0 ) {
157                 end = (end & (e820_addr_t)~PAGE_MASK);
158             }
159 
160             /* If Physical Address Extension (PAE) is disabled, memory above the
161              * 4GB mark is not usable. */
162             if(! use_pae) {
163                 /* If this memory region is completely above the 4GB mark, exclude it. */
164                 if(start >= ADDR_4GB) {
165                     continue;
166                 }
167 
168                 /* If this memory region starts below the 4GB mark but extends
169                  * beyond it, crop at 4GB. */
170                 if(end > ADDR_4GB) {
171                     end = ADDR_4GB;
172                 }
173             }
174 
175             /* add entry to linked list */
176             if(end > start) {
177                 new_ram_map_entry(ADDR_TO_PFADDR(start), (uint32_t)(end - start) /
      PAGE_SIZE, &ram_map);
178             }
```

```
        }
    }

    /* apply every unavailable entries from the e820 map as holes */
    for(idx = 0; idx < boot_info->e820_entries; ++idx) {
        const e820_t *e820_entry = &boot_info->e820_map[idx];

        if(! e820_is_valid(e820_entry)) {
            continue;
        }

        if( e820_is_available(e820_entry) ) {
            continue;
        }

        e820_addr_t start = e820_entry->addr;
        e820_addr_t end   = start + e820_entry->size;

        apply_mem_hole(start, end, &ram_map);
    }

    /* Apparently, the first 64k of memory are corrupted by some BIOSes.
         * It would be nice to try to detect this. In the meantime, let's
         * assume the problem is present. */
    apply_mem_hole(0, 0x10000, &ram_map);

    /* the kernel image, its heap and stack, and early-allocated pages */
    apply_mem_hole((uint32_t)boot_info->image_start, (uint32_t)kernel_region_top, &
  ram_map);

    /* Entry removal may have left garbage on the heap (bootmem_t
     * structures which were allocated on the heap but are no longer
     * linked). Let's clean up. */
    temp_root = NULL;

    for(ptr = ram_map; ptr != NULL; ptr = ptr->next) {
        new_ram_map_entry(ptr->addr, ptr->count, &temp_root);
    }

    ram_map   = NULL;
    boot_heap = initial_boot_heap;

    for(ptr = temp_root; ptr != NULL; ptr = ptr->next) {
        new_ram_map_entry(ptr->addr, ptr->count, &ram_map);
    }

    /* at this point, we should have at least one block of available RAM */
    if( ram_map == NULL ) {
        panic("no available memory.");
    }

    /* Let's count and display the total amount of available memory */
    uint32_t page_count = 0;
    for(ptr = ram_map; ptr != NULL; ptr = ptr->next) {
        page_count += ptr->count;
    }

    printk("%u kilobytes (%u pages) of memory available.\n",
        (uint32_t)(page_count * PAGE_SIZE / KB),
        (uint32_t)(page_count) );

    /* head pointer for bootmem_get_block() */
    bootmem_root = ram_map;
}
```

Here is the call graph for this function:



4.89.1.4 void new_ram_map_entry (pfaddr_t *addr,* uint32_t *count,* bootmem_t ∗∗ *head*)

Definition at line 55 of file bootmem.c.

References bootmem_t::addr, boot_heap, bootmem_t::count, and bootmem_t::next.

Referenced by apply_mem_hole(), and bootmem_init().

```
55                                                                              {
56      bootmem_t    *entry;
57
58      entry      = (bootmem_t *)boot_heap;
59      boot_heap = (bootmem_t *)boot_heap + 1;
60
61      entry->next  = *head;
62      entry->addr  = addr;
63      entry->count = count;
64
65      *head = entry;
66 }
```

4.89.2 Variable Documentation

4.89.2.1 void∗ boot_heap

current top of boot heap

Definition at line 52 of file bootmem.c.

Referenced by bootmem_init(), hal_init(), and new_ram_map_entry().

4.89.2.2 bootmem_t∗ bootmem_root

available memory map (allocator)

Definition at line 49 of file bootmem.c.

Referenced by bootmem_get_block(), and dispatch_syscall().

4.89.2.3 bootmem_t∗ ram_map

kernel memory map

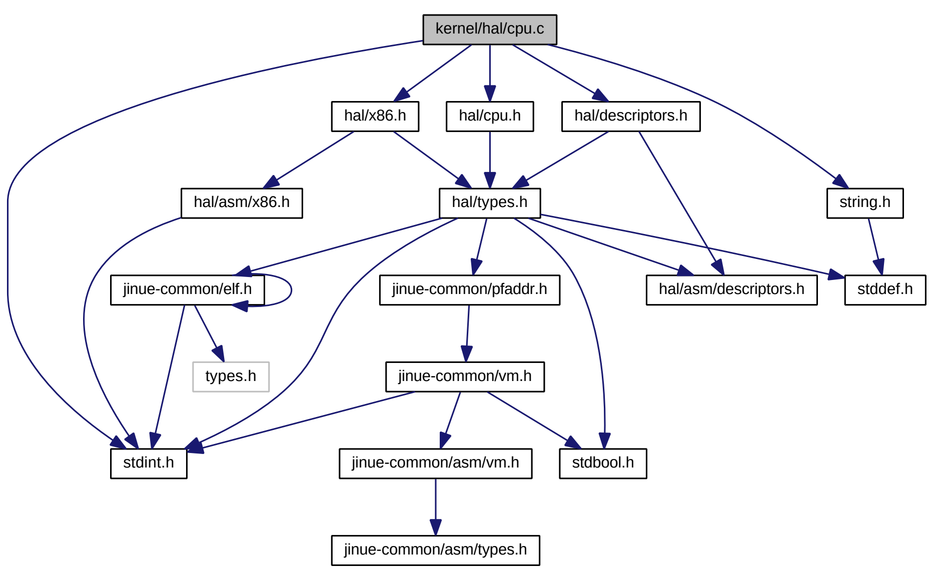
Definition at line 46 of file bootmem.c.

Referenced by bootmem_init().

4.90 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, **addr_t** kernel_stack)
- void **cpu_detect_features** (void)

Variables

- **cpu_info_t cpu_info**

4.90.1 Function Documentation

4.90.1.1 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.

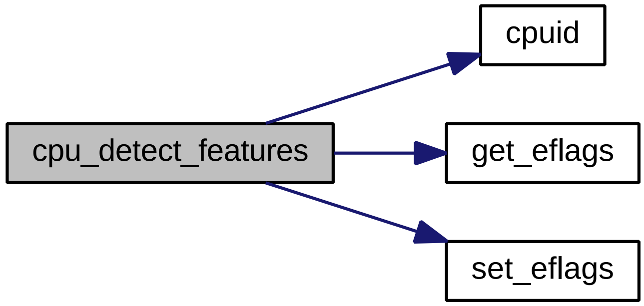
Referenced by hal_init().

```
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
132             && vendor_dw2 == CPU_VENDOR_AMD_DW2) {
133 
134             cpu_info.vendor = CPU_VENDOR_AMD;
135         }
136         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;
            }
        }
    }
```

```
215 }
```

Here is the call graph for this function:



4.90.1.2 void cpu_init_data (cpu_data_t * *data*, addr_t *kernel_stack*)

Definition at line 42 of file cpu.c.

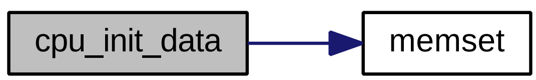
References cpu_data_t::current_addr_space, tss_t::esp0, tss_t::esp1, tss_t::esp2, cpu_data_t::gdt, GDT_KERNEL_-CODE, GDT_KERNEL_DATA, GDT_NULL, GDT_PER_CPU_DATA, GDT_TSS, GDT_USER_CODE, GDT_USER_-DATA, GDT_USER_TLS_DATA, memset(), NULL, RPL_KERNEL, SEG_DESCRIPTOR, SEG_FLAG_32BIT, SEG_-FLAG_IN_BYTES, 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] =
69          SEG_DESCRIPTOR( tss,    TSS_LIMIT-1,            SEG_TYPE_TSS   |
        SEG_FLAG_KERNEL | SEG_FLAG_TSS);
70
71      data->gdt[GDT_PER_CPU_DATA] =
72          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:



4.90.2 Variable Documentation

4.90.2.1 cpu_info_t cpu_info

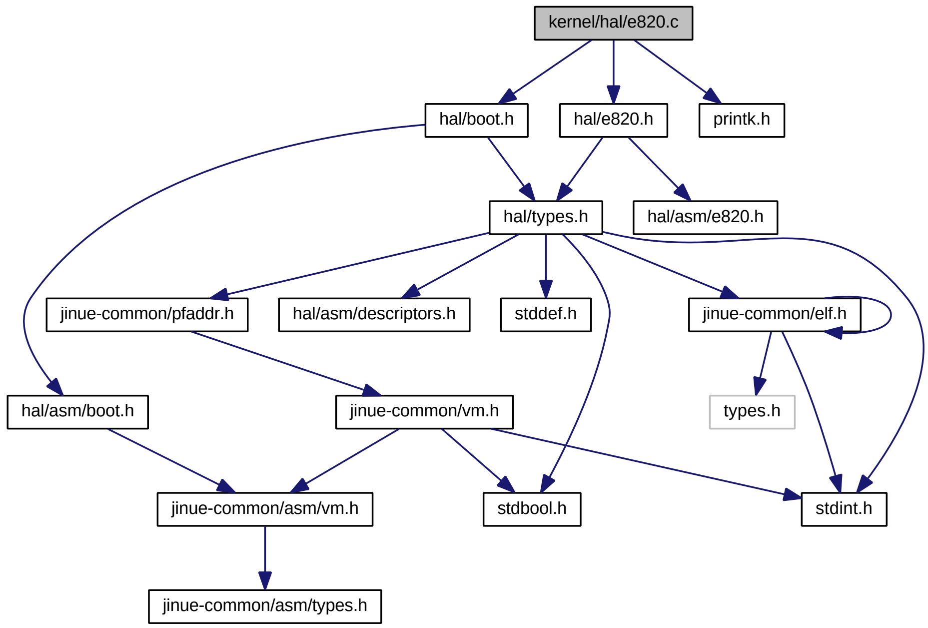
Definition at line 39 of file cpu.c.

Referenced by slab_cache_create().

4.91 kernel/hal/e820.c File Reference

```
#include <hal/boot.h>
#include <hal/e820.h>
#include <printk.h>
```

Include dependency graph for e820.c:



Functions

- **bool e820_is_valid** (const **e820_t** ∗e820_entry)
- **bool e820_is_available** (const **e820_t** ∗e820_entry)
- const char ∗ **e820_type_description** (**e820_type_t** type)
- void **e820_dump** (void)

4.91.1 Function Documentation

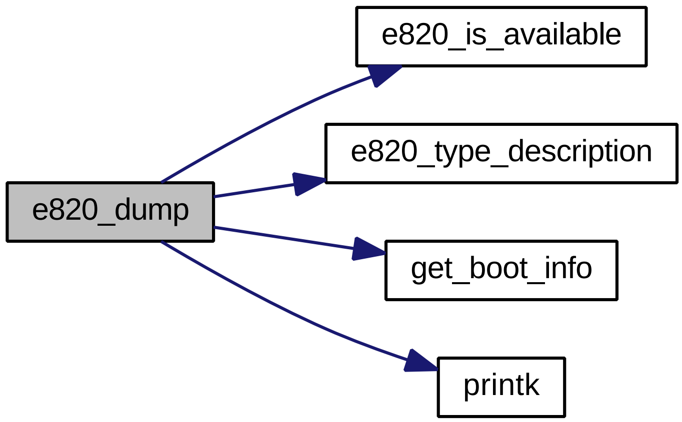
4.91.1.1 void e820_dump (void)

Definition at line 61 of file e820.c.

References e820_t::addr, boot_info, boot_info_t::e820_entries, e820_is_available(), boot_info_t::e820_map, e820_type_description(), get_boot_info(), printk(), e820_t::size, and e820_t::type.

```
61                        {
62     unsigned int idx;
63 
64     printk("Dump of the BIOS memory map:\n");
65 
66     const boot_info_t *boot_info = get_boot_info();
67 
68     for(idx = 0; idx < boot_info->e820_entries; ++idx) {
69         const e820_t *e820_entry = &boot_info->e820_map[idx];
70 
71         printk("%c [%q-%q] %s\n",
72             e820_is_available(e820_entry)?'*':' ',
73             e820_entry->addr,
74             e820_entry->addr + e820_entry->size - 1,
75             e820_type_description(e820_entry->type)
76         );
77     }
78 }
```

Here is the call graph for this function:



4.91.1.2 bool e820_is_available (const e820_t * *e820_entry*)

Definition at line 40 of file e820.c.

References E820_RAM, and e820_t::type.

Referenced by bootmem_init(), and e820_dump().

```
40                                               {
41     return e820_entry->type == E820_RAM;
42 }
```

4.91.1.3 bool e820_is_valid (const e820_t * *e820_entry*)

Definition at line 36 of file e820.c.

References e820_t::size.

Referenced by bootmem_init().

```
36                                           {
37     return e820_entry->size != 0;
38 }
```

4.91.1.4 const char* e820_type_description (e820_type_t *type*)

Definition at line 44 of file e820.c.

References E820_ACPI, E820_RAM, and E820_RESERVED.

Referenced by e820_dump().

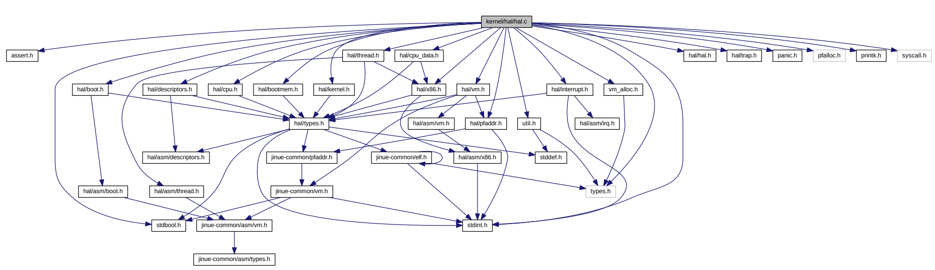
```
44                                                  {
45     switch(type) {
46
47     case E820_RAM:
48         return "available";
49
```

```
    case E820_RESERVED:
        return "unavailable/reserved";

    case E820_ACPI:
        return "unavailable/acpi";

    default:
        return "unavailable/other";
    }
}
```

4.92 kernel/hal/hal.c File Reference

```
#include <assert.h>
#include <hal/boot.h>
#include <hal/bootmem.h>
#include <hal/cpu.h>
#include <hal/cpu_data.h>
#include <hal/descriptors.h>
#include <hal/hal.h>
#include <hal/interrupt.h>
#include <hal/kernel.h>
#include <hal/pfaddr.h>
#include <hal/thread.h>
#include <hal/trap.h>
#include <hal/vm.h>
#include <hal/x86.h>
#include <panic.h>
#include <pfalloc.h>
#include <printk.h>
#include <stdbool.h>
#include <stdint.h>
#include <syscall.h>
#include <types.h>
#include <util.h>
#include <vm_alloc.h>
```

Include dependency graph for hal.c:



Functions

- void **hal_init** (void)

Variables

- **addr_t kernel_region_top**

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

  Specifies the entry point to use for system calls.

4.92.1 Function Documentation

4.92.1.1 void hal_init (void)

ASSERTION: we assume the image starts on a page boundary

ASSERTION: we assume the kernel starts on a page boundary

Definition at line 64 of file hal.c.

References pseudo_descriptor_t::addr, ALIGN_END, assert, boot_info_t::boot_end, boot_heap, boot_info_t::boot_heap, boot_info, boot_info_check(), CPU_DATA_ALIGNMENT, cpu_detect_features(), CPU_FEATURE_SYSCALL, CPU_FEATURE_SYSENTER, cpu_init_data(), EARLY_PTR_TO_PFADDR, fast_amd_entry(), fast_intel_entry(), GATE_DESCRIPTOR, cpu_data_t::gdt, GDT_KERNEL_CODE, GDT_KERNEL_DATA, GDT_LENGTH, GDT_PER_CPU_DATA, GDT_TSS, GDT_USER_CODE, get_boot_info(), global_pfcache, idt, IDT_VECTOR_COUNT, boot_info_t::image_start, init_pfcache(), KERNEL_PAGE_STACK_INIT, kernel_region_top, boot_info_t::kernel_size, boot_info_t::kernel_start, lgdt(), lidt(), pseudo_descriptor_t::limit, ltr(), MSR_EFER, MSR_FLAG_STAR_SCE, MSR_IA32_SYSENTER_CS, MSR_IA32_SYSENTER_EIP, MSR_IA32_SYSENTER_ESP, MSR_STAR, NULL, page_offset_of, PAGE_SIZE, pfalloc_early(), pffree, printk(), rdmsr(), RPL_KERNEL, RPL_USER, SEG_FLAG_KERNEL, SEG_FLAG_NORMAL_GATE, SEG_FLAG_USER, SEG_SELECTOR, SEG_TYPE_INTERRUPT_GATE, set_cs(), set_data_segments(), set_gs(), set_ss(), SYSCALL_IRQ, syscall_method, SYSCALL_METHOD_FAST_AMD, SYSCALL_METHOD_FAST_INTEL, SYSCALL_METHOD_INTR, use_pfalloc_early, vm_boot_init(), and wrmsr().

Referenced by kmain().

