

Jinue

Generated by Doxygen 1.8.5

Fri Jan 4 2019 22:25:11

Contents

1	Data Structure Index	1
1.1	Data Structures	1
2	File Index	3
2.1	File List	3
3	Data Structure Documentation	7
3.1	addr_space_t Struct Reference	7
3.1.1	Detailed Description	8
3.1.2	Field Documentation	8
3.1.2.1	cr3	8
3.1.2.2	pd	8
3.1.2.3	pdpt	8
3.1.2.4	top_level	8
3.2	boot_info_t Struct Reference	8
3.2.1	Detailed Description	9
3.2.2	Field Documentation	9
3.2.2.1	boot_end	9
3.2.2.2	boot_heap	9
3.2.2.3	e820_entries	9
3.2.2.4	e820_map	9
3.2.2.5	image_start	9
3.2.2.6	image_top	10
3.2.2.7	kernel_size	10
3.2.2.8	kernel_start	10
3.2.2.9	page_directory	10
3.2.2.10	page_table	10
3.2.2.11	proc_size	10
3.2.2.12	proc_start	10

3.2.2.13	setup_signature	10
3.3	bootmem_t Struct Reference	11
3.3.1	Detailed Description	11
3.3.2	Field Documentation	11
3.3.2.1	addr	11
3.3.2.2	count	11
3.3.2.3	next	11
3.4	cpu_data_t Struct Reference	11
3.4.1	Detailed Description	12
3.4.2	Field Documentation	12
3.4.2.1	current_addr_space	12
3.4.2.2	gdt	12
3.4.2.3	self	13
3.4.2.4	tss	13
3.5	cpu_info_t Struct Reference	13
3.5.1	Detailed Description	13
3.5.2	Field Documentation	13
3.5.2.1	dcache_alignment	13
3.5.2.2	family	13
3.5.2.3	features	14
3.5.2.4	model	14
3.5.2.5	stepping	14
3.5.2.6	vendor	14
3.6	e820_t Struct Reference	14
3.6.1	Detailed Description	14
3.6.2	Field Documentation	14
3.6.2.1	addr	14
3.6.2.2	size	15
3.6.2.3	type	15
3.7	Elf32_auxv_t Struct Reference	15
3.7.1	Detailed Description	15
3.7.2	Field Documentation	15
3.7.2.1	a_type	15
3.7.2.2	a_un	15
3.7.2.3	a_val	15
3.8	Elf32_Ehdr Struct Reference	16
3.8.1	Detailed Description	16

3.8.2	Field Documentation	16
3.8.2.1	e_ehsize	16
3.8.2.2	e_entry	16
3.8.2.3	e_flags	16
3.8.2.4	e_ident	17
3.8.2.5	e_machine	17
3.8.2.6	e_phentsize	17
3.8.2.7	e_phnum	17
3.8.2.8	e_phoff	17
3.8.2.9	e_shentsize	17
3.8.2.10	e_shnum	17
3.8.2.11	e_shoff	17
3.8.2.12	e_shstrndx	17
3.8.2.13	e_type	18
3.8.2.14	e_version	18
3.9	Elf32_Phdr Struct Reference	18
3.9.1	Detailed Description	18
3.9.2	Field Documentation	18
3.9.2.1	p_align	18
3.9.2.2	p_filesz	18
3.9.2.3	p_flags	18
3.9.2.4	p_memsz	19
3.9.2.5	p_offset	19
3.9.2.6	p_paddr	19
3.9.2.7	p_type	19
3.9.2.8	p_vaddr	19
3.10	Elf32_Shdr Struct Reference	19
3.10.1	Detailed Description	19
3.10.2	Field Documentation	20
3.10.2.1	sh_addr	20
3.10.2.2	sh_addralign	20
3.10.2.3	sh_entsize	20
3.10.2.4	sh_flags	20
3.10.2.5	sh_info	20
3.10.2.6	sh_link	20
3.10.2.7	sh_name	20
3.10.2.8	sh_offset	20

3.10.2.9	sh_size	20
3.10.2.10	sh_type	21
3.11	Elf32_Sym Struct Reference	21
3.11.1	Detailed Description	21
3.11.2	Field Documentation	21
3.11.2.1	st_info	21
3.11.2.2	st_name	21
3.11.2.3	st_other	21
3.11.2.4	st_shndx	21
3.11.2.5	st_size	22
3.11.2.6	st_value	22
3.12	elf_info_t Struct Reference	22
3.12.1	Detailed Description	23
3.12.2	Field Documentation	23
3.12.2.1	addr_space	23
3.12.2.2	at_phdr	23
3.12.2.3	at_phent	23
3.12.2.4	at_phnum	23
3.12.2.5	entry	23
3.12.2.6	stack_addr	23
3.13	elf_symbol_t Struct Reference	24
3.13.1	Detailed Description	24
3.13.2	Field Documentation	24
3.13.2.1	addr	24
3.13.2.2	name	24
3.14	ipc_t Struct Reference	24
3.14.1	Detailed Description	25
3.14.2	Field Documentation	25
3.14.2.1	header	25
3.14.2.2	recv_list	25
3.14.2.3	send_list	25
3.15	jinue_list_t Struct Reference	26
3.15.1	Detailed Description	26
3.15.2	Field Documentation	26
3.15.2.1	head	26
3.15.2.2	tail	26
3.16	jinue_message_t Struct Reference	27

3.16.1 Detailed Description	27
3.16.2 Field Documentation	27
3.16.2.1 buffer_size	27
3.16.2.2 cookie	27
3.16.2.3 data_size	27
3.16.2.4 desc_n	27
3.16.2.5 function	27
3.17 jinue_node_t Struct Reference	27
3.17.1 Detailed Description	28
3.17.2 Field Documentation	28
3.17.2.1 next	28
3.18 jinue_reply_t Struct Reference	28
3.18.1 Detailed Description	28
3.18.2 Field Documentation	28
3.18.2.1 data_size	28
3.18.2.2 desc_n	29
3.19 jinue_syscall_args_t Struct Reference	29
3.19.1 Detailed Description	29
3.19.2 Field Documentation	29
3.19.2.1 arg0	29
3.19.2.2 arg1	29
3.19.2.3 arg2	29
3.19.2.4 arg3	29
3.20 kernel_context_t Struct Reference	30
3.20.1 Detailed Description	30
3.20.2 Field Documentation	30
3.20.2.1 ebp	30
3.20.2.2 ebx	30
3.20.2.3 edi	30
3.20.2.4 eip	30
3.20.2.5 esi	30
3.21 memory_block_t Struct Reference	31
3.21.1 Detailed Description	31
3.21.2 Field Documentation	31
3.21.2.1 addr	31
3.21.2.2 count	31
3.22 message_info_t Struct Reference	31

3.22.1 Detailed Description	31
3.22.2 Field Documentation	32
3.22.2.1 buffer_size	32
3.22.2.2 cookie	32
3.22.2.3 data_size	32
3.22.2.4 desc_n	32
3.22.2.5 function	32
3.22.2.6 total_size	32
3.23 object_header_t Struct Reference	32
3.23.1 Detailed Description	33
3.23.2 Field Documentation	33
3.23.2.1 flags	33
3.23.2.2 ref_count	33
3.23.2.3 type	33
3.24 object_ref_t Struct Reference	33
3.24.1 Detailed Description	34
3.24.2 Field Documentation	34
3.24.2.1 cookie	34
3.24.2.2 flags	34
3.24.2.3 object	34
3.25 pdpt_t Struct Reference	34
3.25.1 Detailed Description	35
3.25.2 Field Documentation	35
3.25.2.1 pd	35
3.26 pfcache_t Struct Reference	35
3.26.1 Detailed Description	35
3.26.2 Field Documentation	35
3.26.2.1 count	35
3.26.2.2 ptr	35
3.27 process_t Struct Reference	35
3.27.1 Detailed Description	36
3.27.2 Field Documentation	36
3.27.2.1 addr_space	36
3.27.2.2 descriptors	36
3.27.2.3 header	37
3.28 pseudo_descriptor_t Struct Reference	37
3.28.1 Detailed Description	37

3.28.2	Field Documentation	37
3.28.2.1	addr	37
3.28.2.2	limit	37
3.28.2.3	padding	37
3.29	pte_t Struct Reference	37
3.29.1	Detailed Description	38
3.29.2	Field Documentation	38
3.29.2.1	entry	38
3.29.2.2	entry	38
3.30	slab_bufctl_t Struct Reference	38
3.30.1	Detailed Description	38
3.30.2	Field Documentation	38
3.30.2.1	next	38
3.31	slab_cache_t Struct Reference	39
3.31.1	Detailed Description	39
3.31.2	Field Documentation	40
3.31.2.1	alignment	40
3.31.2.2	alloc_size	40
3.31.2.3	bufctl_offset	40
3.31.2.4	ctor	40
3.31.2.5	dtor	40
3.31.2.6	empty_count	40
3.31.2.7	flags	40
3.31.2.8	max_colour	40
3.31.2.9	name	41
3.31.2.10	next	41
3.31.2.11	next_colour	41
3.31.2.12	obj_size	41
3.31.2.13	prev	41
3.31.2.14	slabs_empty	41
3.31.2.15	slabs_full	41
3.31.2.16	slabs_partial	41
3.31.2.17	working_set	41
3.32	slab_t Struct Reference	42
3.32.1	Detailed Description	42
3.32.2	Field Documentation	42
3.32.2.1	cache	43

3.32.2.2	colour	43
3.32.2.3	free_list	43
3.32.2.4	next	43
3.32.2.5	obj_count	43
3.32.2.6	prev	43
3.33	thread_context_t Struct Reference	43
3.33.1	Detailed Description	44
3.33.2	Field Documentation	44
3.33.2.1	local_storage_addr	44
3.33.2.2	local_storage_size	44
3.33.2.3	saved_stack_pointer	44
3.34	thread_t Struct Reference	44
3.34.1	Detailed Description	45
3.34.2	Field Documentation	45
3.34.2.1	header	45
3.34.2.2	message_args	45
3.34.2.3	message_buffer	45
3.34.2.4	message_info	45
3.34.2.5	process	45
3.34.2.6	sender	45
3.34.2.7	thread_ctx	46
3.34.2.8	thread_list	46
3.35	trapframe_t Struct Reference	46
3.35.1	Detailed Description	46
3.35.2	Field Documentation	47
3.35.2.1	cs	47
3.35.2.2	ds	47
3.35.2.3	eax	47
3.35.2.4	ebp	47
3.35.2.5	ebx	47
3.35.2.6	ecx	47
3.35.2.7	edi	47
3.35.2.8	edx	47
3.35.2.9	eflags	47
3.35.2.10	eip	47
3.35.2.11	errcode	48
3.35.2.12	es	48

3.35.2.13 esi	48
3.35.2.14 esp	48
3.35.2.15 fs	48
3.35.2.16 gs	48
3.35.2.17 ivt	48
3.35.2.18 ss	48
3.36 tss_t Struct Reference	49
3.36.1 Detailed Description	49
3.36.2 Field Documentation	49
3.36.2.1 cr3	49
3.36.2.2 cs	49
3.36.2.3 debug	50
3.36.2.4 ds	50
3.36.2.5 eax	50
3.36.2.6 ebp	50
3.36.2.7 ebx	50
3.36.2.8 ecx	50
3.36.2.9 edi	50
3.36.2.10 edx	50
3.36.2.11 eflags	50
3.36.2.12 eip	50
3.36.2.13 es	50
3.36.2.14 esi	51
3.36.2.15 esp	51
3.36.2.16 esp0	51
3.36.2.17 esp1	51
3.36.2.18 esp2	51
3.36.2.19 fs	51
3.36.2.20 gs	51
3.36.2.21 iomap	51
3.36.2.22 ldt	51
3.36.2.23 prev	51
3.36.2.24 ss	52
3.36.2.25 ss0	52
3.36.2.26 ss1	52
3.36.2.27 ss2	52
3.37 vm_alloc_t Struct Reference	52

3.37.1 Detailed Description	53
3.37.2 Field Documentation	53
3.37.2.1 array_pages	53
3.37.2.2 base_addr	53
3.37.2.3 block_array	53
3.37.2.4 block_count	53
3.37.2.5 end_addr	54
3.37.2.6 free_list	54
3.37.2.7 partial_list	54
3.37.2.8 start_addr	54
3.38 vm_block_t Struct Reference	54
3.38.1 Detailed Description	55
3.38.2 Field Documentation	55
3.38.2.1 allocator	55
3.38.2.2 base_addr	56
3.38.2.3 next	56
3.38.2.4 prev	56
3.38.2.5 stack_addr	56
3.38.2.6 stack_next	56
3.38.2.7 stack_ptr	56
3.39 x86_cpuid_regs_t Struct Reference	57
3.39.1 Detailed Description	57
3.39.2 Field Documentation	57
3.39.2.1 eax	57
3.39.2.2 ebx	57
3.39.2.3 ecx	57
3.39.2.4 edx	57
4 File Documentation	59
4.1 include/ascii.h File Reference	59
4.1.1 Macro Definition Documentation	59
4.1.1.1 CHAR_BS	59
4.1.1.2 CHAR_CR	59
4.1.1.3 CHAR_HT	60
4.1.1.4 CHAR_LF	60
4.2 include/console.h File Reference	60
4.3 include/jinue/console.h File Reference	60

4.4	include/jinue-common/console.h File Reference	60
4.4.1	Macro Definition Documentation	61
4.4.1.1	CONSOLE_SERIAL_BAUD_RATE	61
4.4.1.2	CONSOLE_SERIAL_IOPORT	61
4.4.2	Function Documentation	61
4.4.2.1	console_init	61
4.4.2.2	console_print	62
4.4.2.3	console_printn	62
4.4.2.4	console_putc	62
4.5	include/core.h File Reference	63
4.5.1	Function Documentation	63
4.5.1.1	kmain	63
4.6	include/debug.h File Reference	66
4.6.1	Function Documentation	66
4.6.1.1	dump_call_stack	66
4.7	include/elf.h File Reference	67
4.8	include/jinue/elf.h File Reference	67
4.9	include/jinue-common/elf.h File Reference	68
4.9.1	Macro Definition Documentation	73
4.9.1.1	AT_BASE	73
4.9.1.2	AT_DCACHEBSIZE	73
4.9.1.3	AT_ENTRY	73
4.9.1.4	AT_EXECFD	73
4.9.1.5	AT_FLAGS	73
4.9.1.6	AT_HWCAP	73
4.9.1.7	AT_HWCAP2	73
4.9.1.8	AT_ICACHEBSIZE	73
4.9.1.9	AT_IGNORE	74
4.9.1.10	AT_NULL	74
4.9.1.11	AT_PAGESZ	74
4.9.1.12	AT_PHDR	74
4.9.1.13	AT_PHEM	74
4.9.1.14	AT_PHNUM	74
4.9.1.15	AT_STACKBASE	74
4.9.1.16	AT_SYSINFO_EHDR	75
4.9.1.17	AT_UCACHEBSIZE	75
4.9.1.18	EI_CLASS	75

4.9.1.19	EI_DATA	75
4.9.1.20	EI_MAG0	75
4.9.1.21	EI_MAG1	75
4.9.1.22	EI_MAG2	75
4.9.1.23	EI_MAG3	75
4.9.1.24	EI_NIDENT	76
4.9.1.25	EI_PAD	76
4.9.1.26	EI_VERSION	76
4.9.1.27	ELF32_ST_BIND	76
4.9.1.28	ELF32_ST_TYPE	76
4.9.1.29	ELF_MAGIC0	76
4.9.1.30	ELF_MAGIC1	76
4.9.1.31	ELF_MAGIC2	76
4.9.1.32	ELF_MAGIC3	77
4.9.1.33	ELFCLASS32	77
4.9.1.34	ELFCLASS64	77
4.9.1.35	ELFCLASSNONE	77
4.9.1.36	ELFDATA2LSB	77
4.9.1.37	ELFDATA2MSB	77
4.9.1.38	ELFDATANONE	77
4.9.1.39	EM_386	78
4.9.1.40	EM_AARCH64	78
4.9.1.41	EM_ALTERA_NIOS2	78
4.9.1.42	EM_ARM	78
4.9.1.43	EM_MICROBLAZE	78
4.9.1.44	EM_MIPS	78
4.9.1.45	EM_NONE	78
4.9.1.46	EM_OPENRISC	78
4.9.1.47	EM_SPARC	79
4.9.1.48	EM_SPARC32PLUS	79
4.9.1.49	EM_X86_64	79
4.9.1.50	ET_CORE	79
4.9.1.51	ET_DYN	79
4.9.1.52	ET_EXEC	79
4.9.1.53	ET_NONE	79
4.9.1.54	ET_REL	79
4.9.1.55	PF_R	79

4.9.1.56	PF_W	80
4.9.1.57	PF_X	80
4.9.1.58	PT_DYNAMIC	80
4.9.1.59	PT_INTERP	80
4.9.1.60	PT_LOAD	80
4.9.1.61	PT_NOTE	80
4.9.1.62	PT_NULL	80
4.9.1.63	PT_PHDR	80
4.9.1.64	PT_SHLIB	80
4.9.1.65	SHT_DYNAMIC	81
4.9.1.66	SHT_DYNSYM	81
4.9.1.67	SHT_HASH	81
4.9.1.68	SHT_NOBITS	81
4.9.1.69	SHT_NOTE	81
4.9.1.70	SHT_NULL	81
4.9.1.71	SHT_PROGBITS	81
4.9.1.72	SHT_REL	81
4.9.1.73	SHT_RELA	81
4.9.1.74	SHT_SHLIB	82
4.9.1.75	SHT_STRTAB	82
4.9.1.76	SHT_SYMTAB	82
4.9.1.77	STB_GLOBAL	82
4.9.1.78	STB_LOCAL	82
4.9.1.79	STB_WEAK	82
4.9.1.80	STN_UNDEF	82
4.9.1.81	STT_FILE	82
4.9.1.82	STT_FUNCTION	83
4.9.1.83	STT_NOTYPE	83
4.9.1.84	STT_OBJECT	83
4.9.1.85	STT_SECTION	83
4.9.2	Typedef Documentation	83
4.9.2.1	auxv_t	83
4.9.2.2	Elf32_Addr	83
4.9.2.3	Elf32_Half	83
4.9.2.4	Elf32_Off	83
4.9.2.5	Elf32_Sword	83
4.9.2.6	Elf32_Word	84

4.9.3	Function Documentation	84
4.9.3.1	elf_check	84
4.9.3.2	elf_load	85
4.9.3.3	elf_lookup_symbol	87
4.9.3.4	elf_setup_stack	88
4.10	include/hal/asm/boot.h File Reference	90
4.10.1	Macro Definition Documentation	91
4.10.1.1	BOOT_DATA_STRUCT	91
4.10.1.2	BOOT_E820_ENTRIES	91
4.10.1.3	BOOT_E820_MAP	91
4.10.1.4	BOOT_E820_MAP_END	91
4.10.1.5	BOOT_E820_MAP_SIZE	91
4.10.1.6	BOOT_MAGIC	91
4.10.1.7	BOOT_SETUP	91
4.10.1.8	BOOT_SETUP32_ADDR	91
4.10.1.9	BOOT_SETUP32_SIZE	91
4.10.1.10	BOOT_SETUP_ADDR	91
4.10.1.11	BOOT_SETUP_HEADER	91
4.10.1.12	BOOT_SETUP_MAGIC	92
4.10.1.13	BOOT_SETUP_SECTS	92
4.10.1.14	BOOT_SIGNATURE	92
4.10.1.15	BOOT_STACK_SIZE	92
4.10.1.16	BOOT_SYSIZE	92
4.11	include/hal/boot.h File Reference	92
4.11.1	Function Documentation	93
4.11.1.1	boot_info_check	93
4.11.1.2	boot_info_dump	94
4.11.1.3	get_boot_info	95
4.12	include/hal/asm/descriptors.h File Reference	95
4.12.1	Macro Definition Documentation	97
4.12.1.1	GDT_KERNEL_CODE	97
4.12.1.2	GDT_KERNEL_DATA	97
4.12.1.3	GDT_LENGTH	97
4.12.1.4	GDT_NULL	97
4.12.1.5	GDT_PER_CPU_DATA	97
4.12.1.6	GDT_TSS	97
4.12.1.7	GDT_USER_CODE	98

4.12.1.8	GDT_USER_DATA	98
4.12.1.9	GDT_USER_TLS_DATA	98
4.12.1.10	RPL_KERNEL	98
4.12.1.11	RPL_USER	98
4.12.1.12	SEG_FLAG_16BIT	98
4.12.1.13	SEG_FLAG_16BIT_GATE	98
4.12.1.14	SEG_FLAG_32BIT	98
4.12.1.15	SEG_FLAG_32BIT_GATE	99
4.12.1.16	SEG_FLAG_BUSY	99
4.12.1.17	SEG_FLAG_IN_BYTES	99
4.12.1.18	SEG_FLAG_IN_PAGES	99
4.12.1.19	SEG_FLAG_KERNEL	99
4.12.1.20	SEG_FLAG_NORMAL	99
4.12.1.21	SEG_FLAG_NORMAL_GATE	99
4.12.1.22	SEG_FLAG_NOSYSTEM	100
4.12.1.23	SEG_FLAG_PRESENT	100
4.12.1.24	SEG_FLAG_SYSTEM	100
4.12.1.25	SEG_FLAG_TSS	100
4.12.1.26	SEG_FLAG_USER	100
4.12.1.27	SEG_FLAGS_OFFSET	100
4.12.1.28	SEG_SELECTOR	100
4.12.1.29	SEG_TYPE_CALL_GATE	100
4.12.1.30	SEG_TYPE_CODE	101
4.12.1.31	SEG_TYPE_DATA	101
4.12.1.32	SEG_TYPE_INTERRUPT_GATE	101
4.12.1.33	SEG_TYPE_READ_ONLY	101
4.12.1.34	SEG_TYPE_TASK_GATE	101
4.12.1.35	SEG_TYPE_TRAP_GATE	101
4.12.1.36	SEG_TYPE_TSS	101
4.12.1.37	TSS_LIMIT	102
4.13	include/hal/descriptors.h File Reference	102
4.13.1	Macro Definition Documentation	103
4.13.1.1	GATE_DESCRIPTOR	103
4.13.1.2	PACK_DESCRIPTOR	103
4.13.1.3	SEG_DESCRIPTOR	103
4.14	include/hal/asm/e820.h File Reference	103
4.14.1	Macro Definition Documentation	104

4.14.1.1	E820_ACPI	104
4.14.1.2	E820_RAM	104
4.14.1.3	E820_RESERVED	104
4.14.1.4	E820_SMAP	104
4.15	include/hal/e820.h File Reference	104
4.15.1	Function Documentation	105
4.15.1.1	e820_dump	105
4.15.1.2	e820_is_available	106
4.15.1.3	e820_is_valid	106
4.15.1.4	e820_type_description	107
4.16	include/hal/asm/irq.h File Reference	107
4.16.1	Macro Definition Documentation	108
4.16.1.1	EXCEPTION_ALIGNMENT	108
4.16.1.2	EXCEPTION_BOUND	108
4.16.1.3	EXCEPTION_BREAK	108
4.16.1.4	EXCEPTION_DIV_ZERO	109
4.16.1.5	EXCEPTION_DOUBLE_FAULT	109
4.16.1.6	EXCEPTION_GENERAL_PROTECTION	109
4.16.1.7	EXCEPTION_INVALID_OP	109
4.16.1.8	EXCEPTION_INVALID_TSS	109
4.16.1.9	EXCEPTION_MACHINE_CHECK	109
4.16.1.10	EXCEPTION_MATH	109
4.16.1.11	EXCEPTION_NMI	109
4.16.1.12	EXCEPTION_NO_COPROC	109
4.16.1.13	EXCEPTION_OVERFLOW	110
4.16.1.14	EXCEPTION_PAGE_FAULT	110
4.16.1.15	EXCEPTION_SEGMENT_NOT_PRESENT	110
4.16.1.16	EXCEPTION_SIMD	110
4.16.1.17	EXCEPTION_STACK_SEGMENT	110
4.16.1.18	HAS_ERRCODE	110
4.16.1.19	IDT_FIRST_IRQ	110
4.16.1.20	IDT_IRQ_COUNT	110
4.16.1.21	IDT_VECTOR_COUNT	110
4.17	include/hal/asm/thread.h File Reference	111
4.17.1	Macro Definition Documentation	111
4.17.1.1	THREAD_CONTEXT_MASK	111
4.17.1.2	THREAD_CONTEXT_SIZE	111