```
64                    {
65     addr_t addr;
66     addr_t               stack;
67     cpu_data_t          *cpu_data;
68     pseudo_descriptor_t *pseudo;
69     unsigned int         idx;
70     unsigned int         flags;
71     uint64_t             msrval;
72     pfaddr_t            *page_stack_buffer;
73     addr_t               boot_heap_old;
74
75     /* pfalloc() should not be called yet -- use pfalloc_early() instead */
76     use_pfalloc_early = true;
77
78     (void)boot_info_check(true);
79
80     const boot_info_t *boot_info = get_boot_info();
81
83     assert(page_offset_of(boot_info->image_start) == 0);
84
86     assert(page_offset_of(boot_info->kernel_start) == 0);
87
88     printk("Kernel size is %u bytes.\n", boot_info->kernel_size);
89
90     /* This must be done before any boot heap allocation. */
91     boot_heap = boot_info->boot_heap;
92
93     /* This must be done before any call to pfalloc_early(). */
94     kernel_region_top = boot_info->boot_end;
95
96     /* get cpu info */
97     cpu_detect_features();
98
99     /* allocate new kernel stack */
```

```
    stack = pfalloc_early();
    stack += PAGE_SIZE;

    /* 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);

    boot_heap = ALIGN_END(boot_heap, CPU_DATA_ALIGNMENT);

    cpu_data  = boot_heap;
    boot_heap = cpu_data + 1;

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

    /* allocate pseudo-descriptor for GDT and IDT (temporary allocation) */
    boot_heap_old = boot_heap;

    boot_heap = ALIGN_END(boot_heap, sizeof(pseudo_descriptor_t));

    pseudo    = (pseudo_descriptor_t *)boot_heap;
    boot_heap = (pseudo_descriptor_t *)boot_heap + 1;

    /* load new GDT and TSS */
    pseudo->addr    = (addr_t)&cpu_data->gdt;
    pseudo->limit   = GDT_LENGTH * 8 - 1;

    lgdt(pseudo);

    set_cs( SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL) );
    set_ss( SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL) );
    set_data_segments( SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL) );
    set_gs( SEG_SELECTOR(GDT_PER_CPU_DATA, RPL_KERNEL) );

    ltr( SEG_SELECTOR(GDT_TSS, RPL_KERNEL) );

    /* initialize IDT */
    for(idx = 0; idx < IDT_VECTOR_COUNT; ++idx) {
        /* get address, which is already stored in the IDT entry */
        addr  = (addr_t)(uintptr_t)idt[idx];

        /* set interrupt gate flags */
        flags = SEG_TYPE_INTERRUPT_GATE | SEG_FLAG_NORMAL_GATE;

        if(idx == SYSCALL_IRQ) {
            flags |= SEG_FLAG_USER;
        }
        else {
            flags |= SEG_FLAG_KERNEL;
        }

        /* create interrupt gate descriptor */
        idt[idx] = GATE_DESCRIPTOR(
            SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL),
            addr,
            flags,
            NULL );
    }

    pseudo->addr  = (addr_t)idt;
    pseudo->limit = IDT_VECTOR_COUNT * sizeof(seg_descriptor_t) - 1;
    lidt(pseudo);

    /* de-allocate pseudo-descriptor */
    boot_heap = boot_heap_old;

    /* initialize the page frame allocator */
    page_stack_buffer = (pfaddr_t *)pfalloc_early();
    init_pfcache(&global_pfcache, page_stack_buffer);

    for(idx = 0; idx < KERNEL_PAGE_STACK_INIT; ++idx) {
        pffree( EARLY_PTR_TO_PFADDR( pfalloc_early() ) );
    }

    /* initialize virtual memory management, enable paging
     *
     * below this point, it is no longer safe to call pfalloc_early() */
    vm_boot_init();

```

```
    /* choose system call method */
    syscall_method = SYSCALL_METHOD_INTR;

    if(cpu_has_feature(CPU_FEATURE_SYSENTER)) {
        syscall_method = SYSCALL_METHOD_FAST_INTEL;

        wrmsr(MSR_IA32_SYSENTER_CS,  SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL));
        wrmsr(MSR_IA32_SYSENTER_EIP, (uint64_t)(uintptr_t)fast_intel_entry);

        /* kernel stack address is set when switching thread context */
        wrmsr(MSR_IA32_SYSENTER_ESP, (uint64_t)(uintptr_t)NULL);
    }

    if(cpu_has_feature(CPU_FEATURE_SYSCALL)) {
        syscall_method = SYSCALL_METHOD_FAST_AMD;

        msrval  = rdmsr(MSR_EFER);
        msrval |= MSR_FLAG_STAR_SCE;
        wrmsr(MSR_EFER, msrval);

        msrval  = (uint64_t)(uintptr_t)fast_amd_entry;
        msrval |= (uint64_t)SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL)    << 32;
        msrval |= (uint64_t)SEG_SELECTOR(GDT_USER_CODE,   RPL_USER)      << 48;

        wrmsr(MSR_STAR, msrval);
    }
}
```

Here is the call graph for this function:

4.92.2 Variable Documentation

4.92.2.1 addr_t kernel_region_top

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

Definition at line 59 of file hal.c.

Referenced by bootmem_init(), hal_init(), pfalloc_early(), and vm_boot_init().

4.92.2.2 int syscall_method

Specifies the entry point to use for system calls.

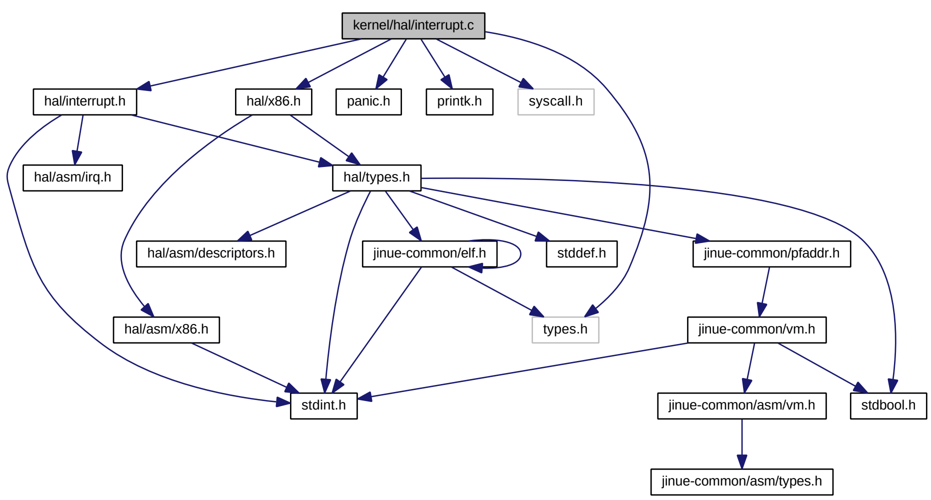
Definition at line 62 of file hal.c.

Referenced by dispatch_syscall(), and hal_init().

4.93 kernel/hal/interrupt.c File Reference

```
#include <hal/interrupt.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)

4.93.1 Function Documentation

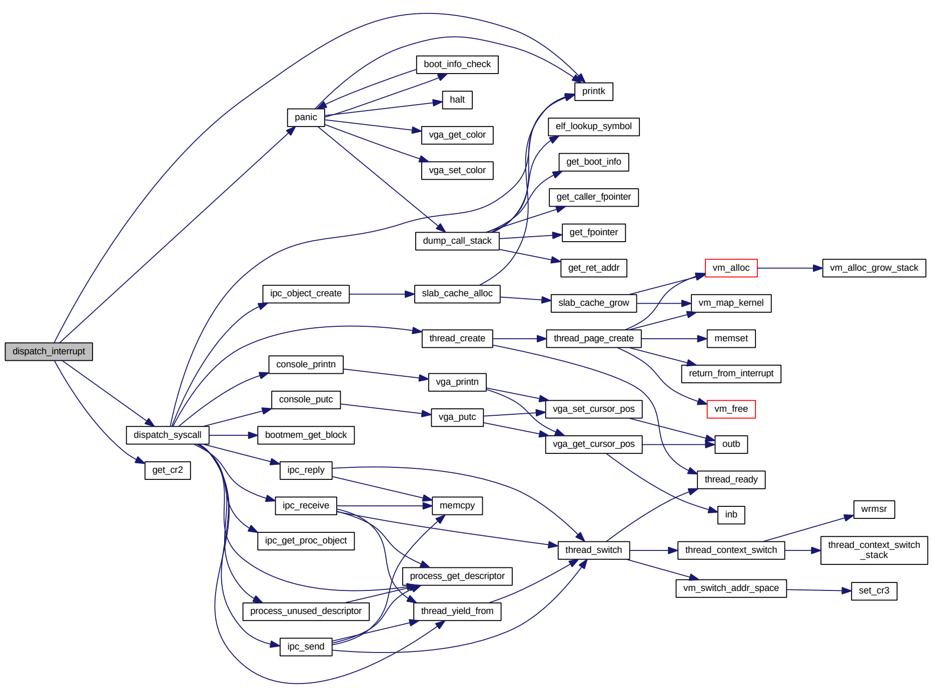
4.93.1.1 void dispatch_interrupt (**trapframe_t** * *trapframe*)

Definition at line 40 of file interrupt.c.

References dispatch_syscall(), trapframe_t::eip, trapframe_t::errcode, get_cr2(), IDT_FIRST_IRQ, trapframe_t::ivt, panic(), printk(), and SYSCALL_IRQ.

```
40                                                  {
41      unsigned int    ivt         = trapframe->ivt;
42      uintptr_t       eip         = trapframe->eip;
43      uint32_t        errcode     = trapframe->errcode;
44
45      /* exceptions */
46      if(ivt < IDT_FIRST_IRQ) {
47          printk("EXCEPT: %u cr2=0x%x errcode=0x%x eip=0x%x\n", ivt, get_cr2(), errcode, eip);
48
49          /* never returns */
50          panic("caught exception");
51      }
52
53      /* slow system call method */
54      if(ivt == SYSCALL_IRQ) {
55          dispatch_syscall(trapframe);
56      }
57      else {
58          printk("INTR: ivt %u (vector %u)\n", ivt - IDT_FIRST_IRQ, ivt);
59      }
60  }
```

Here is the call graph for this function:

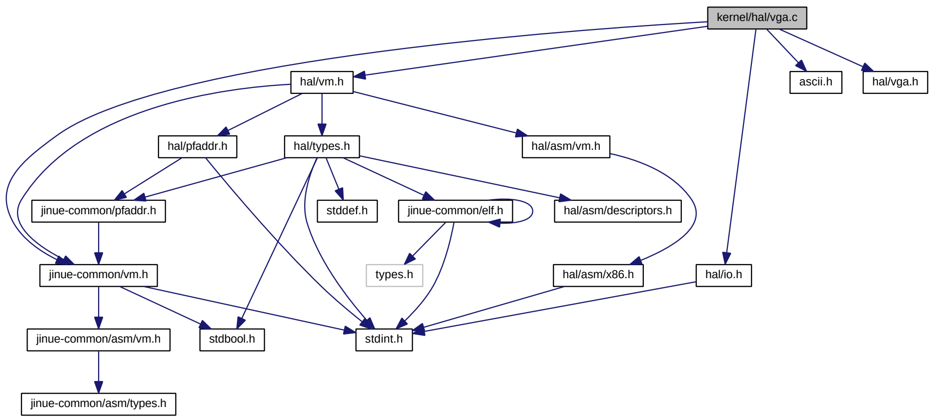


4.94 kernel/hal/vga.c File Reference

`#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)
- unsigned int **vga_get_color** (void)
- void **vga_set_color** (unsigned int color)
- **vga_pos_t vga_get_cursor_pos** (void)
- void **vga_set_cursor_pos** (**vga_pos_t** pos)
- void **vga_print** (const char ∗message)
- void **vga_printn** (const char ∗message, unsigned int n)
- void **vga_putc** (char c)

4.94.1 Function Documentation

4.94.1.1 void vga_clear (void)

Definition at line 71 of file vga.c.

References VGA_COLOR_ERASE, VGA_LINES, and VGA_WIDTH.

Referenced by vga_init().

```
71                        {
72     unsigned int idx = 0;
73 
74     while( idx < (VGA_LINES * VGA_WIDTH * 2) ) {
75         video_base_addr[idx++] = 0x20;
76         video_base_addr[idx++] = VGA_COLOR_ERASE;
77     }
78 }
```

4.94.1.2 unsigned int vga_get_color (void)

Definition at line 95 of file vga.c.

Referenced by panic().

```
95                                {
96     return vga_text_color;
97 }
```

4.94.1.3 vga_pos_t vga_get_cursor_pos (void)

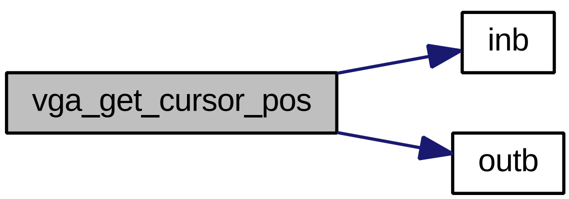
Definition at line 103 of file vga.c.

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

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

```
103                                    {
104     unsigned char h, l;
105 
106     outb(VGA_CRTC_ADDR, 0x0e);
107     h = inb(VGA_CRTC_DATA);
108     outb(VGA_CRTC_ADDR, 0x0f);
109     l = inb(VGA_CRTC_DATA);
110 
111     return (h << 8) | l;
112 }
```

Here is the call graph for this function:



4.94.1.4 void vga_init (void)

Definition at line 46 of file vga.c.

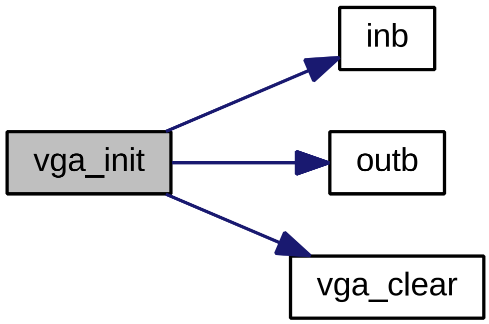
References inb(), outb(), vga_clear(), VGA_COLOR_DEFAULT, VGA_CRTC_ADDR, VGA_CRTC_DATA, VGA_MISC-_OUT_RD, and VGA_MISC_OUT_WR.

Referenced by console_init().

```
46                     {
47     unsigned char data;
48 
49     /* set text color to default */
50     vga_text_color = VGA_COLOR_DEFAULT;
51 
52     /* Set address select bit in a known state: CRTC regs at 0x3dx */
53     data = inb(VGA_MISC_OUT_RD);
54     data |= 1;
55     outb(VGA_MISC_OUT_WR, data);
56 
57     /* Move cursor to line 0 col 0 */
```

```
58      outb(VGA_CRTC_ADDR, 0x0e);
59      outb(VGA_CRTC_DATA, 0x0);
60      outb(VGA_CRTC_ADDR, 0x0f);
61      outb(VGA_CRTC_DATA, 0x0);
62
63      /* Clear the screen */
64      vga_clear();
65 }
```

Here is the call graph for this function:



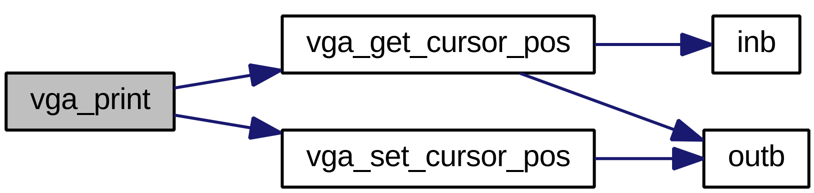
4.94.1.5 void vga_print (const char ∗ *message*)

Definition at line 125 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

```
125                                      {
126     unsigned short int pos = vga_get_cursor_pos();
127     char c;
128
129     while( (c = *(message++)) ) {
130         pos = vga_raw_putc(c, pos);
131     }
132
133     vga_set_cursor_pos(pos);
134 }
```

Here is the call graph for this function:



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

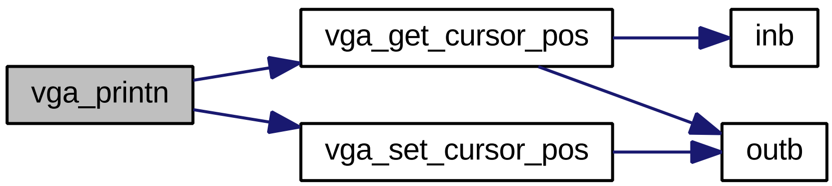
Definition at line 136 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by console_printn().

```
136                                                  {
137     vga_pos_t pos = vga_get_cursor_pos();
138     char c;
139
140     while(n) {
141         c = *(message++);
142         pos = vga_raw_putc(c, pos);
143         --n;
144     }
145
146     vga_set_cursor_pos(pos);
147 }
```

Here is the call graph for this function:



4.94.1.7 void vga_putc (char *c*)

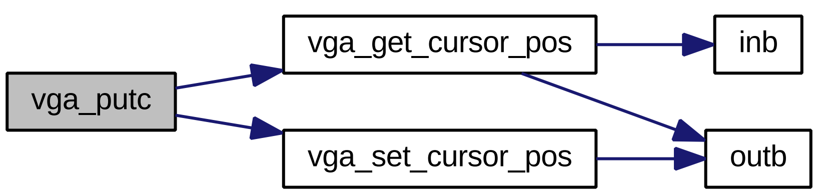
Definition at line 149 of file vga.c.