4.18	include/hal/thread.h File Reference	112
4.18.1	Function Documentation	112
4.18.1.1	thread_context_switch	112
4.18.1.2	thread_page_create	113
4.18.1.3	thread_page_destroy	114
4.19	include/thread.h File Reference	115
4.19.1	Function Documentation	115
4.19.1.1	thread_create	115
4.19.1.2	thread_ready	116
4.19.1.3	thread_switch	116
4.19.1.4	thread_yield_from	117
4.20	include/hal/asm/vm.h File Reference	118
4.20.1	Macro Definition Documentation	119
4.20.1.1	VM_FLAG_ACCESSED	119
4.20.1.2	VM_FLAG_DIRTY	119
4.20.1.3	VM_FLAG_KERNEL	119
4.20.1.4	VM_FLAG_PRESENT	119
4.20.1.5	VM_FLAG_READ_ONLY	119
4.20.1.6	VM_FLAG_READ_WRITE	119
4.20.1.7	VM_FLAG_USER	120
4.21	include/hal/vm.h File Reference	120
4.21.1	Macro Definition Documentation	121
4.21.1.1	ADDR_4GB	121
4.21.1.2	EARLY_PHYS_TO_VIRT	121
4.21.1.3	EARLY_PTR_TO_PFADDR	121
4.21.1.4	EARLY_VIRT_TO_PHYS	121
4.21.2	Function Documentation	122
4.21.2.1	vm_boot_init	122
4.21.2.2	vm_change_flags	124
4.21.2.3	vm_create_addr_space	124
4.21.2.4	vm_create_initial_addr_space	125
4.21.2.5	vm_destroy_addr_space	125
4.21.2.6	vm_lookup_pfaddr	125
4.21.2.7	vm_map_early	126
4.21.2.8	vm_map_kernel	126
4.21.2.9	vm_map_user	126
4.21.2.10	vm_switch_addr_space	126

4.21.2.11 vm_unmap_kernel	127
4.21.2.12 vm_unmap_user	127
4.22 include/jinue/vm.h File Reference	127
4.23 include/jinue-common/asm/vm.h File Reference	128
4.23.1 Macro Definition Documentation	129
4.23.1.1 KLIMIT	129
4.23.1.2 PAGE_BITS	129
4.23.1.3 PAGE_MASK	129
4.23.1.4 PAGE_SIZE	129
4.23.1.5 STACK_BASE	130
4.23.1.6 STACK_SIZE	130
4.23.1.7 STACK_START	130
4.24 include/jinue-common/vm.h File Reference	130
4.24.1 Macro Definition Documentation	131
4.24.1.1 page_address_of	131
4.24.1.2 page_number_of	131
4.24.1.3 page_offset_of	131
4.25 include/hal/asm/x86.h File Reference	131
4.25.1 Macro Definition Documentation	132
4.25.1.1 X86_CR0_PG	132
4.25.1.2 X86_CR0_WP	133
4.25.1.3 X86_CR4_PAE	133
4.25.1.4 X86_CR4_PGE	133
4.25.1.5 X86_CR4_PSE	133
4.25.1.6 X86_PDE_PAGE_SIZE	133
4.25.1.7 X86_PTE_ACCESSED	133
4.25.1.8 X86_PTE_CACHE_DISABLE	133
4.25.1.9 X86_PTE_DIRTY	133
4.25.1.10 X86_PTE_GLOBAL	134
4.25.1.11 X86_PTE_NX	134
4.25.1.12 X86_PTE_PRESENT	134
4.25.1.13 X86_PTE_READ_WRITE	134
4.25.1.14 X86_PTE_USER	134
4.25.1.15 X86_PTE_WRITE_THROUGH	134
4.26 include/hal/x86.h File Reference	134
4.26.1 Typedef Documentation	136
4.26.1.1 msr_addr_t	136

4.26.2	Function Documentation	136
4.26.2.1	cli	136
4.26.2.2	cpuid	136
4.26.2.3	get_cr0	136
4.26.2.4	get_cr1	136
4.26.2.5	get_cr2	136
4.26.2.6	get_cr3	136
4.26.2.7	get_cr4	136
4.26.2.8	get_eflags	137
4.26.2.9	get_esp	137
4.26.2.10	get_gs_ptr	137
4.26.2.11	invalidate_tlb	137
4.26.2.12	lgdt	137
4.26.2.13	lidt	137
4.26.2.14	ltr	137
4.26.2.15	rdmsr	137
4.26.2.16	set_cr0	137
4.26.2.17	set_cr1	137
4.26.2.18	set_cr2	137
4.26.2.19	set_cr3	137
4.26.2.20	set_cr4	137
4.26.2.21	set_cs	137
4.26.2.22	set_data_segments	138
4.26.2.23	set_ds	138
4.26.2.24	set_eflags	138
4.26.2.25	set_es	138
4.26.2.26	set_fs	138
4.26.2.27	set_gs	138
4.26.2.28	set_ss	138
4.26.2.29	sti	138
4.26.2.30	wrmsr	138
4.27	include/hal/bootmem.h File Reference	139
4.27.1	Typedef Documentation	140
4.27.1.1	bootmem_t	140
4.27.2	Function Documentation	140
4.27.2.1	apply_mem_hole	140
4.27.2.2	bootmem_get_block	141

4.27.2.3	bootmem_init	141
4.27.2.4	new_ram_map_entry	143
4.27.3	Variable Documentation	144
4.27.3.1	boot_heap	144
4.27.3.2	bootmem_root	144
4.27.3.3	ram_map	144
4.28	include/hal/cpu.h File Reference	144
4.28.1	Macro Definition Documentation	146
4.28.1.1	CPU_EFLAGS_ID	146
4.28.1.2	CPU_FEATURE_CPUID	146
4.28.1.3	CPU_FEATURE_LOCAL_APIC	146
4.28.1.4	CPU_FEATURE_PAE	146
4.28.1.5	CPU_FEATURE_SYSCALL	147
4.28.1.6	CPU_FEATURE_SYSENTER	147
4.28.1.7	CPU_VENDOR_AMD	147
4.28.1.8	CPU_VENDOR_AMD_DW0	147
4.28.1.9	CPU_VENDOR_AMD_DW1	147
4.28.1.10	CPU_VENDOR_AMD_DW2	147
4.28.1.11	CPU_VENDOR_GENERIC	147
4.28.1.12	CPU_VENDOR_INTEL	147
4.28.1.13	CPU_VENDOR_INTEL_DW0	147
4.28.1.14	CPU_VENDOR_INTEL_DW1	148
4.28.1.15	CPU_VENDOR_INTEL_DW2	148
4.28.1.16	CPUID_EXT_FEATURE_SYSCALL	148
4.28.1.17	CPUID_FEATURE_APIC	148
4.28.1.18	CPUID_FEATURE_CLFLUSH	148
4.28.1.19	CPUID_FEATURE_FPU	148
4.28.1.20	CPUID_FEATURE_HTTP	148
4.28.1.21	CPUID_FEATURE_PAE	148
4.28.1.22	CPUID_FEATURE_SEP	148
4.28.1.23	MSR_EFER	149
4.28.1.24	MSR_FLAG_STAR_SCE	149
4.28.1.25	MSR_IA32_SYSENTER_CS	149
4.28.1.26	MSR_IA32_SYSENTER_EIP	149
4.28.1.27	MSR_IA32_SYSENTER_ESP	149
4.28.1.28	MSR_STAR	149
4.28.2	Function Documentation	149

4.28.2.1	cpu_detect_features	149
4.28.2.2	cpu_init_data	151
4.28.3	Variable Documentation	152
4.28.3.1	cpu_info	153
4.29	include/hal/cpu_data.h File Reference	153
4.29.1	Macro Definition Documentation	153
4.29.1.1	CPU_DATA_ALIGNMENT	154
4.30	include/hal/frame_pointer.h File Reference	154
4.30.1	Function Documentation	155
4.30.1.1	get_caller_fpointer	155
4.30.1.2	get_fpointer	155
4.30.1.3	get_program_counter	155
4.30.1.4	get_ret_addr	155
4.31	include/hal/hal.h File Reference	155
4.31.1	Function Documentation	155
4.31.1.1	hal_init	155
4.32	include/hal/interrupt.h File Reference	158
4.32.1	Function Documentation	159
4.32.1.1	dispatch_interrupt	159
4.32.2	Variable Documentation	160
4.32.2.1	idt	160
4.33	include/hal/io.h File Reference	161
4.33.1	Function Documentation	161
4.33.1.1	inb	161
4.33.1.2	inl	161
4.33.1.3	inw	161
4.33.1.4	outb	161
4.33.1.5	outl	162
4.33.1.6	outw	162
4.34	include/hal/kernel.h File Reference	162
4.34.1	Variable Documentation	162
4.34.1.1	kernel_region_top	162
4.35	include/hal/pfaddr.h File Reference	163
4.35.1	Macro Definition Documentation	164
4.35.1.1	ADDR_TO_PFADDR	164
4.35.1.2	PFADDR_CHECK	164
4.35.1.3	PFADDR_CHECK_4GB	164

4.35.1.4	PFADDR_TO_ADDR	164
4.36	include/jinue/pfaddr.h File Reference	164
4.37	include/jinue-common/pfaddr.h File Reference	165
4.37.1	Macro Definition Documentation	166
4.37.1.1	PFADDR_SHIFT	166
4.37.1.2	PFNULL	167
4.37.2	Typedef Documentation	167
4.37.2.1	pfaddr_t	167
4.38	include/hal/startup.h File Reference	167
4.38.1	Function Documentation	167
4.38.1.1	halt	167
4.39	include/hal/trap.h File Reference	168
4.39.1	Function Documentation	168
4.39.1.1	fast_amd_entry	168
4.39.1.2	fast_intel_entry	168
4.39.1.3	return_from_interrupt	168
4.39.2	Variable Documentation	168
4.39.2.1	syscall_method	168
4.40	include/hal/types.h File Reference	169
4.40.1	Macro Definition Documentation	170
4.40.1.1	msg_arg0	170
4.40.1.2	msg_arg1	170
4.40.1.3	msg_arg2	170
4.40.1.4	msg_arg3	170
4.40.2	Typedef Documentation	170
4.40.2.1	addr_t	170
4.40.2.2	cpu_data_t	170
4.40.2.3	e820_addr_t	171
4.40.2.4	e820_size_t	171
4.40.2.5	e820_type_t	171
4.40.2.6	pdpt_t	171
4.40.2.7	pte_t	171
4.40.2.8	seg_descriptor_t	171
4.40.2.9	seg_selector_t	171
4.41	include/jinue/types.h File Reference	172
4.42	include/jinue-common/asm/types.h File Reference	172
4.42.1	Macro Definition Documentation	172

4.42.1.1	KB	172
4.42.1.2	MB	172
4.43	include/jinue-common/types.h File Reference	173
4.44	include/types.h File Reference	173
4.44.1	Macro Definition Documentation	174
4.44.1.1	PROCESS_MAX_DESCRIPTOR	174
4.44.2	Typedef Documentation	174
4.44.2.1	thread_t	174
4.45	include/hal/vga.h File Reference	174
4.45.1	Macro Definition Documentation	176
4.45.1.1	VGA_COL	176
4.45.1.2	VGA_COLOR_BLACK	176
4.45.1.3	VGA_COLOR_BLUE	176
4.45.1.4	VGA_COLOR_BRIGHTBLUE	176
4.45.1.5	VGA_COLOR_BRIGHTCYAN	176
4.45.1.6	VGA_COLOR_BRIGHTGREEN	176
4.45.1.7	VGA_COLOR_BRIGHTMAGENTA	176
4.45.1.8	VGA_COLOR_BRIGHTRED	176
4.45.1.9	VGA_COLOR_BRIGHTWHITE	176
4.45.1.10	VGA_COLOR_BROWN	176
4.45.1.11	VGA_COLOR_CYAN	176
4.45.1.12	VGA_COLOR_DEFAULT	177
4.45.1.13	VGA_COLOR_ERASE	177
4.45.1.14	VGA_COLOR_GRAY	177
4.45.1.15	VGA_COLOR_GREEN	177
4.45.1.16	VGA_COLOR_MAGENTA	177
4.45.1.17	VGA_COLOR_RED	177
4.45.1.18	VGA_COLOR_WHITE	177
4.45.1.19	VGA_COLOR_YELLOW	177
4.45.1.20	VGA_CRTC_ADDR	177
4.45.1.21	VGA_CRTC_DATA	177
4.45.1.22	VGA_FB_FLAG_ACTIVE	178
4.45.1.23	VGA_LINE	178
4.45.1.24	VGA_LINES	178
4.45.1.25	VGA_MISC_OUT_RD	178
4.45.1.26	VGA_MISC_OUT_WR	178
4.45.1.27	VGA_TAB_WIDTH	178

4.45.1.28	VGA_TEXT_VID_BASE	178
4.45.1.29	VGA_TEXT_VID_SIZE	178
4.45.1.30	VGA_TEXT_VID_TOP	178
4.45.1.31	VGA_WIDTH	178
4.45.2	Typedef Documentation	179
4.45.2.1	vga_pos_t	179
4.45.3	Function Documentation	179
4.45.3.1	vga_clear	179
4.45.3.2	vga_get_color	179
4.45.3.3	vga_get_cursor_pos	179
4.45.3.4	vga_init	180
4.45.3.5	vga_print	180
4.45.3.6	vga_printn	181
4.45.3.7	vga_putc	181
4.45.3.8	vga_scroll	182
4.45.3.9	vga_set_base_addr	182
4.45.3.10	vga_set_color	182
4.45.3.11	vga_set_cursor_pos	183
4.46	include/hal/vm_pae.h File Reference	183
4.46.1	Function Documentation	184
4.46.1.1	vm_pae_boot_init	184
4.46.1.2	vm_pae_create_pdpt_cache	184
4.46.1.3	vm_pae_enable	185
4.47	include/hal/vm_private.h File Reference	185
4.47.1	Macro Definition Documentation	187
4.47.1.1	PAGE_DIRECTORY_OFFSET_OF	187
4.47.1.2	PAGE_TABLE_ENTRIES	187
4.47.1.3	PAGE_TABLE_MASK	187
4.47.1.4	PAGE_TABLE_OFFSET_OF	187
4.47.2	Function Documentation	187
4.47.2.1	vm_allocate_page_directory	187
4.47.2.2	vm_clone_page_directory	188
4.47.2.3	vm_destroy_page_directory	189
4.47.3	Variable Documentation	189
4.47.3.1	clear_pte	189
4.47.3.2	copy_pte	190
4.47.3.3	create_addr_space	190

4.47.3.4	create_initial_addr_space	190
4.47.3.5	destroy_addr_space	190
4.47.3.6	get_pte_flags	190
4.47.3.7	get_pte_pfaddr	190
4.47.3.8	get_pte_with_offset	190
4.47.3.9	global_page_tables	190
4.47.3.10	initial_addr_space	190
4.47.3.11	lookup_page_directory	191
4.47.3.12	page_directory_offset_of	191
4.47.3.13	page_table_entries	191
4.47.3.14	page_table_offset_of	191
4.47.3.15	set_pte	191
4.47.3.16	set_pte_flags	191
4.48	include/ipc.h File Reference	191
4.49	include/jinue/ipc.h File Reference	191
4.49.1	Function Documentation	192
4.49.1.1	jinue_create_ipc	192
4.49.1.2	jinue_receive	192
4.49.1.3	jinue_reply	192
4.49.1.4	jinue_send	192
4.50	include/jinue-common/asm/ipc.h File Reference	192
4.50.1	Macro Definition Documentation	193
4.50.1.1	JINUE_ARGS_PACK_BUFFER_SIZE	193
4.50.1.2	JINUE_ARGS_PACK_DATA_SIZE	193
4.50.1.3	JINUE_ARGS_PACK_N_DESC	193
4.50.1.4	JINUE_SEND_BUFFER_SIZE_OFFSET	193
4.50.1.5	JINUE_SEND_DATA_SIZE_OFFSET	194
4.50.1.6	JINUE_SEND_MAX_N_DESC	194
4.50.1.7	JINUE_SEND_MAX_SIZE	194
4.50.1.8	JINUE_SEND_N_DESC_BITS	194
4.50.1.9	JINUE_SEND_N_DESC_MASK	194
4.50.1.10	JINUE_SEND_N_DESC_OFFSET	194
4.50.1.11	JINUE_SEND_SIZE_BITS	194
4.50.1.12	JINUE_SEND_SIZE_MASK	194
4.51	include/jinue-common/ipc.h File Reference	195
4.51.1	Macro Definition Documentation	196
4.51.1.1	IPC_FLAG_NONE	196

4.51.1.2	IPC_FLAG_SYSTEM	196
4.51.1.3	JINUE_IPC_NONE	196
4.51.1.4	JINUE_IPC_PROC	196
4.51.1.5	JINUE_IPC_SYSTEM	196
4.51.2	Typedef Documentation	196
4.51.2.1	jinue_ipc_descriptor_t	196
4.51.3	Function Documentation	196
4.51.3.1	ipc_boot_init	196
4.51.3.2	ipc_get_proc_object	197
4.51.3.3	ipc_object_create	197
4.51.3.4	ipc_receive	198
4.51.3.5	ipc_reply	199
4.51.3.6	ipc_send	201
4.52	include/jinue/errno.h File Reference	203
4.53	include/jinue-common/errno.h File Reference	203
4.53.1	Macro Definition Documentation	204
4.53.1.1	JINUE_E2BIG	204
4.53.1.2	JINUE_EAGAIN	204
4.53.1.3	JINUE_EBADF	204
4.53.1.4	JINUE_EINVAL	204
4.53.1.5	JINUE_EIO	204
4.53.1.6	JINUE_EMORE	204
4.53.1.7	JINUE_ENOMEM	204
4.53.1.8	JINUE_ENOSYS	204
4.53.1.9	JINUE_EPERM	205
4.54	include/jinue/list.h File Reference	205
4.55	include/jinue-common/list.h File Reference	205
4.55.1	Macro Definition Documentation	206
4.55.1.1	jinue_cursor_entry	206
4.55.1.2	jinue_list_pop	206
4.55.1.3	JINUE_LIST_STATIC	206
4.55.1.4	jinue_node_entry	206
4.55.1.5	JINUE_OFFSETOF	207
4.55.2	Typedef Documentation	207
4.55.2.1	jinue_cursor_t	207
4.55.2.2	jinue_node_t	207
4.56	include/jinue/pfalloc.h File Reference	207

4.57	include/jinue-common/pfalloc.h File Reference	208
4.57.1	Macro Definition Documentation	208
4.57.1.1	KERNEL_PAGE_STACK_INIT	209
4.57.1.2	KERNEL_PAGE_STACK_SIZE	209
4.58	include/pfalloc.h File Reference	209
4.58.1	Macro Definition Documentation	210
4.58.1.1	pfalloc	210
4.58.1.2	pffree	210
4.58.2	Function Documentation	210
4.58.2.1	init_pfcache	210
4.58.2.2	pfalloc_early	211
4.58.2.3	pfalloc_from	211
4.58.2.4	pffree_to	212
4.58.3	Variable Documentation	212
4.58.3.1	global_pfcache	212
4.58.3.2	use_pfalloc_early	212
4.59	include/jinue/syscall.h File Reference	213
4.59.1	Function Documentation	213
4.59.1.1	jinue_call	213
4.59.1.2	jinue_call_raw	214
4.59.1.3	jinue_get_free_memory	214
4.59.1.4	jinue_get_syscall_implementation	214
4.59.1.5	jinue_get_syscall_implementation_name	214
4.59.1.6	jinue_get_thread_local_storage	214
4.59.1.7	jinue_set_thread_local_storage	214
4.59.1.8	jinue_thread_create	214
4.59.1.9	jinue_thread_exit	214
4.59.1.10	jinue_yield	214
4.60	include/jinue-common/asm/syscall.h File Reference	214
4.60.1	Macro Definition Documentation	215
4.60.1.1	SYSCALL_FUNCT_CONSOLE_PUTC	215
4.60.1.2	SYSCALL_FUNCT_CONSOLE_PUTS	215
4.60.1.3	SYSCALL_FUNCT_CREATE_IPC	215
4.60.1.4	SYSCALL_FUNCT_GET_FREE_MEMORY	215
4.60.1.5	SYSCALL_FUNCT_GET_THREAD_LOCAL_ADDR	216
4.60.1.6	SYSCALL_FUNCT_PROC_BASE	216
4.60.1.7	SYSCALL_FUNCT_RECEIVE	216

4.60.1.8	SYSCALL_FUNC_T_REPLY	216
4.60.1.9	SYSCALL_FUNC_T_SET_THREAD_LOCAL_ADDR	216
4.60.1.10	SYSCALL_FUNC_T_SYSCALL_METHOD	216
4.60.1.11	SYSCALL_FUNC_T_SYSTEM_BASE	216
4.60.1.12	SYSCALL_FUNC_T_THREAD_CREATE	217
4.60.1.13	SYSCALL_FUNC_T_THREAD_YIELD	217
4.60.1.14	SYSCALL_FUNC_T_USER_BASE	217
4.60.1.15	SYSCALL_IRQ	217
4.60.1.16	SYSCALL_METHOD_FAST_AMD	217
4.60.1.17	SYSCALL_METHOD_FAST_INTEL	217
4.60.1.18	SYSCALL_METHOD_INTR	217
4.61	include/jinue-common/syscall.h File Reference	218
4.62	include/syscall.h File Reference	218
4.62.1	Function Documentation	219
4.62.1.1	dispatch_syscall	219
4.63	include/kbd.h File Reference	222
4.63.1	Function Documentation	222
4.63.1.1	any_key	222
4.64	include/kstdc/assert.h File Reference	223
4.64.1	Macro Definition Documentation	223
4.64.1.1	assert	223
4.64.2	Function Documentation	224
4.64.2.1	__assert_failed	224
4.65	include/kstdc/stdarg.h File Reference	224
4.65.1	Macro Definition Documentation	225
4.65.1.1	va_arg	225
4.65.1.2	va_copy	225
4.65.1.3	va_end	225
4.65.1.4	va_start	225
4.65.2	Typedef Documentation	225
4.65.2.1	va_list	225
4.66	include/kstdc/stdbool.h File Reference	225
4.66.1	Macro Definition Documentation	226
4.66.1.1	__bool_true_false_are_defined	226
4.66.1.2	bool	226
4.66.1.3	false	226
4.66.1.4	true	226