References vga_get_cursor_pos(), and vga_set_cursor_pos().

Referenced by console_putc().

```
149                         {
150     vga_pos_t pos = vga_get_cursor_pos();
151
152     pos = vga_raw_putc(c, pos);
153
154     vga_set_cursor_pos(pos);
155 }
```

Here is the call graph for this function:



4.94.1.8 void vga_scroll (void)

Definition at line 80 of file vga.c.

References VGA_COLOR_ERASE, VGA_LINES, and VGA_WIDTH.

```
80                        {
81     unsigned char *di = video_base_addr;
82     unsigned char *si = video_base_addr + 2 * VGA_WIDTH;
```

```
83     unsigned int idx;
84
85     for(idx = 0; idx < 2 * VGA_WIDTH * (VGA_LINES - 1); ++idx) {
86         *(di++) = *(si++);
87     }
88
89     for(idx = 0; idx < VGA_WIDTH; ++idx) {
90         *(di++) = 0x20;
91         *(di++) = VGA_COLOR_ERASE;
92     }
93 }
```

4.94.1.9 void vga_set_base_addr (void * *base_addr*)

Definition at line 67 of file vga.c.

References vm_block_t::base_addr.

Referenced by vm_boot_init().

```
67                                         {
68     video_base_addr = base_addr;
69 }
```

4.94.1.10 void vga_set_color (unsigned int *color*)

Definition at line 99 of file vga.c.

Referenced by panic().

```
99                                          {
100     vga_text_color = color;
101 }
```

4.94.1.11 void vga_set_cursor_pos (vga_pos_t *pos*)

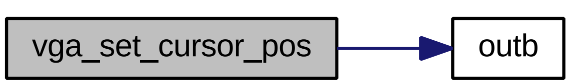
Definition at line 114 of file vga.c.

References outb(), VGA_CRTC_ADDR, and VGA_CRTC_DATA.

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

```
114                                         {
115     unsigned char h = pos >> 8;
116     unsigned char l = pos;
117
118     outb(VGA_CRTC_ADDR, 0x0e);
119     outb(VGA_CRTC_DATA, h);
120     outb(VGA_CRTC_ADDR, 0x0f);
121     outb(VGA_CRTC_DATA, l);
122 }
```

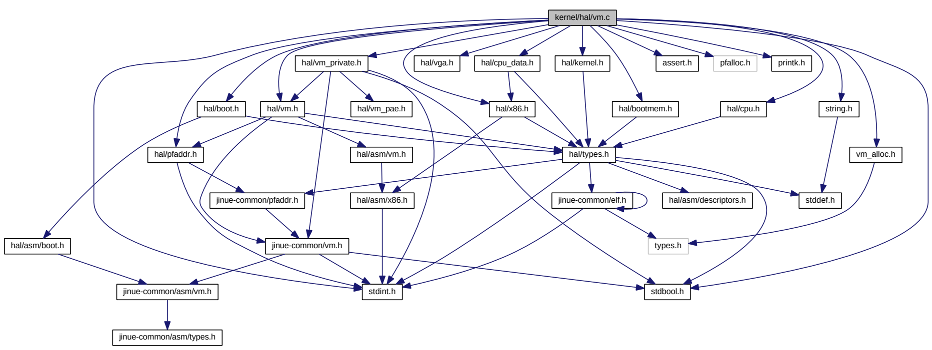
Here is the call graph for this function:



4.95 kernel/hal/vm.c File Reference

```
#include <hal/boot.h>
#include <hal/bootmem.h>
#include <hal/cpu.h>
#include <hal/cpu_data.h>
#include <hal/kernel.h>
#include <hal/pfaddr.h>
#include <hal/vga.h>
#include <hal/vm.h>
#include <hal/vm_private.h>
#include <hal/x86.h>
#include <assert.h>
#include <pfalloc.h>
#include <printk.h>
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include <vm_alloc.h>
```

Include dependency graph for vm.c:



Data Structures

- struct **pte_t**

Functions

- void **vm_boot_init** (void)
- void **vm_unmap** (**addr_space_t** ∗addr_space, **addr_t** addr)

  Unmap a page from virtual memory.
- void **vm_map_kernel** (**addr_t** vaddr, **pfaddr_t** paddr, int flags)
- void **vm_map_user** (**addr_space_t** ∗addr_space, **addr_t** vaddr, **pfaddr_t** paddr, int flags)
- void **vm_unmap_kernel** (**addr_t** addr)
- void **vm_unmap_user** (**addr_space_t** ∗addr_space, **addr_t** addr)
- **pfaddr_t vm_lookup_pfaddr** (**addr_space_t** ∗addr_space, **addr_t** addr)
- void **vm_change_flags** (**addr_space_t** ∗addr_space, **addr_t** addr, int flags)
- void **vm_map_early** (**addr_t** vaddr, **pfaddr_t** paddr, int flags)
- **pfaddr_t vm_clone_page_directory** (**pfaddr_t** template_pfaddr, unsigned int start_index)

- **addr_space_t** ∗ **vm_create_addr_space** (**addr_space_t** ∗addr_space)
- **pte_t** ∗ **vm_allocate_page_directory** (unsigned int start_index, **bool** first_pd)
- **addr_space_t** ∗ **vm_x86_create_initial_addr_space** (void)
- **addr_space_t** ∗ **vm_create_initial_addr_space** (void)
- void **vm_destroy_page_directory** (**pfaddr_t** pdpfaddr, 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)

Variables

- **pte_t** ∗ **global_page_tables**
- **addr_space_t initial_addr_space**
- **vm_alloc_t** ∗ **global_page_allocator**

  global page allocator (region 0..KLIMIT)
- **size_t page_table_entries** = (**size_t**)**PAGE_TABLE_ENTRIES**
- **addr_space_t** ∗(∗ **create_addr_space**)(**addr_space_t** ∗) = vm_x86_create_addr_space
- **addr_space_t** ∗(∗ **create_initial_addr_space**)(void) = **vm_x86_create_initial_addr_space**
- void(∗ **destroy_addr_space**)(**addr_space_t** ∗) = vm_x86_destroy_addr_space
- unsigned int(∗ **page_table_offset_of**)(**addr_t**) = vm_x86_page_table_offset_of

  page table entry offset of virtual (linear) address
- unsigned int(∗ **page_directory_offset_of**)(**addr_t**) = vm_x86_page_directory_offset_of
- **pte_t** ∗(∗ **lookup_page_directory**)(**addr_space_t** ∗, void ∗, **bool**) = vm_x86_lookup_page_directory
- **pte_t** ∗(∗ **get_pte_with_offset**)(**pte_t** ∗, unsigned int) = vm_x86_get_pte_with_offset
- void(∗ **set_pte**)(**pte_t** ∗, **pfaddr_t**, int) = vm_x86_set_pte
- void(∗ **set_pte_flags**)(**pte_t** ∗, int) = vm_x86_set_pte_flags
- int(∗ **get_pte_flags**)(**pte_t** ∗) = vm_x86_get_pte_flags
- **pfaddr_t**(∗ **get_pte_pfaddr**)(**pte_t** ∗) = vm_x86_get_pte_pfaddr
- void(∗ **clear_pte**)(**pte_t** ∗) = vm_x86_clear_pte
- void(∗ **copy_pte**)(**pte_t** ∗, **pte_t** ∗) = vm_x86_copy_pte

4.95.1 Function Documentation

4.95.1.1 pte_t∗ vm_allocate_page_directory (unsigned int *start_index*, bool *first_pd*)

Definition at line 524 of file vm.c.

References clear_pte, EARLY_PTR_TO_PFADDR, get_pte_with_offset, page_table_entries, pfalloc_early(), set_pte, and VM_FLAG_READ_WRITE.

Referenced by vm_x86_create_initial_addr_space().

```
524                                                                         {
525     unsigned int idx, idy;
526     pte_t *page_directory;
527     pte_t *page_table;
528
529     /* Allocate page directory. */
530     page_directory = (pte_t *)pfalloc_early();
531
532     /* clear user space page directory entries */
533     for(idx = 0; idx < start_index; ++idx) {
534         clear_pte( get_pte_with_offset(page_directory, idx) );
535     }
536
537     /* allocate page tables for kernel data/code region (above KLIMIT) */
538     for(idx = start_index; idx < page_table_entries; ++idx) {
```

```
        /* allocate the page table
         *
         * Note that the use of pfalloc_early() here guarantees that the
         * page table are allocated contiguously, and that they keep the
         * same address once paging is enabled. */
        page_table = (pte_t *)pfalloc_early();

        if(first_pd && 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_PFADDR(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) );
        }
    }

    return page_directory;
}
```

Here is the call graph for this function:

4.95.1.2 void vm_boot_init (void)

below this point, it is no longer safe to call **pfalloc_early()** (p. 211)

Definition at line 87 of file vm.c.

References ADDR_TO_PFADDR, boot_info, bootmem_init(), CPU_FEATURE_PAE, EARLY_PTR_TO_PFADDR, get_boot_info(), boot_info_t::image_start, kernel_region_top, KLIMIT, MB, PAGE_SIZE, printk(), use_pfalloc_early, vga_set_base_addr(), VGA_TEXT_VID_BASE, VGA_TEXT_VID_TOP, vm_alloc_add_region(), vm_alloc_init_allocator(), vm_create_initial_addr_space(), VM_FLAG_KERNEL, VM_FLAG_READ_WRITE, vm_map_early(), vm_pae_boot_init(), vm_pae_create_pdpt_cache(), vm_pae_enable(), and vm_switch_addr_space().

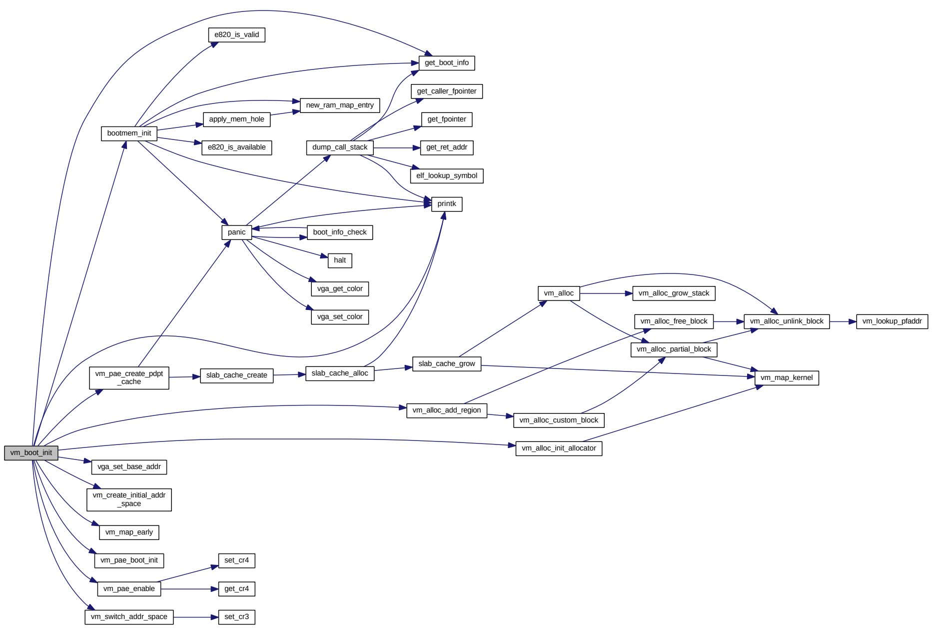
Referenced by hal_init().

```
87                          {
88      bool             use_pae;
89      addr_t           addr;
90      addr_space_t    *addr_space;
91
92      if(cpu_has_feature(CPU_FEATURE_PAE)) {
93          printk("Enabling Physical Address Extension (PAE).\n");
94          vm_pae_boot_init();
95
96          use_pae = true;
97      }
98      else {
99          use_pae = false;
100     }
101
102     /* create initial address space */
103     addr_space = vm_create_initial_addr_space();
104
106     use_pfalloc_early = false;
107
108     /* create system usable physical memory (RAM) map
```

```
     *
     * Among other things, this function marks the memory used by the kernel
     * (i.e. image_start..kernel_region_top) as in use. This must be done after
     * all early page frame allocations with fpalloc_early() have been done.
     *
     * This function needs to know whether Physical Address Extension (PAE) is
     * enabled (use_pae) because, if it isn't, all memory above the 4GB mark is
     * excluded from the usable memory map. */
    bootmem_init(use_pae);

    /* perform 1:1 mapping of kernel image and data

       note: page tables for memory region (0..KLIMIT) are contiguous in
       physical memory */
    const boot_info_t *boot_info = get_boot_info();

    for(addr = (addr_t)boot_info->image_start; addr < kernel_region_top; addr += PAGE_SIZE) {
        vm_map_early((addr_t)addr, EARLY_PTR_TO_PFADDR(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 kernel_vm_top = kernel_region_top;
    addr = (addr_t)VGA_TEXT_VID_BASE;

    addr_t vga_text_base = kernel_vm_top;

    while(addr < (addr_t)VGA_TEXT_VID_TOP) {
        vm_map_early(kernel_vm_top, ADDR_TO_PFADDR((uintptr_t)addr), VM_FLAG_KERNEL | VM_FLAG_READ_WRITE);
        kernel_vm_top   += PAGE_SIZE;
        addr            += 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", kernel_vm_top);

    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.
         *
         * From Intel 64 and IA-32 Architectures Software Developer's Manual
         * Volume 3: System Programming Guide, section 4.4.1 "PDPTE Registers":
         *
         *  " The logical processor loads [the PDPTE] registers from the PDPTEs
         *    in memory as part of certain operations:
         *      * If PAE paging would be in use following an execution of MOV to
         *        CR0 or MOV to CR4 (see Section 4.1.1) and the instruction is
         *        modifying any of (...) CR4.PAE, (...); then the PDPTEs are
         *        loaded from the address in CR3. "
         *
         * There are bootstrapping issues when enabling PAE while paging is enabled.
         * See the comment at the top of the vm_pae_create_initial_addr_space()
         * function in vm_pae.c for more detail. */
        vm_pae_enable();
    }

    /* switch to new address space */
    vm_switch_addr_space(addr_space);

```

```
    /* initialize global page allocator (region starting at KLIMIT)
     *
     * TODO Some work needs to be done in the page allocator to support allocating
     * up to the top of memory (i.e. 0x100000000, which cannot be represented on
     * 32 bits). In the mean time, we leave a 4MB gap. */
    global_page_allocator = &__global_page_allocator;
    vm_alloc_init_allocator(global_page_allocator, (addr_t)KLIMIT, (addr_t)0 - 4 * MB);

    vm_alloc_add_region(global_page_allocator, (addr_t)KLIMIT,          (addr_t)boot_info->image_start);
    vm_alloc_add_region(global_page_allocator, (addr_t)kernel_vm_top,   (addr_t)0 - 4 * MB);

    /* 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();
    }
}
```

Here is the call graph for this function:



4.95.1.3 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

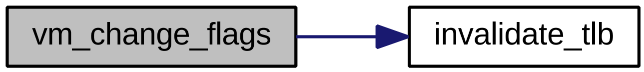
Definition at line 444 of file vm.c.

References assert, get_pte_flags, invalidate_tlb(), NULL, and set_pte_flags.

```
                                                                         {
    pte_t *pte = vm_lookup_page_table_entry(addr_space, addr, false);

    assert(pte != NULL && (get_pte_flags(pte) & VM_FLAG_PRESENT));

```

```
    /* perform the flags change */
    set_pte_flags(pte, flags | VM_FLAG_PRESENT);

    vm_free_page_table_entry(addr, pte);

    /* invalidate TLB entry for the affected page */
    invalidate_tlb(addr);
}
```

Here is the call graph for this function:



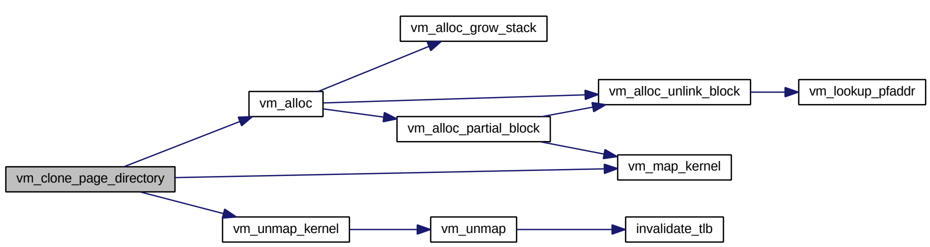
4.95.1.4 pfaddr_t vm_clone_page_directory (pfaddr_t *template_pfaddr,* unsigned int *start_index*)

Definition at line 472 of file vm.c.