4.67	include/kstdc/stddef.h File Reference	226
4.67.1	Macro Definition Documentation	226
4.67.1.1	NULL	226
4.67.1.2	offsetof	227
4.67.2	Typedef Documentation	227
4.67.2.1	ptrdiff_t	227
4.67.2.2	size_t	227
4.67.2.3	wchar_t	227
4.68	include/kstdc/stdint.h File Reference	227
4.68.1	Macro Definition Documentation	228
4.68.1.1	INT64_C	228
4.68.1.2	UINT64_C	228
4.68.2	Typedef Documentation	228
4.68.2.1	int16_t	228
4.68.2.2	int32_t	228
4.68.2.3	int64_t	228
4.68.2.4	int8_t	228
4.68.2.5	intptr_t	228
4.68.2.6	uint16_t	228
4.68.2.7	uint32_t	228
4.68.2.8	uint64_t	229
4.68.2.9	uint8_t	229
4.68.2.10	uintptr_t	229
4.69	include/kstdc/string.h File Reference	229
4.69.1	Function Documentation	229
4.69.1.1	memcpy	229
4.69.1.2	memset	230
4.69.1.3	strlen	230
4.70	include/object.h File Reference	230
4.70.1	Macro Definition Documentation	231
4.70.1.1	OBJECT_FLAG_DESTROYED	231
4.70.1.2	OBJECT_FLAG_NONE	231
4.70.1.3	OBJECT_REF_FLAG_CLOSED	231
4.70.1.4	OBJECT_REF_FLAG_NONE	231
4.70.1.5	OBJECT_REF_FLAG_OWNER	231
4.70.1.6	OBJECT_REF_FLAG_VALID	232
4.70.1.7	OBJECT_TYPE_IPC	232

4.70.1.8	OBJECT_TYPE_PROCESS	232
4.70.1.9	OBJECT_TYPE_THREAD	232
4.71	include/panic.h File Reference	232
4.71.1	Function Documentation	232
4.71.1.1	panic	232
4.72	include/printk.h File Reference	233
4.72.1	Function Documentation	234
4.72.1.1	print_hex_b	234
4.72.1.2	print_hex_l	234
4.72.1.3	print_hex_nibble	234
4.72.1.4	print_hex_q	234
4.72.1.5	print_hex_w	234
4.72.1.6	print_unsigned_int	234
4.72.1.7	printk	234
4.73	include/process.h File Reference	234
4.73.1	Function Documentation	235
4.73.1.1	process_boot_init	235
4.73.1.2	process_create	235
4.73.1.3	process_get_descriptor	236
4.73.1.4	process_unused_descriptor	236
4.74	include/slab.h File Reference	237
4.74.1	Macro Definition Documentation	238
4.74.1.1	SLAB_COMPACT	238
4.74.1.2	SLAB_DEFAULT_WORKING_SET	238
4.74.1.3	SLAB_DEFAULTS	238
4.74.1.4	SLAB_HWCACHE_ALIGN	239
4.74.1.5	SLAB_POISON	239
4.74.1.6	SLAB_POISON_ALIVE_VALUE	239
4.74.1.7	SLAB_POISON_DEAD_VALUE	239
4.74.1.8	SLAB_RED_ZONE	239
4.74.1.9	SLAB_RED_ZONE_VALUE	239
4.74.1.10	SLAB_SIZE	239
4.74.2	Typedef Documentation	239
4.74.2.1	slab_bufctl_t	239
4.74.2.2	slab_cache_t	239
4.74.2.3	slab_ctor_t	240
4.74.2.4	slab_t	240

4.74.3	Function Documentation	240
4.74.3.1	slab_cache_alloc	240
4.74.3.2	slab_cache_create	242
4.74.3.3	slab_cache_destroy	243
4.74.3.4	slab_cache_free	244
4.74.3.5	slab_cache_grow	246
4.74.3.6	slab_cache_reap	247
4.74.3.7	slab_cache_set_working_set	248
4.74.4	Variable Documentation	248
4.74.4.1	slab_cache_list	248
4.75	include/util.h File Reference	248
4.75.1	Macro Definition Documentation	249
4.75.1.1	ALIGN_END	249
4.75.1.2	ALIGN_START	249
4.75.1.3	alloc_backward	249
4.75.1.4	alloc_forward	249
4.75.1.5	OFFSET_OF	249
4.76	include/vm_alloc.h File Reference	249
4.76.1	Macro Definition Documentation	251
4.76.1.1	VM_ALLOC_BLOCK_MASK	251
4.76.1.2	VM_ALLOC_BLOCK_SIZE	251
4.76.1.3	VM_ALLOC_CANNOT_GROW	251
4.76.1.4	VM_ALLOC_EMPTY_STACK	251
4.76.1.5	VM_ALLOC_FULL_STACK	251
4.76.1.6	VM_ALLOC_IS_FREE	251
4.76.1.7	VM_ALLOC_IS_PARTIAL	251
4.76.1.8	VM_ALLOC_IS_USED	252
4.76.1.9	VM_ALLOC_STACK_ENTRIES	252
4.76.1.10	VM_ALLOC_WAS_FREE	252
4.76.1.11	VM_ALLOC_WAS_USED	252
4.76.2	Typedef Documentation	252
4.76.2.1	vm_alloc_t	252
4.76.2.2	vm_block_t	252
4.76.3	Function Documentation	252
4.76.3.1	vm_alloc	252
4.76.3.2	vm_alloc_add_region	253
4.76.3.3	vm_alloc_custom_block	254

4.76.3.4	vm_alloc_destroy	255
4.76.3.5	vm_alloc_free_block	256
4.76.3.6	vm_alloc_grow_single	257
4.76.3.7	vm_alloc_grow_stack	258
4.76.3.8	vm_alloc_init	259
4.76.3.9	vm_alloc_init_allocator	259
4.76.3.10	vm_alloc_low_latency	260
4.76.3.11	vm_alloc_partial_block	261
4.76.3.12	vm_alloc_unlink_block	263
4.76.3.13	vm_free	265
4.76.4	Variable Documentation	266
4.76.4.1	global_page_allocator	266
4.77	kernel/core/core.c File Reference	266
4.77.1	Function Documentation	266
4.77.1.1	kmain	266
4.78	kernel/core/elf.c File Reference	268
4.78.1	Function Documentation	269
4.78.1.1	elf_check	269
4.78.1.2	elf_load	271
4.78.1.3	elf_lookup_symbol	273
4.78.1.4	elf_setup_stack	274
4.79	kernel/core/ipc.c File Reference	276
4.79.1	Function Documentation	277
4.79.1.1	ipc_boot_init	277
4.79.1.2	ipc_get_proc_object	277
4.79.1.3	ipc_object_create	277
4.79.1.4	ipc_receive	278
4.79.1.5	ipc_reply	280
4.79.1.6	ipc_send	281
4.80	kernel/core/process.c File Reference	283
4.80.1	Function Documentation	284
4.80.1.1	process_boot_init	284
4.80.1.2	process_create	284
4.80.1.3	process_get_descriptor	285
4.80.1.4	process_unused_descriptor	285
4.81	kernel/core/syscall.c File Reference	285
4.81.1	Function Documentation	286

4.81.1.1	dispatch_syscall	286
4.82	kernel/core/thread.c File Reference	289
4.82.1	Function Documentation	290
4.82.1.1	thread_create	290
4.82.1.2	thread_ready	291
4.82.1.3	thread_switch	291
4.82.1.4	thread_yield_from	292
4.83	kernel/hal/thread.c File Reference	292
4.83.1	Function Documentation	293
4.83.1.1	thread_context_switch	293
4.83.1.2	thread_context_switch_stack	294
4.83.1.3	thread_page_create	294
4.83.1.4	thread_page_destroy	295
4.84	kernel/debug/console.c File Reference	296
4.84.1	Function Documentation	296
4.84.1.1	console_init	296
4.84.1.2	console_print	297
4.84.1.3	console_printn	297
4.84.1.4	console_putc	298
4.85	kernel/debug/debug.c File Reference	298
4.85.1	Function Documentation	299
4.85.1.1	dump_call_stack	299
4.86	kernel/debug/kbd.c File Reference	300
4.86.1	Function Documentation	301
4.86.1.1	any_key	301
4.87	kernel/debug/panic.c File Reference	301
4.87.1	Function Documentation	302
4.87.1.1	panic	302
4.88	kernel/hal/boot.c File Reference	303
4.88.1	Function Documentation	304
4.88.1.1	boot_info_check	304
4.88.1.2	boot_info_dump	305
4.88.1.3	get_boot_info	306
4.88.2	Variable Documentation	306
4.88.2.1	boot_info	306
4.89	kernel/hal/bootmem.c File Reference	306
4.89.1	Function Documentation	307

4.89.1.1	apply_mem_hole	307
4.89.1.2	bootmem_get_block	308
4.89.1.3	bootmem_init	309
4.89.1.4	new_ram_map_entry	311
4.89.2	Variable Documentation	311
4.89.2.1	boot_heap	311
4.89.2.2	bootmem_root	312
4.89.2.3	ram_map	312
4.90	kernel/hal/cpu.c File Reference	312
4.90.1	Function Documentation	313
4.90.1.1	cpu_detect_features	313
4.90.1.2	cpu_init_data	315
4.90.2	Variable Documentation	316
4.90.2.1	cpu_info	316
4.91	kernel/hal/e820.c File Reference	316
4.91.1	Function Documentation	317
4.91.1.1	e820_dump	317
4.91.1.2	e820_is_available	318
4.91.1.3	e820_is_valid	318
4.91.1.4	e820_type_description	318
4.92	kernel/hal/hal.c File Reference	319
4.92.1	Function Documentation	320
4.92.1.1	hal_init	320
4.92.2	Variable Documentation	323
4.92.2.1	kernel_region_top	323
4.92.2.2	syscall_method	323
4.93	kernel/hal/interrupt.c File Reference	323
4.93.1	Function Documentation	323
4.93.1.1	dispatch_interrupt	324
4.94	kernel/hal/vga.c File Reference	324
4.94.1	Function Documentation	325
4.94.1.1	vga_clear	325
4.94.1.2	vga_get_color	326
4.94.1.3	vga_get_cursor_pos	326
4.94.1.4	vga_init	326
4.94.1.5	vga_print	327
4.94.1.6	vga_printn	327

4.94.1.7	vga_putc	328
4.94.1.8	vga_scroll	328
4.94.1.9	vga_set_base_addr	329
4.94.1.10	vga_set_color	329
4.94.1.11	vga_set_cursor_pos	329
4.95	kernel/hal/vm.c File Reference	330
4.95.1	Function Documentation	331
4.95.1.1	vm_allocate_page_directory	331
4.95.1.2	vm_boot_init	332
4.95.1.3	vm_change_flags	334
4.95.1.4	vm_clone_page_directory	335
4.95.1.5	vm_create_addr_space	336
4.95.1.6	vm_create_initial_addr_space	336
4.95.1.7	vm_destroy_addr_space	336
4.95.1.8	vm_destroy_page_directory	336
4.95.1.9	vm_lookup_pfaddr	337
4.95.1.10	vm_map_early	337
4.95.1.11	vm_map_kernel	338
4.95.1.12	vm_map_user	338
4.95.1.13	vm_switch_addr_space	338
4.95.1.14	vm_unmap	338
4.95.1.15	vm_unmap_kernel	339
4.95.1.16	vm_unmap_user	340
4.95.1.17	vm_x86_create_initial_addr_space	340
4.95.2	Variable Documentation	340
4.95.2.1	clear_pte	340
4.95.2.2	copy_pte	340
4.95.2.3	create_addr_space	341
4.95.2.4	create_initial_addr_space	341
4.95.2.5	destroy_addr_space	341
4.95.2.6	get_pte_flags	341
4.95.2.7	get_pte_pfaddr	341
4.95.2.8	get_pte_with_offset	341
4.95.2.9	global_page_allocator	341
4.95.2.10	global_page_tables	341
4.95.2.11	initial_addr_space	341
4.95.2.12	lookup_page_directory	342

4.95.2.13	page_directory_offset_of	342
4.95.2.14	page_table_entries	342
4.95.2.15	page_table_offset_of	342
4.95.2.16	set_pte	342
4.95.2.17	set_pte_flags	342
4.96	kernel/hal/vm_pae.c File Reference	342
4.96.1	Macro Definition Documentation	344
4.96.1.1	PDPT_BITS	344
4.96.1.2	PDPT_ENTRIES	344
4.96.2	Function Documentation	344
4.96.2.1	vm_pae_boot_init	344
4.96.2.2	vm_pae_create_pdpt_cache	344
4.96.2.3	vm_pae_enable	345
4.96.3	Variable Documentation	345
4.96.3.1	initial_pdpt	345
4.97	kernel/kstdc/assert.c File Reference	345
4.97.1	Function Documentation	346
4.97.1.1	__assert_failed	346
4.98	kernel/kstdc/string.c File Reference	347
4.98.1	Function Documentation	347
4.98.1.1	memcpy	347
4.98.1.2	memset	348
4.98.1.3	strlen	348
4.99	kernel/mem/pfalloc.c File Reference	348
4.99.1	Function Documentation	349
4.99.1.1	init_pfcache	349
4.99.1.2	pfalloc_early	350
4.99.1.3	pfalloc_from	350
4.99.1.4	pf_free_to	351
4.99.2	Variable Documentation	351
4.99.2.1	global_pfcache	351
4.99.2.2	use_pfalloc_early	351
4.100	kernel/mem/slab.c File Reference	352
4.100.1	Function Documentation	352
4.100.1.1	slab_cache_alloc	352
4.100.1.2	slab_cache_create	354
4.100.1.3	slab_cache_destroy	356

4.100.1.4 slab_cache_free	357
4.100.1.5 slab_cache_grow	358
4.100.1.6 slab_cache_reap	360
4.100.1.7 slab_cache_set_working_set	360
4.100.2 Variable Documentation	360
4.100.2.1 slab_cache_list	360
4.101 kernel/mem/vm_alloc.c File Reference	361
4.101.1 Detailed Description	362
4.101.2 Function Documentation	362
4.101.2.1 vm_alloc	362
4.101.2.2 vm_alloc_add_region	363
4.101.2.3 vm_alloc_custom_block	364
4.101.2.4 vm_alloc_destroy	365
4.101.2.5 vm_alloc_free_block	366
4.101.2.6 vm_alloc_grow_single	367
4.101.2.7 vm_alloc_grow_stack	368
4.101.2.8 vm_alloc_init	369
4.101.2.9 vm_alloc_init_allocator	369
4.101.2.10 vm_alloc_low_latency	370
4.101.2.11 vm_alloc_partial_block	371
4.101.2.12 vm_alloc_unlink_block	373
4.101.2.13 vm_free	375

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

addr_space_t	7
boot_info_t	8
bootmem_t	11
cpu_data_t	11
cpu_info_t	13
e820_t	14
Elf32_auxv_t	15
Elf32_Ehdr	16
Elf32_Phdr	18
Elf32_Shdr	19
Elf32_Sym	21
elf_info_t	22
elf_symbol_t	24
ipc_t	24
jinue_list_t	26
jinue_message_t	27
jinue_node_t	27
jinue_reply_t	28
jinue_syscall_args_t	29
kernel_context_t	30
memory_block_t	31
message_info_t	31
object_header_t	32
object_ref_t	33
pdpt_t	34
pfcache_t	35
process_t	35
pseudo_descriptor_t	37
pte_t	37
slab_bufctl_t	38
slab_cache_t	39
slab_t	42
thread_context_t	43
thread_t	44

trapframe_t	46
tss_t	49
vm_alloc_t	52
vm_block_t	54
x86_cpuid_regs_t	57

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

include/ ascii.h	59
include/ console.h	60
include/ core.h	63
include/ debug.h	66
include/ elf.h	67
include/ ipc.h	191
include/ kbd.h	222
include/ object.h	230
include/ panic.h	232
include/ pfalloc.h	209
include/ printk.h	233
include/ process.h	234
include/ slab.h	237
include/ syscall.h	218
include/ thread.h	115
include/ types.h	173
include/ util.h	248
include/ vm_alloc.h	249
include/hal/ boot.h	92
include/hal/ bootmem.h	139
include/hal/ cpu.h	144
include/hal/ cpu_data.h	153
include/hal/ descriptors.h	102
include/hal/ e820.h	104
include/hal/ frame_pointer.h	154
include/hal/ hal.h	155
include/hal/ interrupt.h	158
include/hal/ io.h	161
include/hal/ kernel.h	162
include/hal/ pfaddr.h	163
include/hal/ startup.h	167
include/hal/ thread.h	112
include/hal/ trap.h	168
include/hal/ types.h	169

include/hal/ vga.h	174
include/hal/ vm.h	120
include/hal/ vm_pae.h	183
include/hal/ vm_private.h	185
include/hal/ x86.h	134
include/hal/asm/ boot.h	90
include/hal/asm/ descriptors.h	95
include/hal/asm/ e820.h	103
include/hal/asm/ irq.h	107
include/hal/asm/ thread.h	111
include/hal/asm/ vm.h	118
include/hal/asm/ x86.h	131
include/jinue-common/ console.h	60
include/jinue-common/ elf.h	68
include/jinue-common/ errno.h	203
include/jinue-common/ ipc.h	195
include/jinue-common/ list.h	205
include/jinue-common/ pfaddr.h	165
include/jinue-common/ pfalloc.h	208
include/jinue-common/ syscall.h	218
include/jinue-common/ types.h	173
include/jinue-common/ vm.h	130
include/jinue-common/asm/ ipc.h	192
include/jinue-common/asm/ syscall.h	214
include/jinue-common/asm/ types.h	172
include/jinue-common/asm/ vm.h	128
include/jinue/ console.h	60
include/jinue/ elf.h	67
include/jinue/ errno.h	203
include/jinue/ ipc.h	191
include/jinue/ list.h	205
include/jinue/ pfaddr.h	164
include/jinue/ pfalloc.h	207
include/jinue/ syscall.h	213
include/jinue/ types.h	172
include/jinue/ vm.h	127
include/kstdc/ assert.h	223
include/kstdc/ stdarg.h	224
include/kstdc/ stdbool.h	225
include/kstdc/ stddef.h	226
include/kstdc/ stdint.h	227
include/kstdc/ string.h	229
kernel/core/ core.c	266
kernel/core/ elf.c	268
kernel/core/ ipc.c	276
kernel/core/ process.c	283
kernel/core/ syscall.c	285
kernel/core/ thread.c	289
kernel/debug/ console.c	296
kernel/debug/ debug.c	298
kernel/debug/ kbd.c	300
kernel/debug/ panic.c	301
kernel/hal/ boot.c	303
kernel/hal/ bootmem.c	306

kernel/hal/ cpu.c	312
kernel/hal/ e820.c	316
kernel/hal/ hal.c	319
kernel/hal/ interrupt.c	323
kernel/hal/ thread.c	292
kernel/hal/ vga.c	324
kernel/hal/ vm.c	330
kernel/hal/ vm_pae.c	342
kernel/kstdc/ assert.c	345
kernel/kstdc/ string.c	347
kernel/mem/ pfalloc.c	348
kernel/mem/ slab.c	352
kernel/mem/ vm_alloc.c	
Virtual memory allocator	361

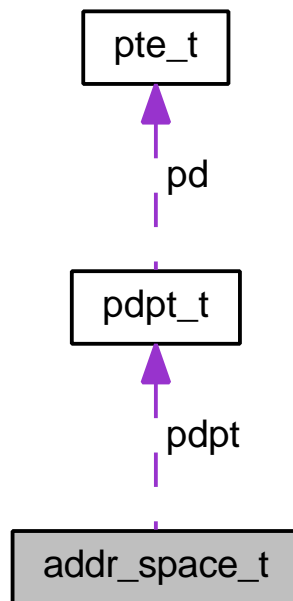
Chapter 3

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()`.

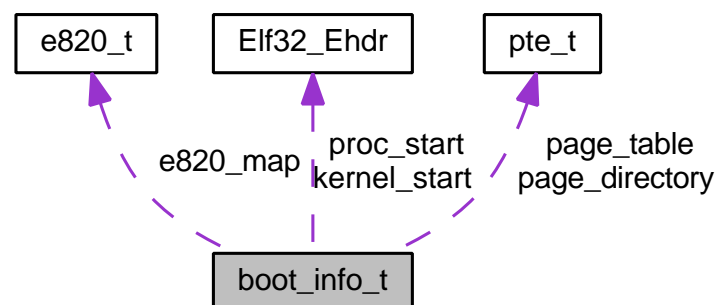
The documentation for this struct was generated from the following file:

- `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().

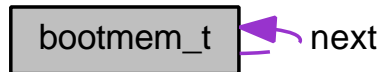
The documentation for this struct was generated from the following file:

- 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()`.

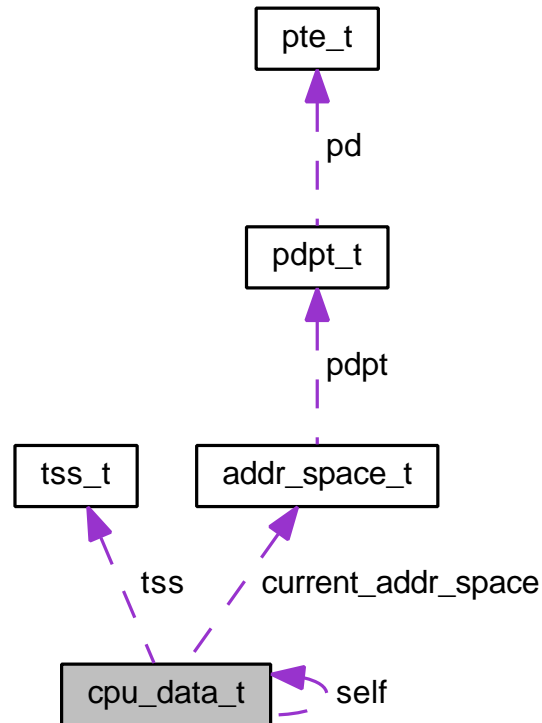
The documentation for this struct was generated from the following file:

- `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().

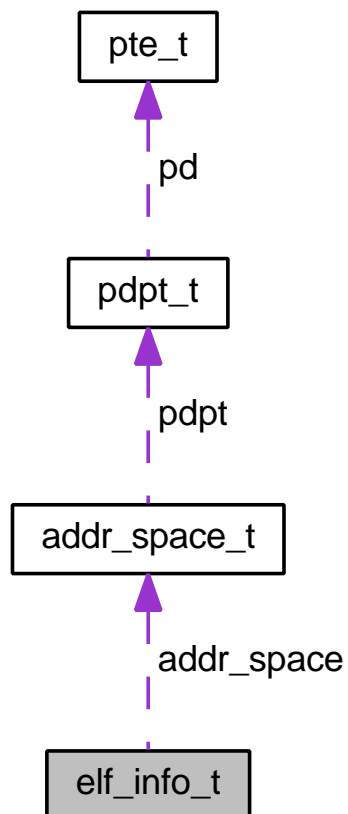
The documentation for this struct was generated from the following file:

- 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()`.

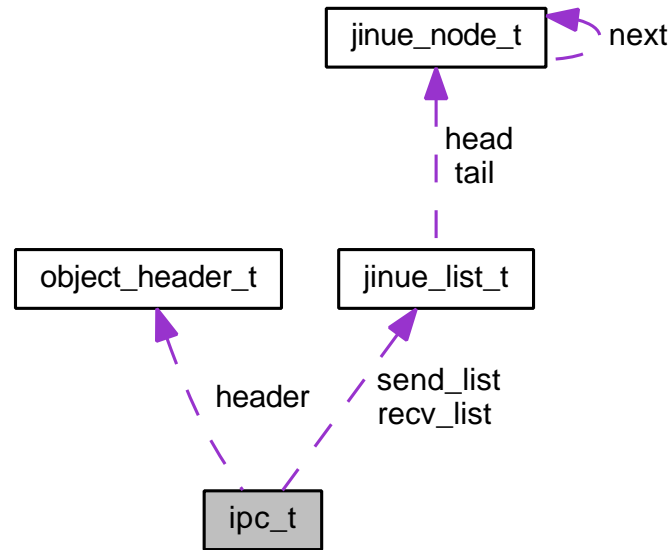
The documentation for this struct was generated from the following file:

- `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 rcv_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::rcv_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()`.

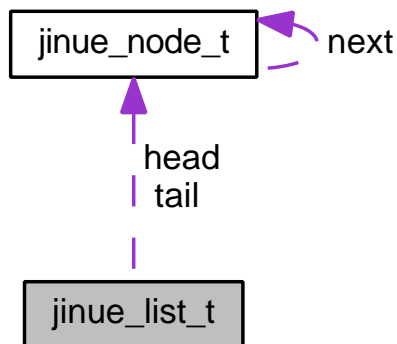
The documentation for this struct was generated from the following file:

- 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.