References clear_pte, copy_pte, get_pte_with_offset, page_table_entries, pfalloc, vm_alloc(), VM_FLAG_READ_WRITE, vm_map_kernel(), and vm_unmap_kernel().

```
472                                                                              {
473     unsigned int idx;
474     pfaddr_t pfaddr;
475     pte_t *page_directory;
476     pte_t *template;
477 
478     /* allocate and map new page directory */
479     page_directory = (pte_t *)vm_alloc(global_page_allocator);
480     pfaddr = pfalloc();
481     vm_map_kernel((addr_t)page_directory, pfaddr, VM_FLAG_READ_WRITE);
482 
483     /* map page directory template */
484     template = (pte_t *)vm_alloc(global_page_allocator);
485     vm_map_kernel((addr_t)template, template_pfaddr, VM_FLAG_READ_WRITE);
486 
487     /* clear all entries below index start_index */
488     for(idx = 0; idx < start_index; ++idx) {
489         clear_pte( get_pte_with_offset(page_directory, idx) );
490     }
491 
492     /* copy entries from template for indexes start_index and above */
493     for(idx = start_index; idx < page_table_entries; ++idx) {
494         copy_pte(
495             get_pte_with_offset(page_directory, idx),
496             get_pte_with_offset(template, idx)
497         );
498     }
499 
500     vm_unmap_kernel((addr_t)page_directory);
501     vm_unmap_kernel((addr_t)template);
502 
503     return pfaddr;
504 }
```

Here is the call graph for this function:



4.95.1.5 addr_space_t∗ vm_create_addr_space (addr_space_t ∗ *addr_space*)

Definition at line 520 of file vm.c.

References create_addr_space.

Referenced by process_create().

```
520                                                           {
521     return create_addr_space(addr_space);
522 }
```

4.95.1.6 addr_space_t∗ vm_create_initial_addr_space (void)

Definition at line 577 of file vm.c.

References create_initial_addr_space.

Referenced by vm_boot_init().

```
577                                               {
578     return create_initial_addr_space();
579 }
```

4.95.1.7 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 609 of file vm.c.

References assert, destroy_addr_space, and NULL.

```
609                                                   {
611     assert(addr_space != NULL);
612
614     assert(addr_space != &initial_addr_space);
615
617     assert( addr_space != get_current_addr_space() );
618
619     destroy_addr_space(addr_space);
620 }
```

4.95.1.8 void vm_destroy_page_directory (pfaddr_t *pdpfaddr,* unsigned int *from_index,* unsigned int *to_index*)

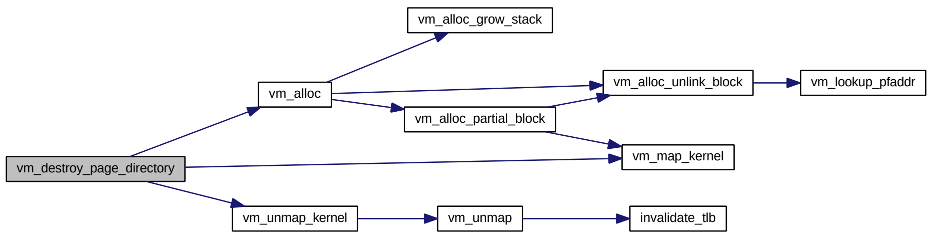
Definition at line 581 of file vm.c.

References get_pte_flags, get_pte_pfaddr, get_pte_with_offset, pffree, vm_alloc(), VM_FLAG_READ_WRITE, vm_-map_kernel(), and vm_unmap_kernel().

```
581                                                                                                   {
582     unsigned int idx;
583
584     pte_t *page_directory = (pte_t *)vm_alloc(global_page_allocator);
585     vm_map_kernel((addr_t)page_directory, pdpfaddr, VM_FLAG_READ_WRITE);
586
587     /* be careful not to free the kernel page tables */
588     for(idx = from_index; idx < to_index; ++idx) {
```

```
        pte_t *pte = get_pte_with_offset(page_directory, idx);

        if(get_pte_flags(pte) & VM_FLAG_PRESENT) {
            pffree( get_pte_pfaddr(pte) );
        }
    }

    vm_unmap_kernel((addr_t)page_directory);
    pffree(pdpfaddr);
}
```

Here is the call graph for this function:



4.95.1.9 pfaddr_t vm_lookup_pfaddr (addr_space_t * *addr_space,* addr_t *addr*)

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

Definition at line 431 of file vm.c.

References assert, get_pte_flags, get_pte_pfaddr, and NULL.

Referenced by thread_page_destroy(), vm_alloc_destroy(), and vm_alloc_unlink_block().

```
431                                                             {
432     pte_t *pte = vm_lookup_page_table_entry(addr_space, addr, false);
433
435     assert(pte != NULL && (get_pte_flags(pte) & VM_FLAG_PRESENT));
436
437     pfaddr_t pfaddr = get_pte_pfaddr(pte);
438
439     vm_free_page_table_entry(addr, pte);
440
441     return pfaddr;
442 }
```

4.95.1.10 void vm_map_early (addr_t *vaddr,* pfaddr_t *paddr,* int *flags*)

ASSERTION: we are mapping in the kernel region

ASSERTION: we assume vaddr is aligned on a page boundary

Definition at line 459 of file vm.c.

References assert, EARLY_VIRT_TO_PHYS, get_pte_with_offset, page_number_of, page_offset_of, and set_pte.

Referenced by vm_boot_init().

```
459                                                        {
460     pte_t *pte;
461
463     assert( is_fast_map_pointer(vaddr) );
464
466     assert( page_offset_of(vaddr) == 0 );
```

```
467
468     pte = get_pte_with_offset(global_page_tables, page_number_of(
      EARLY_VIRT_TO_PHYS((uintptr_t)vaddr) ));
469     set_pte(pte, paddr, flags | VM_FLAG_PRESENT);
470 }
```

4.95.1.11 void vm_map_kernel (addr_t *vaddr*, pfaddr_t *paddr*, int *flags*)

Definition at line 415 of file vm.c.

References NULL, and VM_FLAG_KERNEL.

Referenced by elf_load(), elf_setup_stack(), slab_cache_grow(), thread_page_create(), vm_alloc_init_allocator(), vm_-alloc_partial_block(), vm_clone_page_directory(), and vm_destroy_page_directory().

```
415                                                                    {
416     vm_map(NULL, vaddr, paddr, flags | VM_FLAG_KERNEL);
417 }
```

4.95.1.12 void vm_map_user (addr_space_t ∗ *addr_space*, addr_t *vaddr*, pfaddr_t *paddr*, int *flags*)

Definition at line 419 of file vm.c.

References VM_FLAG_USER.

Referenced by elf_load(), and elf_setup_stack().

```
419                                                                                          {
420     vm_map(addr_space, vaddr, paddr, flags | VM_FLAG_USER);
421 }
```

4.95.1.13 void vm_switch_addr_space (addr_space_t ∗ *addr_space*)

Definition at line 622 of file vm.c.

References addr_space_t::cr3, and set_cr3().

Referenced by thread_switch(), and vm_boot_init().

```
622                                                          {
623     set_cr3(addr_space->cr3);
624
625     get_cpu_local_data()->current_addr_space = addr_space;
626 }
```

Here is the call graph for this function:

4.95.1.14 void vm_unmap (addr_space_t ∗ *addr_space*, addr_t *addr*)

Unmap a page from virtual memory.

Parameters

addr_space	address space from which to unmap, can be NULL for global mappings (addr >= KLIMIT)
addr	address of page to unmap

ASSERTION: we assume addr is aligned on a page boundary

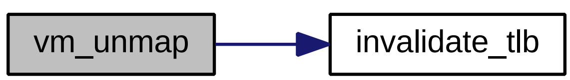
Definition at line 388 of file vm.c.

References assert, clear_pte, invalidate_tlb(), NULL, and page_offset_of.

Referenced by vm_unmap_kernel(), and vm_unmap_user().

```
388                                                     {
390     assert( page_offset_of(addr) == 0 );
391
392 #ifdef NDEBUG
393     /* Performance optimization: vm_unmap is a no-op for kernel mappings when
394      * compiling non-debug.
395      *
396      * When compiling in debug mode, the unmap operation is actually performed
397      * to help detect use-after-unmap bugs. */
398     if(is_kernel_pointer(addr)) {
399         return;
400     }
401 #endif
402
403     pte_t *pte = vm_lookup_page_table_entry(addr_space, addr, false);
404
405     if(pte != NULL) {
406         clear_pte(pte);
407
408         vm_free_page_table_entry(addr, pte);
409
410         /* invalidate TLB entry for newly mapped page */
411         invalidate_tlb(addr);
412     }
413 }
```

Here is the call graph for this function:



4.95.1.15 void vm_unmap_kernel (addr_t *addr*)

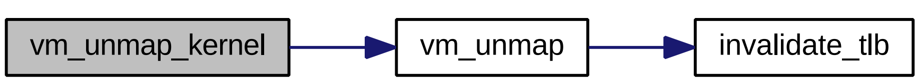
Definition at line 423 of file vm.c.

References NULL, and vm_unmap().

Referenced by elf_load(), elf_setup_stack(), thread_page_destroy(), vm_clone_page_directory(), and vm_destroy_page_directory().

```
423                                  {
424     vm_unmap(NULL, addr);
425 }
```

Here is the call graph for this function:



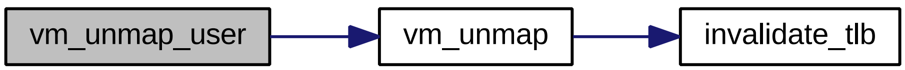
4.95.1.16 void vm_unmap_user (addr_space_t ∗ *addr_space,* addr_t *addr*)

Definition at line 427 of file vm.c.

References vm_unmap().

```
427                                                          {
428     vm_unmap(addr_space, addr);
429 }
```

Here is the call graph for this function:



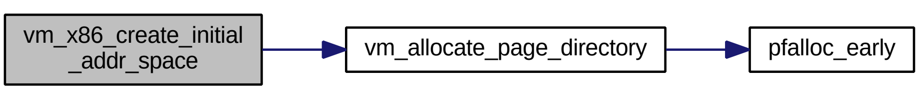
4.95.1.17 addr_space_t∗ vm_x86_create_initial_addr_space (void)

Definition at line 566 of file vm.c.

References addr_space_t::cr3, EARLY_PTR_TO_PFADDR, EARLY_VIRT_TO_PHYS, initial_addr_space, KLIMIT, page_directory_offset_of, addr_space_t::pd, addr_space_t::top_level, and vm_allocate_page_directory().

```
566                                                     {
567     unsigned int klimit_pd_index = page_directory_offset_of((addr_t)KLIMIT);
568 
569     pte_t *page_directory = vm_allocate_page_directory(klimit_pd_index, true);
570 
571     initial_addr_space.top_level.pd = EARLY_PTR_TO_PFADDR(page_directory);
572     initial_addr_space.cr3          = EARLY_VIRT_TO_PHYS((uintptr_t)page_directory);
573 
574     return &initial_addr_space;
575 }
```

Here is the call graph for this function:



4.95.2 Variable Documentation

4.95.2.1 void(∗ clear_pte)(pte_t ∗) = vm_x86_clear_pte

Definition at line 703 of file vm.c.

Referenced by vm_allocate_page_directory(), vm_clone_page_directory(), vm_pae_boot_init(), and vm_unmap().

4.95.2.2 void(∗ copy_pte)(pte_t ∗, pte_t ∗) = vm_x86_copy_pte

Definition at line 705 of file vm.c.

Referenced by vm_clone_page_directory(), and vm_pae_boot_init().

4.95.2.3 addr_space_t*(* create_addr_space)(addr_space_t *) = vm_x86_create_addr_space

Definition at line 680 of file vm.c.

Referenced by vm_create_addr_space(), and vm_pae_boot_init().

4.95.2.4 addr_space_t*(* create_initial_addr_space)(void) = vm_x86_create_initial_addr_space

Definition at line 682 of file vm.c.

Referenced by vm_create_initial_addr_space(), and vm_pae_boot_init().

4.95.2.5 void(* destroy_addr_space)(addr_space_t *) = vm_x86_destroy_addr_space

Definition at line 684 of file vm.c.

Referenced by vm_destroy_addr_space(), and vm_pae_boot_init().

4.95.2.6 int(* get_pte_flags)(pte_t *) = vm_x86_get_pte_flags

Definition at line 699 of file vm.c.

Referenced by vm_change_flags(), vm_destroy_page_directory(), vm_lookup_pfaddr(), and vm_pae_boot_init().

4.95.2.7 pfaddr_t(* get_pte_pfaddr)(pte_t *) = vm_x86_get_pte_pfaddr

Definition at line 701 of file vm.c.

Referenced by vm_destroy_page_directory(), vm_lookup_pfaddr(), and vm_pae_boot_init().

4.95.2.8 pte_t*(* get_pte_with_offset)(pte_t *, unsigned int) = vm_x86_get_pte_with_offset

Definition at line 693 of file vm.c.

Referenced by vm_allocate_page_directory(), vm_clone_page_directory(), vm_destroy_page_directory(), vm_map_-early(), and vm_pae_boot_init().

4.95.2.9 vm_alloc_t* global_page_allocator

global page allocator (region 0..KLIMIT)

Definition at line 58 of file vm.c.

Referenced by elf_load(), elf_setup_stack(), slab_cache_grow(), thread_page_create(), and thread_page_destroy().

4.95.2.10 pte_t* global_page_tables

Definition at line 51 of file vm.c.

4.95.2.11 addr_space_t initial_addr_space

Definition at line 53 of file vm.c.

Referenced by vm_x86_create_initial_addr_space().

4.95.2.12 pte_t∗(∗ lookup_page_directory)(addr_space_t ∗, void ∗, bool) = vm_x86_lookup_page_directory

Definition at line 691 of file vm.c.

Referenced by vm_pae_boot_init().

4.95.2.13 unsigned int(∗ page_directory_offset_of)(addr_t) = vm_x86_page_directory_offset_of

Definition at line 689 of file vm.c.

Referenced by vm_pae_boot_init(), and vm_x86_create_initial_addr_space().

4.95.2.14 size_t page_table_entries = (size_t)PAGE_TABLE_ENTRIES

Definition at line 678 of file vm.c.

Referenced by vm_allocate_page_directory(), vm_clone_page_directory(), and vm_pae_boot_init().

4.95.2.15 unsigned int(∗ page_table_offset_of)(addr_t) = vm_x86_page_table_offset_of

page table entry offset of virtual (linear) address

Definition at line 687 of file vm.c.

Referenced by vm_pae_boot_init().

4.95.2.16 void(∗ set_pte)(pte_t ∗, pfaddr_t, int) = vm_x86_set_pte

Definition at line 695 of file vm.c.

Referenced by vm_allocate_page_directory(), vm_map_early(), and vm_pae_boot_init().

4.95.2.17 void(∗ set_pte_flags)(pte_t ∗, int) = vm_x86_set_pte_flags

Definition at line 697 of file vm.c.

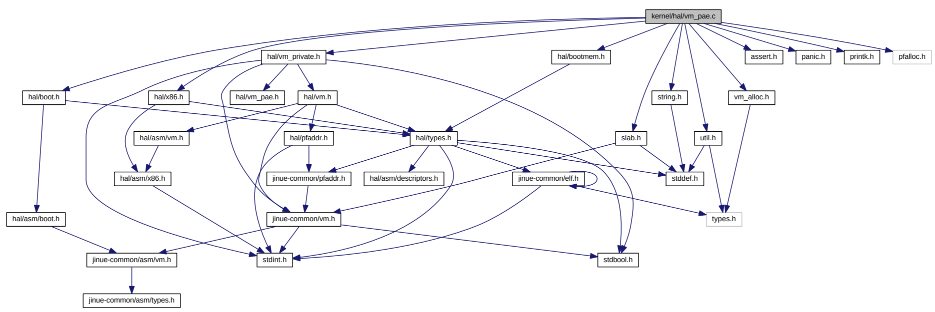
Referenced by vm_change_flags(), and vm_pae_boot_init().

4.96 kernel/hal/vm_pae.c File Reference

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

```
#include <hal/boot.h>
#include <hal/bootmem.h>
#include <hal/x86.h>
#include <assert.h>
#include <panic.h>
#include <printk.h>
#include <pfalloc.h>
#include <slab.h>
#include <string.h>
#include <util.h>
#include <vm_alloc.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_enable** (void)

    *This header file contains declarations for the PAE functions defined in **hal/vm_pae.c** (p. 342).*

- void **vm_pae_create_pdpt_cache** (void)
- void **vm_pae_boot_init** (void)

Variables

- **pdpt_t** $*$ **initial_pdpt**

4.96.1 Macro Definition Documentation

4.96.1.1 #define PDPT_BITS 2

number of address bits that encode the PDPT offset

Definition at line 47 of file vm_pae.c.

4.96.1.2 #define PDPT_ENTRIES (1 << PDPT_BITS)

number of entries in a Page Directory Pointer Table (PDPT)

Definition at line 50 of file vm_pae.c.

4.96.2 Function Documentation

4.96.2.1 void vm_pae_boot_init (void)

Definition at line 358 of file vm_pae.c.

References clear_pte, copy_pte, create_addr_space, create_initial_addr_space, destroy_addr_space, get_pte_flags, get_pte_pfaddr, get_pte_with_offset, lookup_page_directory, page_directory_offset_of, PAGE_TABLE_ENTRIES, page_table_entries, page_table_offset_of, set_pte, and set_pte_flags.

Referenced by vm_boot_init().