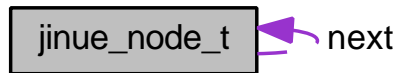
The documentation for this struct was generated from the following file:

- 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().

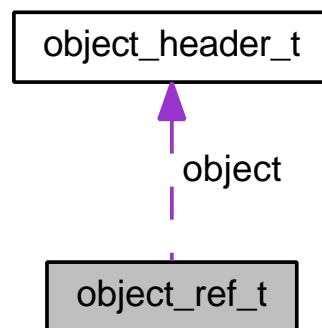
The documentation for this struct was generated from the following file:

- 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.

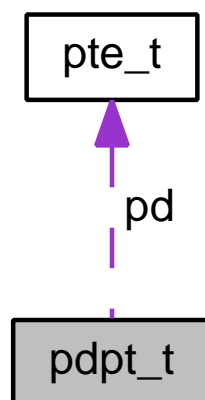
Referenced by `dispatch_syscall()`.

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().

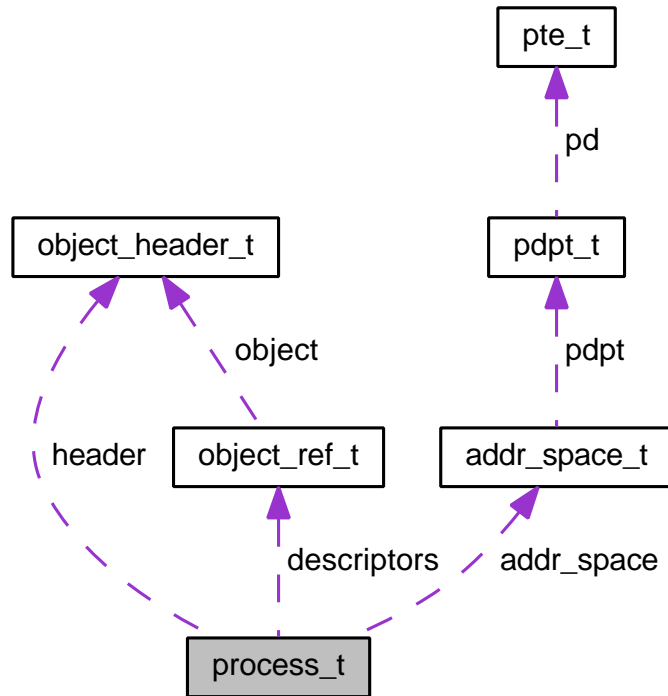
The documentation for this struct was generated from the following file:

- 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_DESCRIPTOR`]

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_DESCRIPTOR`]

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`.

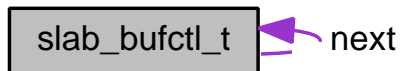
The documentation for this struct was generated from the following files:

- `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()`.

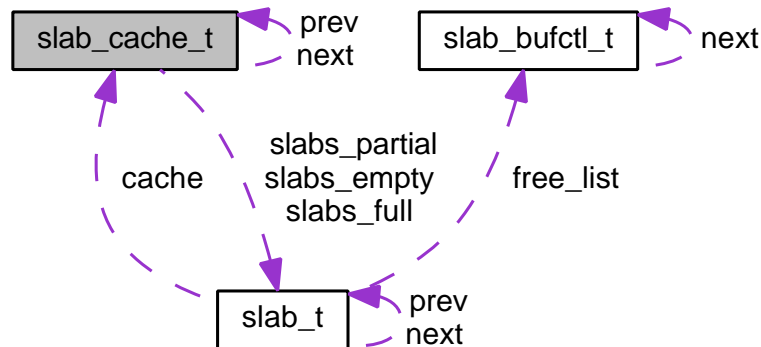
The documentation for this struct was generated from the following file:

- 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()`.

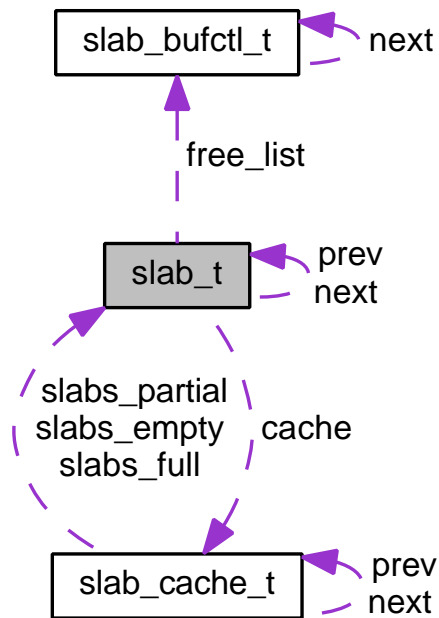
The documentation for this struct was generated from the following file:

- `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**
- struct `slab_cache_t` * **cache**
- unsigned int **obj_count**
- `size_t` **colour**
- struct `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()`.

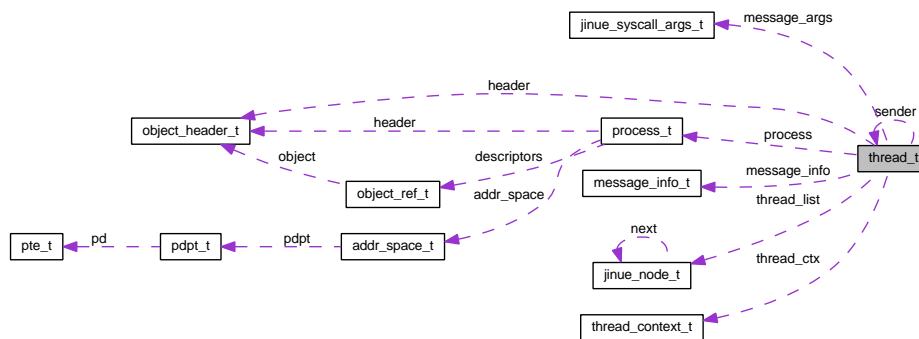
The documentation for this struct was generated from the following file:

- `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().

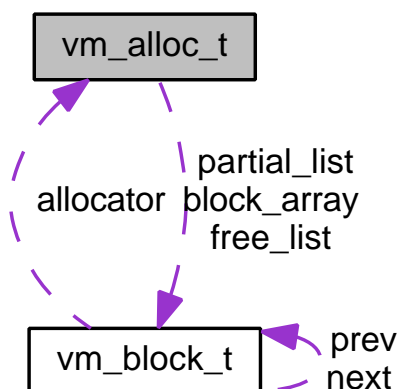
The documentation for this struct was generated from the following file:

- 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()`.

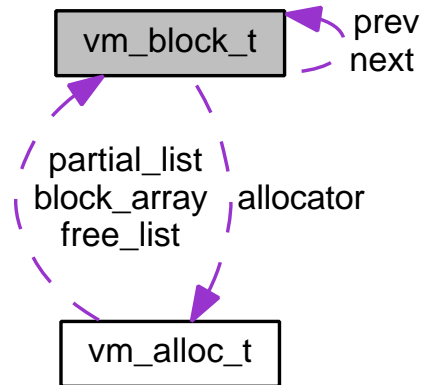
The documentation for this struct was generated from the following file:

- `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:

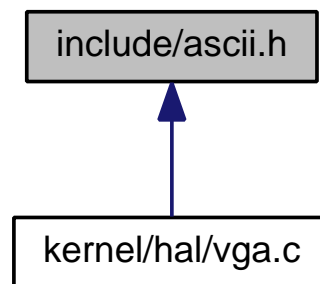
- `include/hal/x86.h`

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

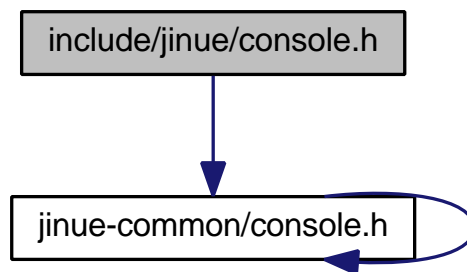
Definition at line 39 of file ascii.h.

4.2 include/console.h File Reference

4.3 include/jinue/console.h File Reference

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

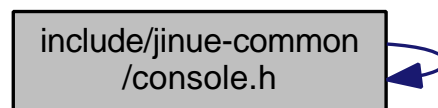
Include dependency graph for console.h:



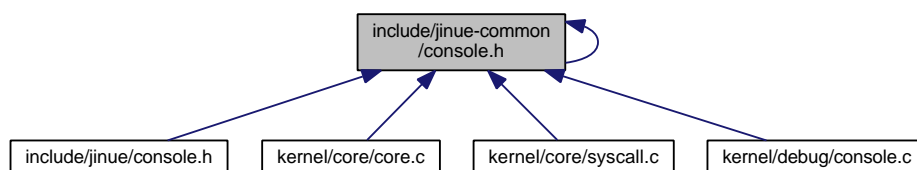
4.4 include/jinue-common/console.h File Reference

```
#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_printn** (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)

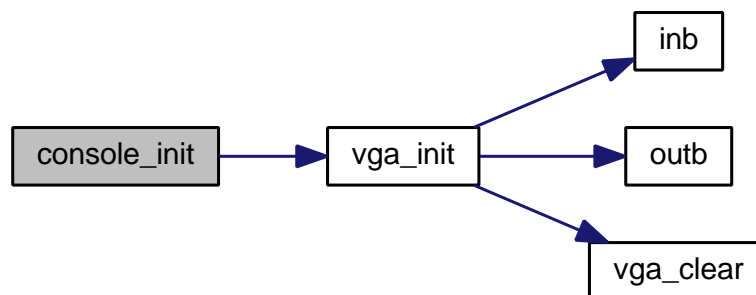
Definition at line 37 of file console.c.

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.

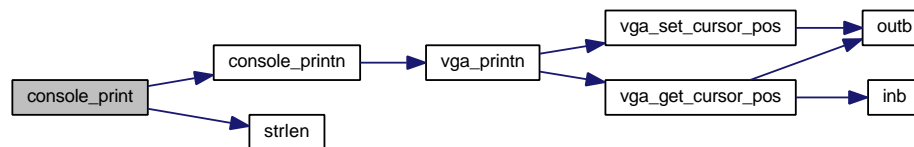
References console_printn(), and strlen().

```

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().

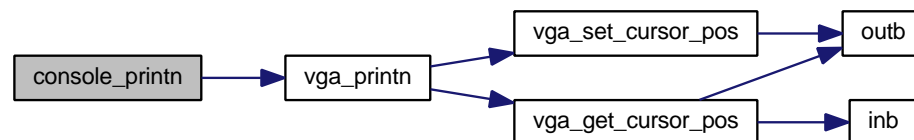
Referenced by console_print(), and dispatch_syscall().

```

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().

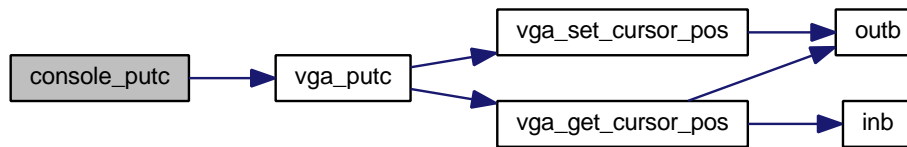
Referenced by dispatch_syscall().