```
358                         {
359     page_table_entries          = (size_t)PAGE_TABLE_ENTRIES;
360     create_addr_space           = vm_pae_create_addr_space;
361     create_initial_addr_space   = vm_pae_create_initial_addr_space;
362     destroy_addr_space          = vm_pae_destroy_addr_space;
363     page_table_offset_of        = vm_pae_page_table_offset_of;
364     page_directory_offset_of    = vm_pae_page_directory_offset_of;
365     lookup_page_directory       = vm_pae_lookup_page_directory;
366     get_pte_with_offset         = vm_pae_get_pte_with_offset;
367     set_pte                     = vm_pae_set_pte;
368     set_pte_flags               = vm_pae_set_pte_flags;
369     get_pte_flags               = vm_pae_get_pte_flags;
370     get_pte_pfaddr              = vm_pae_get_pte_pfaddr;
371     clear_pte                   = vm_pae_clear_pte;
372     copy_pte                    = vm_pae_copy_pte;
373 }
```

4.96.2.2 void vm_pae_create_pdpt_cache (void)

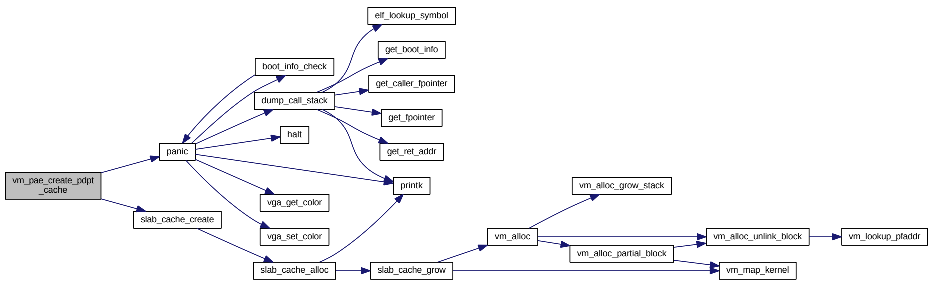
Definition at line 159 of file vm_pae.c.

References NULL, panic(), slab_cache_create(), and SLAB_DEFAULTS.

Referenced by vm_boot_init().

```
159                                 {
160     pdpt_cache = slab_cache_create(
161             "vm_pae_pdpt_cache",
162             sizeof(pdpt_t),
163             sizeof(pdpt_t),
164             NULL,
165             NULL,
166             SLAB_DEFAULTS);
167 
168     if(pdpt_cache == NULL) {
169         panic("Cannot create Page Directory Pointer Table (PDPT) slab cache.");
170     }
171 }
```

Here is the call graph for this function:



4.96.2.3 void vm_pae_enable (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. 330) and **hal/vm_pae.c** (p. 342). There should be no reason to include it anywhere else.

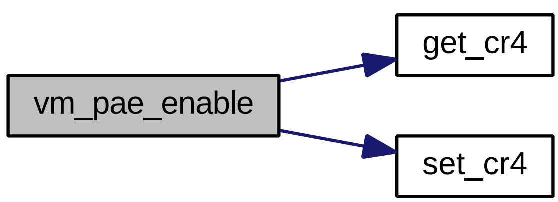
Definition at line 154 of file vm_pae.c.

References get_cr4(), set_cr4(), and X86_CR4_PAE.

Referenced by vm_boot_init().

```
154                            {
155     uint32_t temp = get_cr4();
156     set_cr4(temp | X86_CR4_PAE);
157 }
```

Here is the call graph for this function:



4.96.3 Variable Documentation

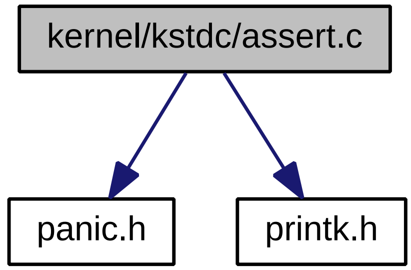
4.96.3.1 pdpt_t∗ initial_pdpt

Definition at line 64 of file vm_pae.c.

4.97 kernel/kstdc/assert.c File Reference

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

Include dependency graph for assert.c:



Functions

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

4.97.1 Function Documentation

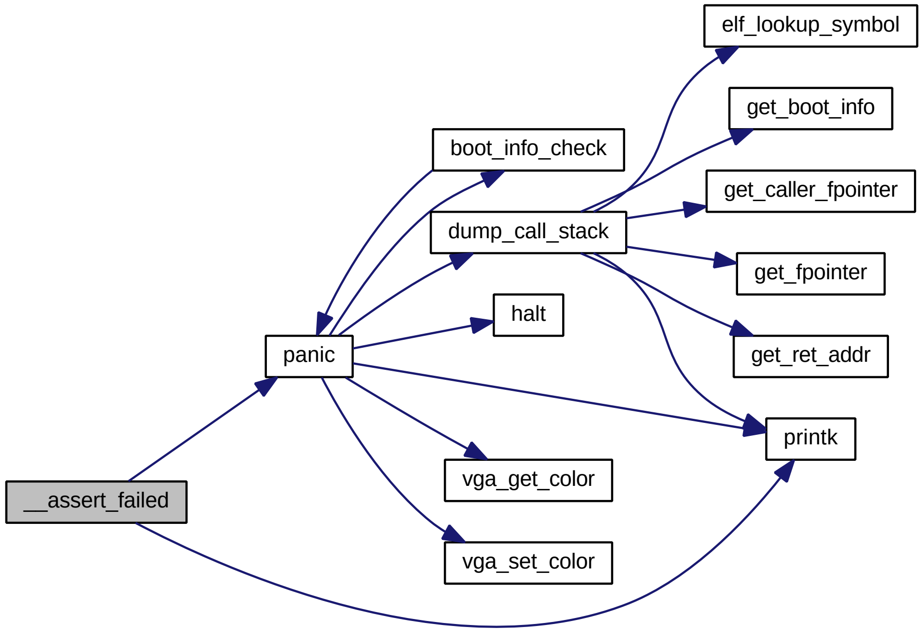
4.97.1.1 void __assert_failed (const char ∗ *expr,* const char ∗ *file,* unsigned int *line,* const char ∗ *func*)

Definition at line 36 of file assert.c.

References panic(), and printk().

```
                         {

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

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

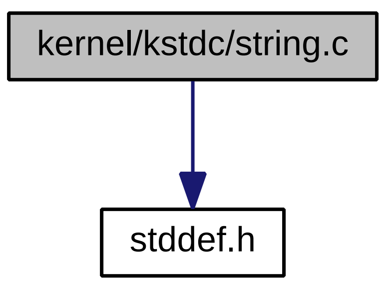
Here is the call graph for this function:



4.98 kernel/kstdc/string.c File Reference

`#include <stddef.h>`

Include dependency graph for 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)

4.98.1 Function Documentation

4.98.1.1 void∗ memcpy (void ∗ *dest,* const void ∗ *src,* size_t *n*)

Definition at line 45 of file string.c.

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

```
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 }
```

4.98.1.2 void∗ memset (void ∗ *s*, int *c*, size_t *n*)

Definition at line 34 of file string.c.

Referenced by cpu_init_data(), process_create(), and thread_page_create().

```
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 }
```

4.98.1.3 size_t strlen (const char ∗ *s*)

Definition at line 57 of file string.c.

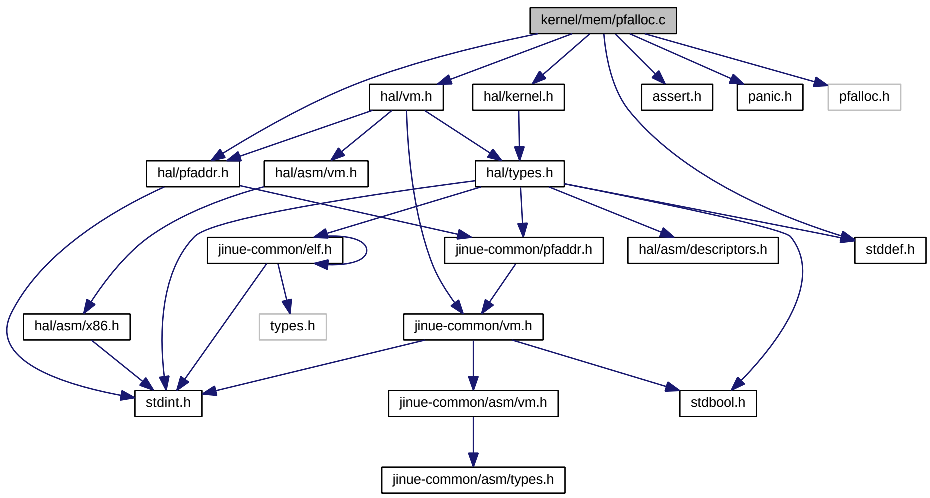
Referenced by console_print().

```
57                              {
58     size_t count = 0;
59
60     while(*s != 0) {
61         ++s;
62         ++count;
63     }
64
65     return count;
66 }
```

4.99 kernel/mem/pfalloc.c File Reference

```
#include <hal/kernel.h>
#include <hal/pfaddr.h>
#include <hal/vm.h>
#include <assert.h>
#include <panic.h>
#include <pfalloc.h>
#include <stddef.h>
```

Include dependency graph for pfalloc.c:



Functions

- **addr_t pfalloc_early** (void)
- void **init_pfcache** (**pfcache_t** ∗pfcache, **pfaddr_t** ∗stack_page)
- **pfaddr_t pfalloc_from** (**pfcache_t** ∗pfcache)
- void **pffree_to** (**pfcache_t** ∗pfcache, **pfaddr_t** pf)

Variables

- **bool use_pfalloc_early**
- **pfcache_t global_pfcache**

4.99.1 Function Documentation

4.99.1.1 void init_pfcache (pfcache_t ∗ *pfcache*, pfaddr_t ∗ *stack_page*)

Definition at line 58 of file pfalloc.c.

References pfcache_t::count, KERNEL_PAGE_STACK_SIZE, PFNULL, and pfcache_t::ptr.

Referenced by hal_init().

```
58                                                          {
59     pfaddr_t *ptr;
60     unsigned int idx;
61 
62     ptr = stack_page;
63 
64     for(idx = 0;idx < KERNEL_PAGE_STACK_SIZE; ++idx) {
65         ptr[idx] = PFNULL;
66     }
67 
68     pfcache->ptr   = stack_page;
69     pfcache->count = 0;
70 }
```

4.99.1.2 addr_t pfalloc_early (void)

ASSERTION: pfalloc_early is used early only

Definition at line 46 of file pfalloc.c.

References assert, kernel_region_top, PAGE_SIZE, and use_pfalloc_early.

Referenced by hal_init(), and vm_allocate_page_directory().

```
46                          {
47     addr_t page;
48
50     assert(use_pfalloc_early);
51
52     page = kernel_region_top;
53     kernel_region_top += PAGE_SIZE;
54
55     return page;
56 }
```

4.99.1.3 pfaddr_t pfalloc_from (pfcache_t ∗ *pfcache*)

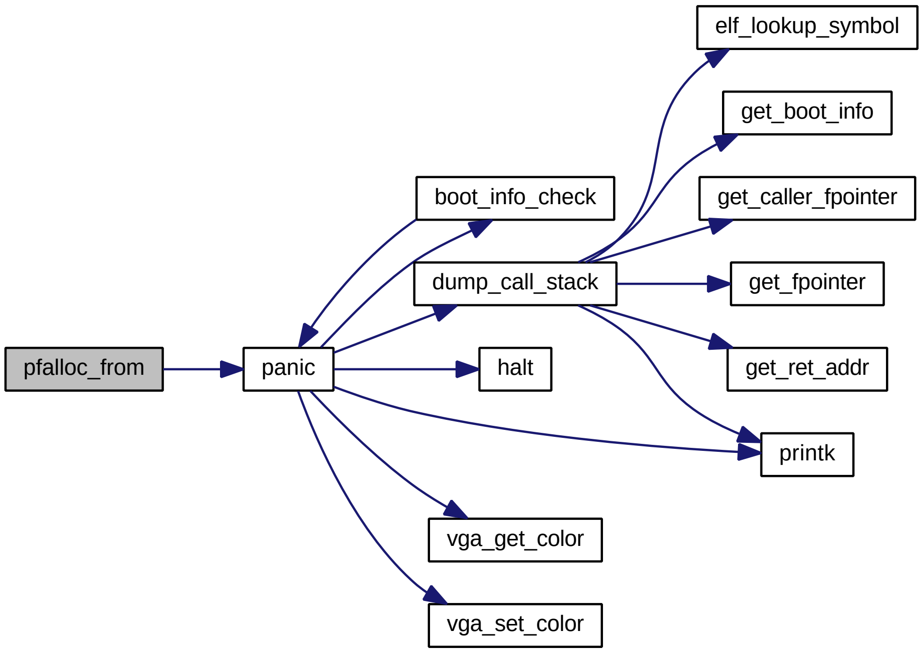
ASSERTION: pfalloc_early must be used early

Definition at line 72 of file pfalloc.c.

References assert, pfcache_t::count, panic(), pfcache_t::ptr, and use_pfalloc_early.

```
72                                       {
74     assert( ! use_pfalloc_early );
75
76     if(pfcache->count == 0) {
77         panic("pfalloc_from(): no more pages to allocate");
78     }
79
80     --pfcache->count;
81
82     return *(--pfcache->ptr);
83 }
```

Here is the call graph for this function:



4.99.1.4 void pffree_to (**pfcache_t** ∗ *pfcache,* **pfaddr_t** *pf*)

We are leaking memory here. Should we panic instead?

Definition at line 85 of file pfalloc.c.

References pfcache_t::count, KERNEL_PAGE_STACK_SIZE, and pfcache_t::ptr.

```
85                                                {
86     if(pfcache->count >= KERNEL_PAGE_STACK_SIZE) {
88         return;
89     }
90 
91     ++pfcache->count;
92 
93     (pfcache->ptr++)[0] = pf;
94 }
```

4.99.2 Variable Documentation

4.99.2.1 pfcache_t global_pfcache

Definition at line 43 of file pfalloc.c.

Referenced by hal_init().

4.99.2.2 bool use_pfalloc_early

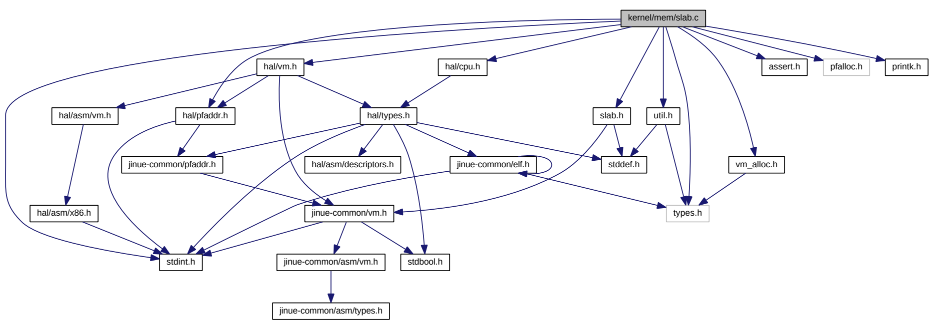
Definition at line 41 of file pfalloc.c.

Referenced by hal_init(), pfalloc_early(), pfalloc_from(), and vm_boot_init().

4.100 kernel/mem/slab.c File Reference

```
#include <hal/cpu.h>
#include <hal/pfaddr.h>
#include <hal/vm.h>
#include <assert.h>
#include <pfalloc.h>
#include <printk.h>
#include <slab.h>
#include <stdint.h>
#include <types.h>
#include <util.h>
#include <vm_alloc.h>
```

Include dependency graph for slab.c:



Functions

- **slab_cache_t** ∗ **slab_cache_create** (char ∗name, **size_t** size, **size_t** alignment, **slab_ctor_t** ctor, **slab_ctor_t** dtor, int flags)
- void **slab_cache_destroy** (**slab_cache_t** ∗cache)
- void ∗ **slab_cache_alloc** (**slab_cache_t** ∗cache)
- void **slab_cache_free** (void ∗buffer)
- void **slab_cache_grow** (**slab_cache_t** ∗cache)
- void **slab_cache_reap** (**slab_cache_t** ∗cache)
- void **slab_cache_set_working_set** (**slab_cache_t** ∗cache, unsigned int n)

Variables

- **slab_cache_t** ∗ **slab_cache_list** = &slab_cache_cache

4.100.1 Function Documentation

4.100.1.1 void∗ slab_cache_alloc (slab_cache_t ∗ *cache*)

ASSERTION: now that **slab_cache_grow()** (p. 246) 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 228 of file slab.c.

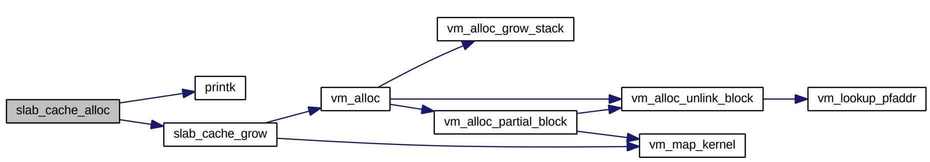
References assert, slab_cache_t::bufctl_offset, slab_cache_t::ctor, slab_cache_t::empty_count, slab_cache_t::flags, slab_t::free_list, slab_cache_t::name, slab_bufctl_t::next, slab_t::next, slab_t::obj_count, slab_cache_t::obj_size, slab_t::prev, printk(), slab_cache_grow(), SLAB_POISON, SLAB_POISON_ALIVE_VALUE, 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(), and slab_cache_create().

```
228                                             {
229     slab_t          *slab;
230     slab_bufctl_t   *bufctl;
231     uint32_t        *buffer;
232     unsigned int     idx;
233     unsigned int     dump_lines;
234 
235     if(cache->slabs_partial != NULL) {
236         slab = cache->slabs_partial;
237     }
238     else {
239         if(cache->slabs_empty == NULL) {
240             slab_cache_grow(cache);
241         }
242 
243         slab = cache->slabs_empty;
244 
246         assert(slab != NULL);
247 
257         /* We are about to allocate one object from this slab, so it will
258          *  not be empty anymore...*/
259         cache->slabs_empty      = slab->next;
260 
261         --(cache->empty_count);
262 
263         slab->next              = cache->slabs_partial;
264         if(slab->next != NULL) {
265             slab->next->prev = slab;
266         }
267         cache->slabs_partial    = slab;
268     }
269 
270     bufctl = slab->free_list;
271 
273     assert(bufctl != NULL);
274 
275     slab->free_list  = bufctl->next;
276     slab->obj_count += 1;
277 
278     /* If we just allocated the last buffer, move the slab to the full
279      * list */
280     if(slab->free_list == NULL) {
281         /* remove from the partial slabs list */
282 
284         assert(cache->slabs_partial == slab);
285 
286         cache->slabs_partial = slab->next;
287 
288         if(slab->next != NULL) {
289             slab->next->prev = slab->prev;
290         }
291 
292         /* add to the full slabs list */
293         slab->next        = cache->slabs_full;
294         cache->slabs_full = slab;
295 
296         if(slab->next != NULL) {
297             slab->next->prev = slab;
298         }
299     }
```

```c

    buffer = (uint32_t *)( (char *)bufctl - cache->bufctl_offset );

    if(cache->flags & SLAB_POISON) {
        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;
        }

        if(cache->ctor != NULL) {
            cache->ctor((void *)buffer, cache->obj_size);
        }
    }
    else if(cache->flags & SLAB_RED_ZONE) {
        buffer[cache->obj_size / sizeof(uint32_t)] = SLAB_RED_ZONE_VALUE;
    }

    return (void *)buffer;
}
```

Here is the call graph for this function:



4.100.1.2 slab_cache_t* slab_cache_create (char * *name,* size_t *size,* size_t *alignment,* slab_ctor_t *ctor,* slab_ctor_t *dtor,* int *flags*)

ASSERTION: ensure buffer size is at least the size of a pointer

ASSERTION: name is not NULL string

Definition at line 89 of file slab.c.

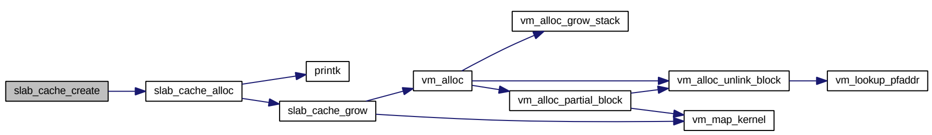
References slab_cache_t::alignment, slab_cache_t::alloc_size, assert, slab_cache_t::bufctl_offset, cpu_info, slab_cache_t::ctor, cpu_info_t::dcache_alignment, slab_cache_t::dtor, slab_cache_t::empty_count, slab_cache_t::flags, slab_cache_t::max_colour, slab_cache_t::name, slab_cache_t::next, slab_cache_t::next_colour, NULL, slab_cache_t-::obj_size, slab_cache_t::prev, slab_cache_alloc(), slab_cache_list, 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().

```
                             {

    slab_cache_t    *cache;
    size_t           avail_space;
    size_t           wasted_space;
    unsigned int     buffers_per_slab;

    assert( size >= sizeof(void *) );

    assert(name != NULL);

    cache = slab_cache_alloc(&slab_cache_cache);

    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;

    /* add new cache to cache list */
    cache->next             = slab_cache_list;
    slab_cache_list         = cache;

    if(cache->next != NULL) {
        cache->next->prev = cache;
    }

    /* 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;
    }

    if(cache->alignment % sizeof(uint32_t) != 0) {
        cache->alignment += sizeof(uint32_t) - cache->alignment % sizeof(
  uint32_t);
    }

    /* reserve space for bufctl and/or redzone word */
    cache->obj_size = size;

    if(cache->obj_size % sizeof(uint32_t) != 0) {
        cache->obj_size += sizeof(uint32_t) - cache->obj_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 and/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);
```

```
166     }
167     else {
168         cache->alloc_size = cache->obj_size;
169     }
170
171     if(cache->alloc_size % cache->alignment != 0) {
172         cache->alloc_size += cache->alignment - cache->alloc_size % cache->
      alignment;
173     }
174
175     avail_space = SLAB_SIZE - sizeof(slab_t);
176
177     buffers_per_slab = avail_space / cache->alloc_size;
178
179     wasted_space = avail_space - buffers_per_slab * cache->alloc_size;
180
181     cache->max_colour = (wasted_space / cache->alignment) * cache->alignment;
182
183     cache->bufctl_offset = cache->alloc_size - sizeof(slab_bufctl_t);
184
185     return cache;
186 }
```

Here is the call graph for this function:



4.100.1.3 void slab_cache_destroy (slab_cache_t * *cache*)

ASSERTION: all memory has been returned to the cache

ASSERTION: empty slabs count is accurate

Definition at line 188 of file slab.c.

References assert, slab_cache_t::empty_count, slab_cache_t::next, slab_t::next, slab_cache_t::prev, slab_cache_free(), slab_cache_t::slabs_empty, slab_cache_t::slabs_full, and slab_cache_t::slabs_partial.

```
188                                            {
189     slab_t       *slab;
190     slab_t       *next;
191     unsigned int  empty_count;
192
194     assert(cache->slabs_full == NULL && cache->slabs_partial == NULL);
195
196     /* remove from cache list */
197     if(slab_cache_list == cache) {
198         slab_cache_list = cache->next;
199     }
200     else {
201         cache->prev->next = cache->next;
202     }
203
204     if(cache->next != NULL) {
205         cache->next->prev = cache->prev;
206     }
207
208     /* release all slabs */
209     slab         = cache->slabs_empty;
210     empty_count = 0;
211
212     while(slab != NULL) {
213         next = slab->next;
214
215         destroy_slab(cache, slab);
216
```

```
        slab = next;
        ++empty_count;
    }

    assert(cache->empty_count == empty_count);

    /* free cache structure */
    slab_cache_free(cache);
}
```

Here is the call graph for this function:

4.100.1.4 void slab_cache_free (void * *buffer*)

Definition at line 350 of file slab.c.

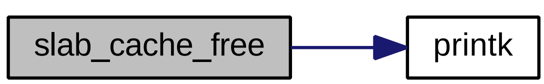
References ALIGN_START, slab_cache_t::bufctl_offset, slab_t::cache, slab_cache_t::dtor, slab_cache_t::empty_count, slab_cache_t::flags, slab_t::free_list, slab_cache_t::name, slab_bufctl_t::next, slab_t::next, slab_t::obj_count, slab_-cache_t::obj_size, slab_t::prev, printk(), SLAB_POISON, SLAB_POISON_DEAD_VALUE, 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 slab_cache_destroy().

```
350                                    {
351     addr_t           slab_start;
352     slab_t          *slab;
353     slab_cache_t    *cache;
354     slab_bufctl_t   *bufctl;
355     uint32_t        *rz_word;
356     uint32_t        *buffer32;
357     unsigned int     idx;
358 
359     /* compute address of slab data structure */
360     slab_start = ALIGN_START(buffer, SLAB_SIZE);
361     slab = (slab_t *)(slab_start + SLAB_SIZE - sizeof(slab_t) );
362 
363     /* obtain address of cache and bufctl */
364     cache  = slab->cache;
365     bufctl = (slab_bufctl_t *)((char *)buffer + cache->bufctl_offset);
366 
367     /* If slab is on the full slabs list, move it to the partial list
368      * since we are about to return a buffer to it. */
369     if(slab->free_list == NULL) {
370         /* remove from full slabs list */
371         if(cache->slabs_full == slab) {
372             cache->slabs_full = slab->next;
373         }
374         else {
375             slab->prev->next = slab->next;
376         }
377 
378         if(slab->next != NULL) {
379             slab->next->prev = slab->prev;
380         }
381 
382         /* add to partial slabs list */
383         slab->next           = cache->slabs_partial;
384         cache->slabs_partial = slab;
385 
386         if(slab->next != NULL) {
387             slab->next->prev = slab;
388         }
389     }
390 
```

```
    if(cache->flags & SLAB_RED_ZONE) {
        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) {
        if(cache->dtor != NULL) {
            cache->dtor(buffer, cache->obj_size);
        }

        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:



4.100.1.5 void slab_cache_grow (slab_cache_t * *cache*)

ASSERTION: slab address is not NULL

TODO: check this condition

Definition at line 449 of file slab.c.

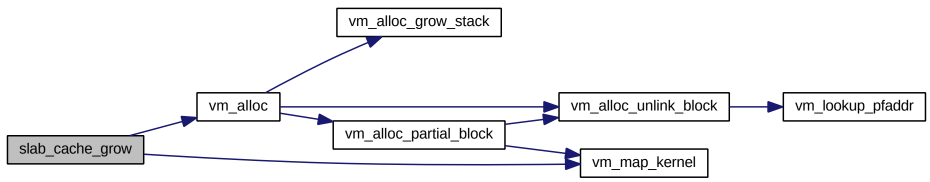
References slab_cache_t::alignment, slab_cache_t::alloc_size, assert, slab_cache_t::bufctl_offset, slab_t::cache, slab_t::colour, slab_cache_t::ctor, slab_cache_t::empty_count, slab_cache_t::flags, slab_t::free_list, global_page_allocator, slab_cache_t::max_colour, slab_bufctl_t::next, slab_t::next, slab_cache_t::next_colour, NULL, slab_t::obj_count, slab_cache_t::obj_size, pfalloc, slab_t::prev, SLAB_POISON, SLAB_POISON_DEAD_VALUE, SLAB_RED_ZONE, SLAB_RED_ZONE_VALUE, SLAB_SIZE, slab_cache_t::slabs_empty, vm_alloc(), VM_FLAG_READ_WRITE, and vm_map_kernel().

Referenced by slab_cache_alloc().

```
                                          {
    void            *slab_addr;
    slab_t          *slab;
    slab_bufctl_t   *bufctl;
    slab_bufctl_t   *next;
    addr_t           buffer;
    uint32_t        *buffer_end;
    uint32_t        *ptr;

    /* allocate new slab */
    slab_addr = vm_alloc( global_page_allocator );

    assert(slab_addr != NULL);

    vm_map_kernel(slab_addr, pfalloc(), VM_FLAG_READ_WRITE);

    slab = (slab_t *)( (char *)slab_addr + SLAB_SIZE - sizeof(slab_t) );

    slab->cache = cache;

    /* slab is initially empty */
    slab->obj_count = 0;

    slab->next         = cache->slabs_empty;
    cache->slabs_empty = slab;

    if(slab->next != NULL) {
        slab->next->prev = slab;
    }

    ++(cache->empty_count);

    /* set slab colour and update cache next colour */
    slab->colour = cache->next_colour;

    if(cache->next_colour < cache->max_colour) {
        cache->next_colour += cache->alignment;
    }
    else {
        cache->next_colour = 0;
    }

    /* compute address of first bufctl */
    bufctl          = (slab_bufctl_t *)( (char *)slab_addr + slab->colour + cache->
  bufctl_offset );
    slab->free_list = bufctl;

    while(1) {
        buffer = (addr_t)bufctl - cache->bufctl_offset;

        if(cache->flags & SLAB_POISON) {
            buffer_end = (uint32_t *)(buffer + cache->obj_size);

            for(ptr = (uint32_t *)buffer; ptr < buffer_end; ++ptr) {
                *ptr = SLAB_POISON_DEAD_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) {
                *ptr = SLAB_RED_ZONE_VALUE;
            }
        }
        else if (cache->ctor != NULL) {
            cache->ctor((void *)buffer, cache->obj_size);
        }

        next = (slab_bufctl_t *)( (char *)bufctl + cache->alloc_size );

```

```
519         if(next >= (slab_bufctl_t *)slab) {
520             bufctl->next = NULL;
521             break;
522         }
523 
524         bufctl->next = next;
525         bufctl = next;
526     }
527 }
```

Here is the call graph for this function:



4.100.1.6 void slab_cache_reap (slab_cache_t * *cache*)

Definition at line 529 of file slab.c.

References slab_cache_t::empty_count, slab_t::next, slab_cache_t::slabs_empty, and slab_cache_t::working_set.

```
529                                         {
530     slab_t          *slab;
531 
532     while(cache->empty_count > cache->working_set) {
533         /* select the first empty slab */
534         slab = cache->slabs_empty;
535 
536         /* unlink it and update count */
537         cache->slabs_empty  = slab->next;
538         cache->empty_count -= 1;
539 
540         /* destroy slab */
541         destroy_slab(cache, slab);
542     }
543 }
```

4.100.1.7 void slab_cache_set_working_set (slab_cache_t * *cache,* unsigned int *n*)

Definition at line 545 of file slab.c.

References slab_cache_t::working_set.

```
545                                                                 {
546     cache->working_set = n;
547 }
```

4.100.2 Variable Documentation

4.100.2.1 slab_cache_t* slab_cache_list = &slab_cache_cache

Definition at line 65 of file slab.c.

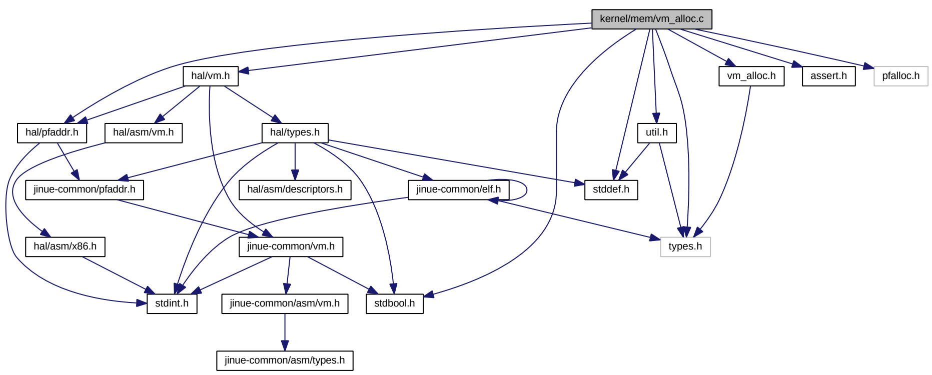
Referenced by slab_cache_create().

4.101 kernel/mem/vm_alloc.c File Reference

Virtual memory allocator.

```
#include <hal/pfaddr.h>
#include <hal/vm.h>
#include <assert.h>
#include <pfalloc.h>
#include <stdbool.h>
#include <stddef.h>
#include <types.h>
#include <util.h>
#include <vm_alloc.h>
```

Include dependency graph for vm_alloc.c:



Functions

- **addr_t vm_alloc** (**vm_alloc_t** *allocator)

  Allocate a page of virtual address space.
- **addr_t vm_alloc_low_latency** (**vm_alloc_t** *allocator)

  Allocate a page of virtual address space for time critical code path.
- void **vm_free** (**vm_alloc_t** *allocator, **addr_t** page)

  Free a page of virtual address space.
- void **vm_alloc_init** (**vm_alloc_t** *allocator, **addr_t** start_addr, **addr_t** end_addr)
- void **vm_alloc_destroy** (**vm_alloc_t** *allocator)
- void **vm_alloc_init_allocator** (**vm_alloc_t** *allocator, **addr_t** start_addr, **addr_t** end_addr)

  Basic initialization of virtual memory allocator.
- void **vm_alloc_add_region** (**vm_alloc_t** *allocator, **addr_t** start_addr, **addr_t** end_addr)

  Add a contiguous region of available virtual memory to the allocator.
- void **vm_alloc_free_block** (**vm_block_t** *block)

  Insert block in the free list.
- void **vm_alloc_partial_block** (**vm_block_t** *block)

  Insert block in the partial blocks list.
- void **vm_alloc_custom_block** (**vm_block_t** *block, **addr_t** start_addr, **addr_t** end_addr)
- void **vm_alloc_unlink_block** (**vm_block_t** *block)

  Unlink memory block from free or partial block list.
- void **vm_alloc_grow_stack** (**vm_block_t** *block)

Initialize the stack of a partial block with all remaining pages which have not yet been allocated.

- **addr_t vm_alloc_grow_single** (**vm_block_t** ∗block)

  Obtain a free page from a partial block, but defer page stack initialization for the block.

4.101.1 Detailed Description

Virtual memory allocator. Functions in this file are used to manage the virtual address space. Each region of the address space is represented by a **vm_alloc_t** (p. 52) structure.

Pages are allocated one at a time. There is no way to allocate groups of contiguous pages in the kernel.

Address space regions are split in 4MB-sized, 4MB-aligned blocks (1024 pages), each represented by **vm_block_t** (p. 54) structures. Each block may be either free (all pages available for allocation), partial (some pages available) or used (all pages allocated). For partial blocks, a page is used as a page stack for fast allocation and de-allocation.

vm_block_t (p. 54) structures for an address space region are placed in an array at the start of region. This array is used to quickly find the right **vm_block_t** (p. 54) structure during de-allocations. There is also a free block list (the free list) and a partial block list (the partial list) for each region (circular doubly-linked lists), which allows the allocator to quickly find a block with free pages during allocations.

Some implementation details:

Page stacks grow downward. We pre-decrement when de-allocating (adding pages on top of the stack) and post-increment when allocating (removing pages from the stack). This means the stack pointer points to the next allocatable page.

The prev and next members of **vm_block_t** (p. 54) link the block to the partial or free list (if applicable), and the stack member is the stack pointer for partial blocks. If the next member is NULL, then the block is unlinked, otherwise it is linked either to the free or the partial list. When the block is unlinked, the prev and stack_ptr members are undefined (probably not NULL). When the block is linked, either the stack_ptr member is NULL, in which case the block is free (linked to the free list), or it is non-NULL, in which case it is a partial block (linked to the partial list).

Definition in file **vm_alloc.c**.

4.101.2 Function Documentation

4.101.2.1 addr_t vm_alloc (vm_alloc_t ∗ *allocator*)

Allocate a page of virtual address space.

Parameters

allocator	allocator which manages the memory region from which we wish to obtain a page

ASSERTION: allocator is not null

ASSERTION: since block is expected to be partial, its stack pointer should not be null

ASSERTION: at this point, the page stack should not be empty (stack underflow check)

Definition at line 87 of file vm_alloc.c.

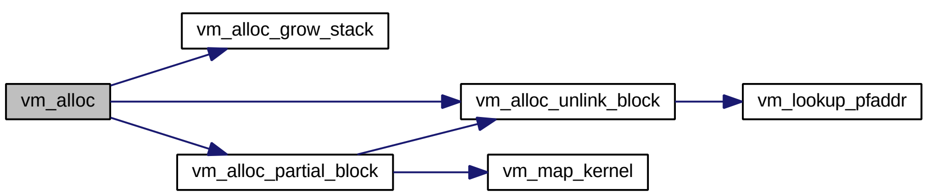
References assert, vm_alloc_t::free_list, NULL, vm_alloc_t::partial_list, vm_block_t::stack_ptr, VM_ALLOC_EMPTY_-STACK, vm_alloc_grow_stack(), vm_alloc_partial_block(), and vm_alloc_unlink_block().

Referenced by elf_load(), elf_setup_stack(), slab_cache_grow(), thread_page_create(), vm_clone_page_directory(), and vm_destroy_page_directory().

```
87                                    {
88      vm_block_t *block;
89      addr_t      page;
```

```
    assert(allocator != NULL);

    block = allocator->partial_list;

    if(block == NULL) {
        block = allocator->free_list;

        if(block == NULL) {
            return (addr_t)NULL;
        }

        vm_alloc_partial_block(block);
    }

    assert(block->stack_ptr != NULL);

    /* if the page stack is empty, perform deferred page stack initialization */
    if( VM_ALLOC_EMPTY_STACK(block) ) {
        vm_alloc_grow_stack(block);
    }

    assert( ! VM_ALLOC_EMPTY_STACK(block) );

    page = *(block->stack_ptr++);

    /* if we just emptied the stack, mark the block as used */
    if( VM_ALLOC_EMPTY_STACK(block) ) {
        vm_alloc_unlink_block(block);
    }

    return page;
}
```

Here is the call graph for this function:



4.101.2.2 void vm_alloc_add_region (vm_alloc_t * *allocator*, addr_t *start_addr*, addr_t *end_addr*)

Add a contiguous region of available virtual memory to the allocator.

Parameters

allocator	**vm_alloc_t** (p. 52) structure for a virtual memory allocator
start_addr	start address of the region
end_addr	end address of the region (first unavailable page)

Definition at line 344 of file vm_alloc.c.

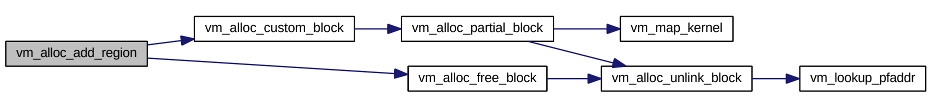
References ALIGN_END, vm_alloc_t::base_addr, vm_block_t::base_addr, vm_alloc_t::block_array, OFFSET_OF, vm_alloc_t::start_addr, VM_ALLOC_BLOCK_SIZE, vm_alloc_custom_block(), and vm_alloc_free_block().

Referenced by vm_alloc_init(), and vm_boot_init().

```
344                                                                            {
345     addr_t       start_addr_adjusted;
346     unsigned int start;
347     unsigned int end;
348     unsigned int end_full;
349     unsigned int idx;
```

```
    addr_t       limit;

    /* skip the block array */
    if(start_addr >= allocator->start_addr) {
        start_addr_adjusted = start_addr;
    }
    else {
        start_addr_adjusted = allocator->start_addr;
    }

    /* start and end block indices */
    start = ((unsigned int)start_addr_adjusted - (unsigned int)allocator->base_addr) / VM_ALLOC_BLOCK_SIZE;
    end   = ((unsigned int)end_addr            - (unsigned int)allocator->base_addr) / VM_ALLOC_BLOCK_SIZE;

    /* check and remember whether last block is partial (last_full < end) or
     * completely free (last_full == end) */
    if( OFFSET_OF(end_addr, VM_ALLOC_BLOCK_SIZE) == 0) {
        end_full = end;
    }
    else {
        end_full = end + 1;
    }

    /* array initialization -- first block (if partial) */
    idx = start;

    if( OFFSET_OF(start_addr_adjusted, VM_ALLOC_BLOCK_SIZE) != 0 ) {
        limit = ALIGN_END(start_addr_adjusted, VM_ALLOC_BLOCK_SIZE);

        if(end_addr < limit) {
            limit = end_addr;
        }

        vm_alloc_custom_block(&allocator->block_array[idx], start_addr_adjusted, limit);

        ++idx;
    }

    /* array initialization -- free blocks */
    for(; idx < end; ++idx) {
        vm_alloc_free_block(&allocator->block_array[idx]);
    }

    /* array initialization -- last block (if partial) */
    if(idx < end_full) {
        vm_alloc_custom_block(&allocator->block_array[idx], allocator->block_array[idx].base_addr, end_addr);
    }
}
```

Here is the call graph for this function:



4.101.2.3 void vm_alloc_custom_block (**vm_block_t** * *block*, **addr_t** *start_addr*, **addr_t** *end_addr*)

ASSERTION: block is not null

ASSERTION: start and end addresses must be page aligned

ASSERTION: start and end addr are inside block, address range is non-empty

ASSERTION: block is not free

ASSERTION: block is partial at this point

ASSERTION: page stack overflow check

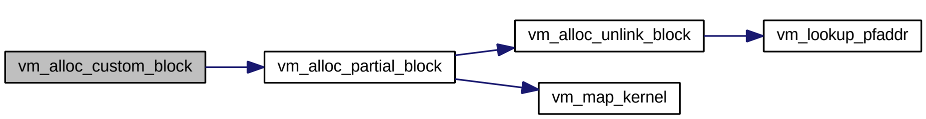
Definition at line 561 of file vm_alloc.c.

References assert, vm_block_t::base_addr, NULL, page_offset_of, PAGE_SIZE, vm_block_t::stack_addr, vm_block_t-::stack_ptr, VM_ALLOC_BLOCK_SIZE, VM_ALLOC_FULL_STACK, VM_ALLOC_IS_FREE, VM_ALLOC_IS_PARTIA-L, VM_ALLOC_IS_USED, and vm_alloc_partial_block().

Referenced by vm_alloc_add_region().

```
561                                                                                    {
562 #ifndef NDEBUG
563     addr_t      limit;
564 #endif
565     addr_t      page;
566     addr_t      adjusted_start;
567 
569     assert(block != NULL);
570 
572     assert(page_offset_of(start_addr) == 0 && page_offset_of(end_addr) == 0);
573 
574 #ifndef NDEBUG
575     limit = block->base_addr + VM_ALLOC_BLOCK_SIZE;
576 #endif
577 
579     assert(start_addr >= block->base_addr && end_addr <= limit && start_addr < end_addr );
580 
582     assert( ! VM_ALLOC_IS_FREE(block) );
583 
584     adjusted_start = start_addr;
585 
586     if( VM_ALLOC_IS_USED(block) ) {
587         /* if no stack address is specified at this point, use the first page
588          * of the address range for this purpose */
589         if( block->stack_addr == NULL ) {
590             block->stack_addr = (addr_t *)start_addr;
591             adjusted_start    = start_addr + PAGE_SIZE;
592 
593             /* if the address range contained only a single page, there is
594              * nothing left to do here */
595             if(adjusted_start >= end_addr) {
596                 return;
597             }
598         }
599 
600         vm_alloc_partial_block(block);
601     }
602 
604     assert( VM_ALLOC_IS_PARTIAL(block) );
605 
606     /* initialize stack */
607     page = adjusted_start;
608     while(page < end_addr) {
610         assert( ! VM_ALLOC_FULL_STACK(block) );
611 
612         *(--block->stack_ptr) = page;
613         page += PAGE_SIZE;
614     }
615 }
```

Here is the call graph for this function:



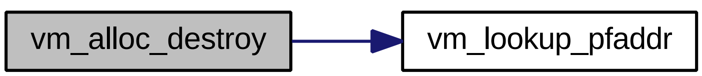
4.101.2.4 void vm_alloc_destroy (vm_alloc_t * *allocator*)

Definition at line 218 of file vm_alloc.c.

References vm_alloc_t::block_array, vm_block_t::next, NULL, PAGE_SIZE, vm_alloc_t::partial_list, pffree, vm_block_t::stack_addr, and vm_lookup_pfaddr().

```
                                          {
    vm_block_t   *head;
    vm_block_t   *block;
    pfaddr_t      paddr;
    addr_t        addr;
    unsigned int  idx;

    /* de-allocate page stacks */
    head  = allocator->partial_list;
    block = head;

    if(block != NULL) {
        do {
            paddr = vm_lookup_pfaddr(NULL, (addr_t)block->stack_addr);
            pffree(paddr);

            block = block->next;
        } while(block != head);
    }

    /* de-allocate block array pages */
    addr = (addr_t)allocator->block_array;
    for(idx = 0; idx < allocator->array_pages; ++idx) {
        paddr = vm_lookup_pfaddr(NULL, addr);
        pffree(paddr);

        addr += PAGE_SIZE;
    }
}
```

Here is the call graph for this function:



4.101.2.5 void vm_alloc_free_block (vm_block_t * *block*)

Insert block in the free list.

This is typically done when the block was a partial one, and the last page has just been returned to it.

Parameters

block	block to insert in the free list

ASSERTION: block is not null

ASSERTION: block->allocator should not be NULL

Definition at line 407 of file vm_alloc.c.

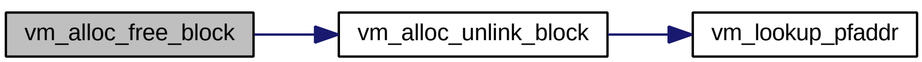
References vm_block_t::allocator, assert, vm_alloc_t::free_list, vm_block_t::next, NULL, vm_block_t::prev, vm_block_t::stack_ptr, and vm_alloc_unlink_block().

Referenced by vm_alloc_add_region(), and vm_free().

```
407                                          {
408     vm_block_t      *prev;
409     vm_block_t      *next;
410 
412     assert(block != NULL);
413 
414     /* unlink from partial list if necessary */
415     vm_alloc_unlink_block(block);
```

```
416
418     assert(block->allocator != NULL);
419
420     /* link block to the free list */
421     if(block->allocator->free_list == NULL) {
422         /* special case: free list is empty */
423         block->allocator->free_list = block;
424
425         block->next = block;
426         block->prev = block;
427     }
428     else {
429         /* block will be at the end of the free list */
430         next = block->allocator->free_list;
431         prev = next->prev;
432
433         /* re-link block */
434         block->prev = prev;
435         block->next = next;
436
437         prev->next = block;
438         next->prev = block;
439     }
440
441     /* set the stack pointer to null to indicate this is a free block */
442     block->stack_ptr = NULL;
443 }
```

Here is the call graph for this function:



4.101.2.6 addr_t vm_alloc_grow_single (vm_block_t * *block*)

Obtain a free page from a partial block, but defer page stack initialization for the block.

This function must only be called on a partial block, and only after checking first that the page stack is empty. This function takes care of unlinking the block from the partial list if the last page is allocated.

Parameters

block	block from which to allocate the page

ASSERTION: block is not null

ASSERTION: block is linked (it should be in the partial list)

ASSERTION: block actually has a stack

ASSERTION: region can still grow

Definition at line 734 of file vm_alloc.c.

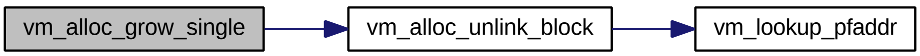
References assert, vm_block_t::next, NULL, PAGE_SIZE, vm_block_t::prev, vm_block_t::stack_next, vm_block_t::stack_ptr, VM_ALLOC_CANNOT_GROW, and vm_alloc_unlink_block().

Referenced by vm_alloc_low_latency().

```
734                                                  {
735     addr_t page;
736
738     assert(block != NULL);
739
741     assert(block->next != NULL && block->prev != NULL);
742
744     assert(block->stack_ptr != NULL);
745
```

```
    assert( ! VM_ALLOC_CANNOT_GROW(block) );

    page              = block->stack_next;
    block->stack_next = page + PAGE_SIZE;

    if( VM_ALLOC_CANNOT_GROW(block) ) {
        /* block is now used up, remove it from the partial list */
        vm_alloc_unlink_block(block);
    }

    return page;
}
```

Here is the call graph for this function:



4.101.2.7 void vm_alloc_grow_stack (vm_block_t * *block*)

Initialize the stack of a partial block with all remaining pages which have not yet been allocated.

Parameters

block	block which will have its stack initialized

ASSERTION: block is not null

ASSERTION: block is linked (it should be in the partial list)

ASSERTION: block actually has a stack

ASSERTION: stack underflow check

Definition at line 695 of file vm_alloc.c.

References assert, vm_block_t::base_addr, vm_block_t::next, NULL, PAGE_SIZE, vm_block_t::prev, vm_block_t::stack_next, vm_block_t::stack_ptr, VM_ALLOC_BLOCK_SIZE, and VM_ALLOC_FULL_STACK.

Referenced by vm_alloc().

```
695                                         {
696     addr_t  limit;
697     addr_t  page;
698     addr_t *stack_ptr;
699
701     assert(block != NULL);
702
704     assert(block->next != NULL && block->prev != NULL);
705
707     assert(block->stack_ptr != NULL);
708
709     stack_ptr = block->stack_ptr;
710     page      = block->stack_next;
711     limit     = block->base_addr + VM_ALLOC_BLOCK_SIZE;
712
713     while(page < limit) {
715         assert( ! VM_ALLOC_FULL_STACK(block) );
716
717         *(--stack_ptr) = page;
718
719         page += PAGE_SIZE;
720     }
721
722     block->stack_ptr  = stack_ptr;
723     block->stack_next = limit;
724 }
```

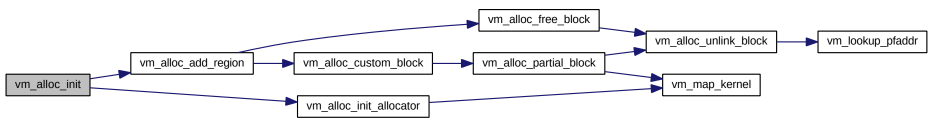
4.101.2.8 void vm_alloc_init (vm_alloc_t * *allocator*, addr_t *start_addr*, addr_t *end_addr*)

Definition at line 213 of file vm_alloc.c.

References vm_alloc_add_region(), and vm_alloc_init_allocator().

```
213                                                                                 {
214     vm_alloc_init_allocator( allocator, start_addr, end_addr);
215     vm_alloc_add_region(     allocator, start_addr, end_addr);
216 }
```

Here is the call graph for this function:



4.101.2.9 void vm_alloc_init_allocator (vm_alloc_t * *allocator*, addr_t *start_addr*, addr_t *end_addr*)

Basic initialization of virtual memory allocator.

Parameters

allocator	**vm_alloc_t** (p. 52) structure for a virtual memory allocator
start_addr	start address of the region managed by the allocator
size	size of the region managed by the allocator

ASSERTION: allocator structure pointer must not be null

ASSERTION: start and end addresses must be multiples of page size (page-aligned memory region)

ASSERTION: once all the array pages are allocated, we should have reached the allocatable pages region

Definition at line 254 of file vm_alloc.c.

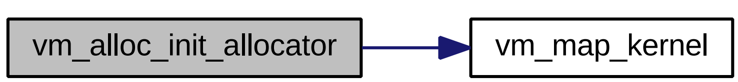
References ALIGN_END, ALIGN_START, vm_block_t::allocator, vm_alloc_t::array_pages, assert, vm_alloc_t::base_addr, vm_block_t::base_addr, vm_alloc_t::block_array, vm_alloc_t::block_count, vm_alloc_t::end_addr, vm_alloc_t::free_list, vm_block_t::next, NULL, page_offset_of, PAGE_SIZE, vm_alloc_t::partial_list, pfalloc, vm_block_t::stack_addr, vm_alloc_t::start_addr, VM_ALLOC_BLOCK_SIZE, VM_FLAG_READ_WRITE, and vm_map_kernel().

Referenced by vm_alloc_init(), and vm_boot_init().

```
254                                                                                           {
255     addr_t        base_addr;        /* block-aligned start address */
256     addr_t        aligned_end;        /* block-aligned end address */
257     addr_t        adjusted_start;    /* actual start of available memory, block array skipped */
258
259     vm_block_t   *block_array;        /* start of array */
260     unsigned int  block_count;        /* array size, in blocks (entries) */
261     size_t        array_size;        /* array size, in bytes */
262     unsigned int  array_page_count;    /* array size, in pages */
263
264     addr_t        addr;                /* some virtual address */
265     pfaddr_t      paddr;            /* some page frame address */
266     unsigned int  idx;                /* an array index */
267
268
270     assert(allocator != NULL);
271
273     assert( page_offset_of(start_addr) == 0 && page_offset_of(end_addr) == 0 );
274
275
276     /* align base and end addresses to block size */
```

```
    base_addr   = (addr_t)ALIGN_START(start_addr, VM_ALLOC_BLOCK_SIZE);
    aligned_end = (addr_t)ALIGN_END(end_addr, VM_ALLOC_BLOCK_SIZE);

    /* calculate number of memory blocks managed by this allocator */
    block_count = ( (char *)aligned_end - (char *)base_addr ) / VM_ALLOC_BLOCK_SIZE;

    /* calculate the number of pages required to store the memory block
     * descriptor array */
    array_size = block_count * sizeof(vm_block_t);
    array_page_count = array_size / PAGE_SIZE;
    if(array_size % PAGE_SIZE != 0) {
        ++array_page_count;
    }

    /* address of the block array */
    block_array = (vm_block_t *)start_addr;

    /* adjust base address to skip block descriptor array */
    adjusted_start = start_addr + array_page_count * PAGE_SIZE;

    /* initialize allocator struct */
    allocator->start_addr   = adjusted_start;
    allocator->end_addr     = end_addr;
    allocator->base_addr    = base_addr;
    allocator->block_count  = block_count;
    allocator->block_array  = block_array;
    allocator->array_pages  = array_page_count;
    allocator->free_list    = NULL;
    allocator->partial_list = NULL;

    /* allocate block descriptor array pages */
    addr = (addr_t)block_array;
    for(idx = 0; idx < array_page_count; ++idx) {
        /* allocate and map page */
        paddr = pfalloc();
        vm_map_kernel(addr, paddr, VM_FLAG_READ_WRITE);

        /* calculate address of next page */
        addr += PAGE_SIZE;
    }

    assert(addr == adjusted_start);

    /* basic initialization of array (all blocks unlinked/used) */
    addr = base_addr;
    for(idx = 0; idx < block_count; ++idx) {
        block_array[idx].base_addr  = addr;
        block_array[idx].allocator  = allocator;

        /* mark block as unlinked for now */
        block_array[idx].next       = NULL;

        /* a null stack base indicates the block is uninitialized */
        block_array[idx].stack_addr = NULL;

        /* calculate address of next block */
        addr += VM_ALLOC_BLOCK_SIZE;
    }
}
```

Here is the call graph for this function:



4.101.2.10 addr_t vm_alloc_low_latency (vm_alloc_t * *allocator*)

Allocate a page of virtual address space for time critical code path.

Same as **vm_alloc()** (p. 252), but some time consuming housekeeping steps are deferred.

Parameters

allocator	allocator which manages the memory region from which we wish to obtain a page

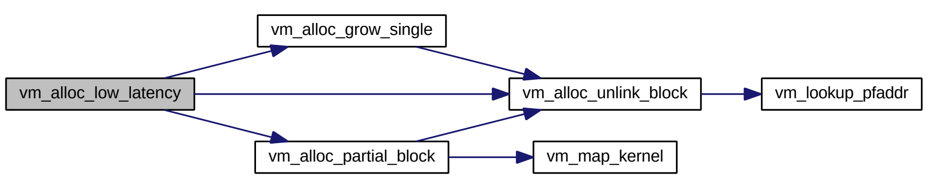
ASSERTION: allocator is not null

Definition at line 133 of file vm_alloc.c.

References assert, vm_alloc_t::free_list, NULL, vm_alloc_t::partial_list, vm_block_t::stack_ptr, VM_ALLOC_EMPTY_-STACK, vm_alloc_grow_single(), vm_alloc_partial_block(), and vm_alloc_unlink_block().

```
133                                                    {
134     vm_block_t *block;
135     addr_t      page;
136 
138     assert(allocator != NULL);
139 
140     block = allocator->partial_list;
141 
142     if(block == NULL) {
143         block = allocator->free_list;
144 
145         if(block == NULL) {
146             return (addr_t)NULL;
147         }
148 
149         vm_alloc_partial_block(block);
150     }
151 
152     /* if the page stack is empty, allocate sequentially from the start of the
153      * block and continue to defer page stack initialization */
154     if( VM_ALLOC_EMPTY_STACK(block) ) {
155         return vm_alloc_grow_single(block);
156     }
157 
158     page = *(block->stack_ptr++);
159 
160     if( VM_ALLOC_EMPTY_STACK(block) ) {
161         /* block is now used up, remove it from the partial blocks list */
162         vm_alloc_unlink_block(block);
163     }
164 
165     return page;
166 }
```

Here is the call graph for this function:



4.101.2.11 void vm_alloc_partial_block (vm_block_t * *block*)

Insert block in the partial blocks list.

This is typically done when the block is a free one from which we intend to allocate pages, or when the block is used (unlinked) and we intend to return pages to it. The stack is initialized empty, but the deferred stack initialization mechanism is enabled if the block is free on function entry.

Parameters

block	block to insert in the partial list

ASSERTION: block is not null

ASSERTION: block stack address is not null

ASSERTION: block->allocator should not be NULL

Definition at line 454 of file vm_alloc.c.

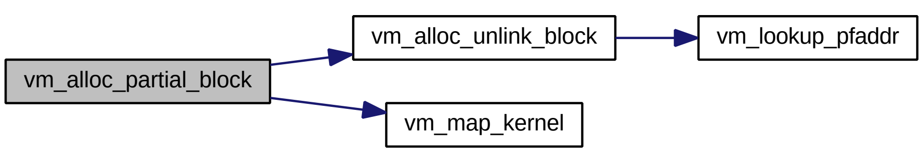
References vm_block_t::allocator, assert, vm_block_t::base_addr, vm_block_t::next, NULL, PAGE_SIZE, vm_alloc_t::partial_list, pfalloc, vm_block_t::prev, vm_block_t::stack_addr, vm_block_t::stack_next, vm_block_t::stack_ptr, VM_ALLOC_BLOCK_SIZE, vm_alloc_unlink_block(), VM_FLAG_READ_WRITE, and vm_map_kernel().

Referenced by vm_alloc(), vm_alloc_custom_block(), vm_alloc_low_latency(), and vm_free().

```
454                                            {
455     vm_block_t      *prev;
456     vm_block_t      *next;
457     addr_t          *stack_addr;
458     pfaddr_t         paddr;
459     bool             was_free;
460 
461 
463     assert(block != NULL);
464 
465     /* To keep in mind...
466      *
467      * When the allocator is initialized, some blocks may be created partial
468      * (typical for the first and the last block of the region). If there is a
469      * hole at the start of the block, the page stack will be at the first
470      * available page, not at the start of the block. Since these blocks have
471      * holes, they will never be in the free state.
472      *
473      * So, when a block is free on function entry, we ensure the stack is placed
474      * at the start of the block so that all the remaining pages can be
475      * allocated sequentially (see deferred stack intialization below). However,
476      * if the block is in the used state on function entry, we leave the stack
477      * at its previous location since the first page of the block might not be
478      * available. */
479 
480     if(block->next == NULL) {
482         assert(block->stack_addr != NULL);
483 
484         /* block was used on function entry */
485         was_free = false;
486     }
487     else {
488         if(block->stack_ptr != NULL) {
489             /* block is already partial, leave it untouched */
490             return;
491         }
492 
493         /* block was free on function entry */
494         was_free = true;
495 
496         /* unlink from free list */
497         vm_alloc_unlink_block(block);
498 
499         /* use first page of block for the stack */
500         block->stack_addr = (addr_t *)block->base_addr;
501     }
502 
503     /* allocate the page stack */
504     stack_addr  = block->stack_addr;
505     paddr       = pfalloc();
506     vm_map_kernel((addr_t)stack_addr, paddr, VM_FLAG_READ_WRITE);
507 
509     assert(block->allocator != NULL);
510 
511     /* link block to the partial list */
512     if(block->allocator->partial_list == NULL) {
513         /* special case: partial list is empty */
514         block->allocator->partial_list = block;
515 
516         block->next = block;
```

```c
        block->prev = block;
    }
    else {
        /* block will be at to the end of the partial block list */
        next = block->allocator->partial_list;
        prev = next->prev;

        /* re-link block */
        block->prev = prev;
        block->next = next;

        prev->next = block;
        next->prev = block;
    }

    /* Ok, here's the deal (deferred stack intialization)...
     *
     * We do not want to initialize the page stack right now because this is
     * a time consuming operation, and we might be in time-critical code
     * (interrupt handling code for example). Instead, the stack initialization
     * is deferred until the next page allocations. The first non-time critical
     * allocation which encounters an empty stack will initialize the whole
     * stack. In the meantime, time critical ones will just allocate pages
     * sequentially from the start of the block.
     *
     * The stack_next pointer in the vm_block_t structure points to the next
     * page available for sequential allocation. The memory block is actually
     * used up (no more pages available) when the page stack is empty AND the
     * stack_next pointer has reached the end of the block. */

    /* initialize the stack as empty */
    block->stack_ptr = (addr_t *)( (char *)stack_addr + PAGE_SIZE );

    if(was_free) {
        /* free block: we skip the first page as it was allocated for the
         * stack itself */
        block->stack_next = block->base_addr + PAGE_SIZE;
    }
    else {
        /* used block: sequential allocation no longer possible */
        block->stack_next = block->base_addr + VM_ALLOC_BLOCK_SIZE;
    }
}
```

Here is the call graph for this function:



4.101.2.12 void vm_alloc_unlink_block (vm_block_t * *block*)

Unlink memory block from free or partial block list.

It is not an error if block is not linked to either list. On exit of this funtion, the block is in the used state.

Parameters

block	block to unlink from list

ASSERTION: block is not null

ASSERTION: block is either properly linked (no null pointers) or not at all (next is null)

ASSERTION: block->allocator should not be NULL

ASSERTION: block should not be the head of both free and partial lists

ASSERTION: if block is alone in its list, the previous node pointer should point to self

ASSERTION: if block is alone in its list, we expect it to be the head of either the free or the partial list

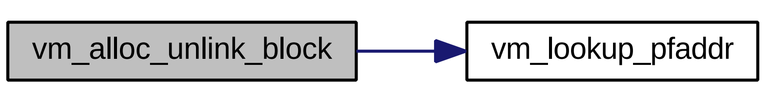
Definition at line 623 of file vm_alloc.c.

References vm_block_t::allocator, assert, vm_alloc_t::free_list, vm_block_t::next, NULL, vm_alloc_t::partial_list, pffree, vm_block_t::prev, vm_block_t::stack_addr, vm_block_t::stack_ptr, and vm_lookup_pfaddr().

Referenced by vm_alloc(), vm_alloc_free_block(), vm_alloc_grow_single(), vm_alloc_low_latency(), and vm_alloc_-partial_block().

```
623                                              {
624     vm_alloc_t *allocator;
625     pfaddr_t    paddr;
626 
628     assert(block != NULL);
629 
632     assert(block->prev != NULL || block->next == NULL);
633 
635     assert(block->allocator != NULL);
636 
637     /* get allocator for block (required for next assert as well as subsequent code) */
638     allocator = block->allocator;
639 
641     assert(allocator->free_list != block || allocator->partial_list != block);
642 
643     /* if block is already unlinked, we have nothing to do here */
644     if(block->next == NULL) {
645         return;
646     }
647 
648     /* if block has a stack, discard it */
649     if(block->stack_ptr != NULL) {
650         paddr = vm_lookup_pfaddr(NULL, (addr_t)block->stack_addr);
651         pffree(paddr);
652     }
653 
654     /* special case: block is alone in its list */
655     if(block->next == block) {
658         assert(block->prev == block);
659 
662         assert(allocator->free_list == block || allocator->partial_list == block);
663 
664         if(allocator->free_list == block) {
665             allocator->free_list = NULL;
666         }
667 
668         if(allocator->partial_list == block) {
669             allocator->partial_list = NULL;
670         }
671     }
672     else {
673         if(allocator->free_list == block) {
674             allocator->free_list = block->next;
675         }
676 
677         if(allocator->partial_list == block) {
678             allocator->partial_list = block->next;
679         }
680 
681         /* unlink block */
682         block->next->prev = block->prev;
683         block->prev->next = block->next;
684     }
685 
686     /* set next pointer to null to indicate block is unlinked */
687     block->next = NULL;
688 }
```

Here is the call graph for this function:



4.101.2.13 void vm_free (vm_alloc_t * *allocator*, addr_t *page*)

Free a page of virtual address space.

Parameters

allocator	allocator which manages the memory region to which the page is freed

ASSERTION: allocator is not null

ASSERTION: ensure we are freeing to the proper allocator/region

ASSERTION: ensure address is page aligned

ASSERTION: block should now be partial

ASSERTION: stack overflow check

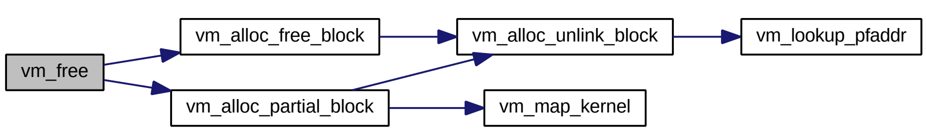
Definition at line 172 of file vm_alloc.c.

References assert, vm_alloc_t::base_addr, vm_alloc_t::block_array, NULL, page_offset_of, vm_block_t::stack_addr, vm_block_t::stack_ptr, vm_alloc_t::start_addr, VM_ALLOC_BLOCK_SIZE, vm_alloc_free_block(), VM_ALLOC_FULL_STACK, VM_ALLOC_IS_PARTIAL, VM_ALLOC_IS_USED, and vm_alloc_partial_block().

Referenced by elf_load(), elf_setup_stack(), thread_page_create(), and thread_page_destroy().

```
172                                                 {
173     vm_block_t   *block;
174     unsigned int  idx;
175 
177     assert(allocator != NULL);
178 
180     assert(page >= allocator->start_addr && page < allocator->end_addr);
181 
183     assert(page_offset_of(page) == 0);
184 
185     /* find the block to which the free page belong */
186     idx = ( (unsigned int)page - (unsigned int)allocator->base_addr ) /
      VM_ALLOC_BLOCK_SIZE;
187     block = &allocator->block_array[idx];
188 
189     /* if the block was a used block, make it a partial block */
190     if( VM_ALLOC_IS_USED(block) ) {
191         if(block->stack_addr == NULL) {
192             block->stack_addr = (addr_t *)page;
193             return;
194         }
195 
196         vm_alloc_partial_block(block);
197     }
198 
200     assert( VM_ALLOC_IS_PARTIAL(block) );
201 
203     assert( ! VM_ALLOC_FULL_STACK(block) );
204 
205     *(--block->stack_ptr) = page;
206 
207     /* check if we just freed the whole block */
208     if( VM_ALLOC_FULL_STACK(block) ) {
209         vm_alloc_free_block(block);
210     }
211 }
```

Here is the call graph for this function:



Index