```

45      {
46  vga_putc(c);
47  }

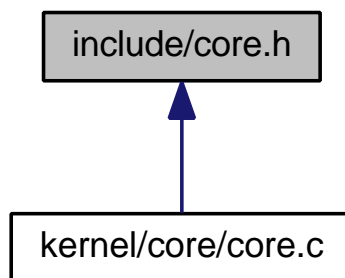
```

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`.

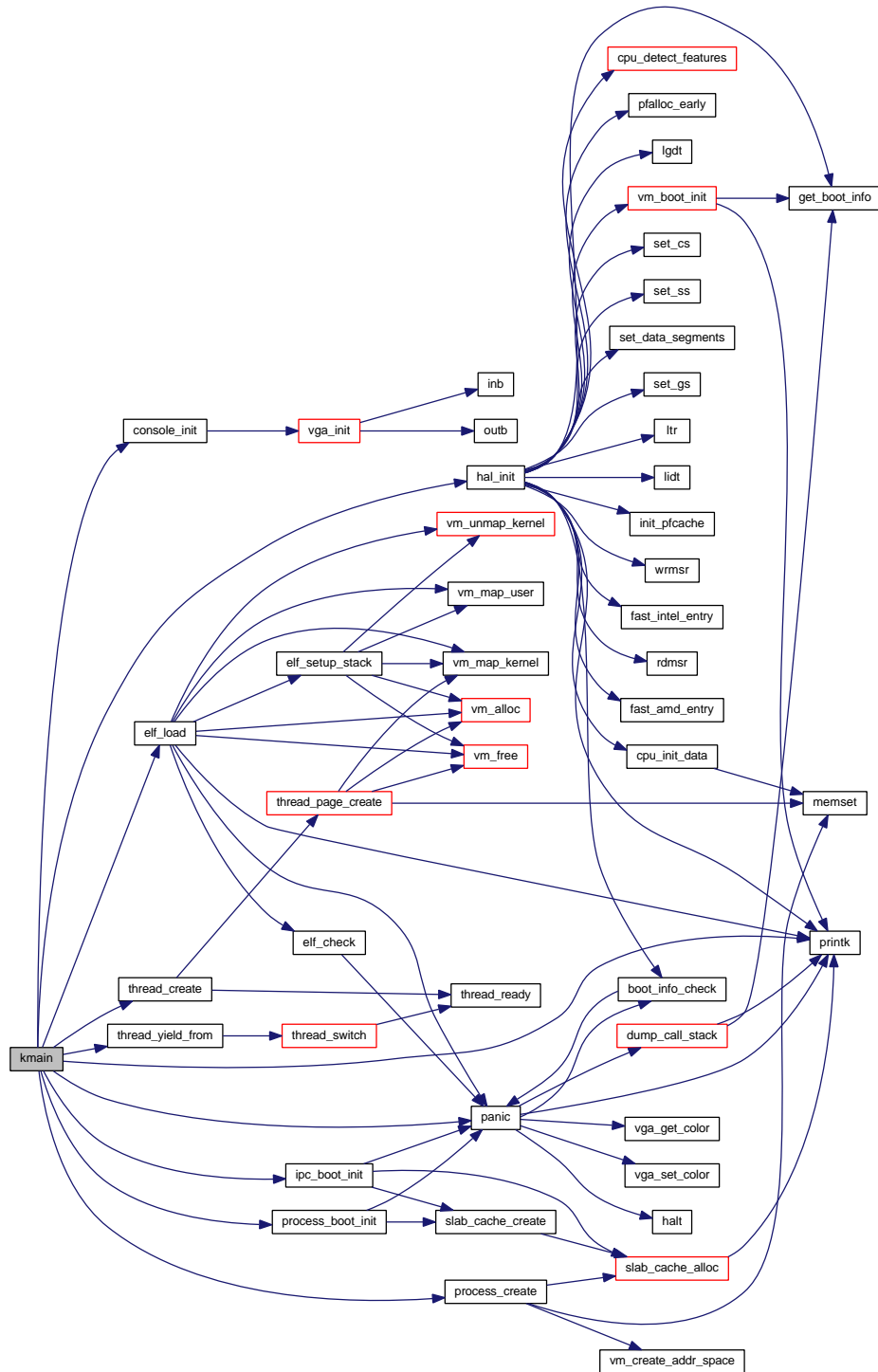
References `process_t::addr_space`, `console_init()`, `elf_load()`, `elf_info_t::entry`, `hal_init()`, `ipc_boot_init()`, `NULL`, `panic()`, `printk()`, `process_boot_init()`, `process_create()`, `elf_info_t::stack_addr`, `thread_create()`, and `thread_yield_from()`.

```

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();
  
```

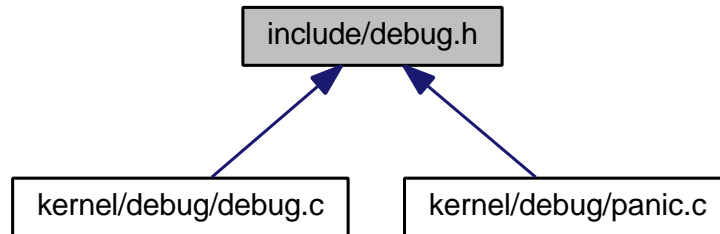
```
86     elf_load(&elf_info, elf, &process->addr_space);
87
88     /* create initial thread */
89     thread_t *thread = thread_create(
90         process,
91         elf_info.entry,
92         elf_info.stack_addr);
93
94     if(thread == NULL) {
95         panic("Could not create initial thread.");
96     }
97
98     /* start process manager
99     *
100     * We switch from NULL since this is the first thread. */
101     thread_yield_from(
102         NULL,
103         false,      /* don't block */
104         false);     /* don't destroy */
105                   /* just be nice */
106
107     /* should never happen */
108     panic("thread_yield_from() returned in kmain()");
109 }
```


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()`.

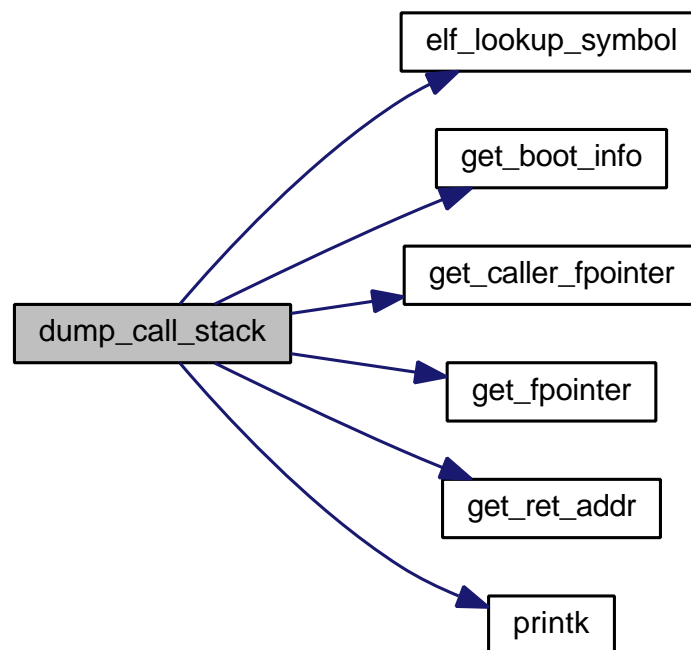
```

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(

```

```
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:

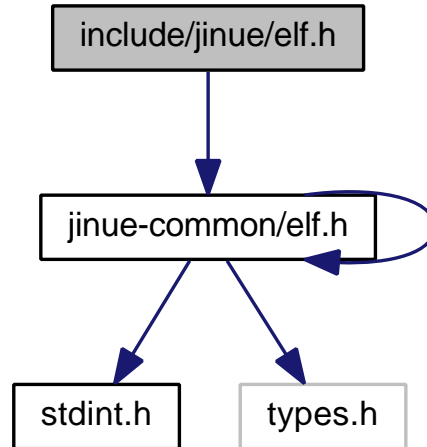


4.7 include/elf.h File Reference

4.8 include/jinue/elf.h File Reference

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

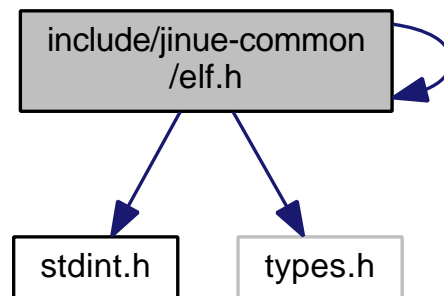
Include dependency graph for elf.h:



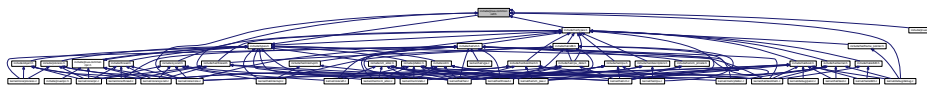
4.9 include/jinue-common/elf.h File Reference

```
#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_EXECD 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_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().

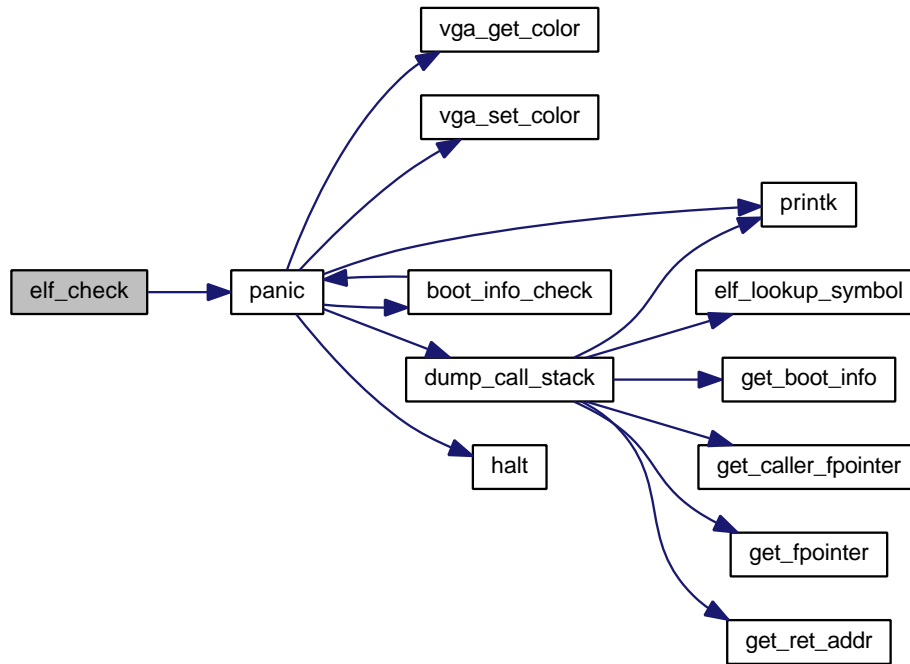
Referenced by elf_load().

```

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 */
52     if(elf->e_ident[EI_CLASS] != ELFCLASS32) {
53         panic("Bad file class");
54     }
55
56     /* check: endianness */
57     if(elf->e_ident[EI_DATA] != ELFDATA2LSB) {
58         panic("Bad endianness");
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 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().

```

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;
  
```

```

117     info->at_phent      = elf->e_phentsize;
118     info->addr_space    = addr_space;
119     info->entry         = (addr_t)elf->e_entry;
120
121     /* temporary page for copies */
122     dest_page = (char *)vm_alloc(global_page_allocator);
123
124     for(idx = 0; idx < elf->e_phnum; ++idx) {
125         if(phdr[idx].p_type != PT_LOAD) {
126             continue;
127         }
128
129         /* check that the segment is not in the region reserved for kernel use */
130         if(! user_buffer_check((void *)phdr[idx].p_vaddr, phdr[idx].p_memsz)) {
131             panic("process manager memory layout -- address of segment too low");
132         }
133
134         /* set start and end addresses for mapping and copying */
135         file_ptr = (char *)elf + phdr[idx].p_offset;
136         vptr     = (char *)phdr[idx].p_vaddr;
137         vend     = vptr + phdr[idx].p_memsz; /* limit for padding */
138         vfind    = vptr + phdr[idx].p_filesz; /* limit for copy */
139
140         /* align on page boundaries, be inclusive,
141            note that vfind is not aligned */
142         file_ptr = (char *) ( (uintptr_t)file_ptr & ~PAGE_MASK );
143         vptr     = (char *) ( (uintptr_t)vptr & ~PAGE_MASK );
144
145         if(page_offset_of(vend) != 0) {
146             vend = (char *) ( (uintptr_t)vend & ~PAGE_MASK );
147             vend += PAGE_SIZE;
148         }
149
150         /* copy if we have to */
151         if( (phdr[idx].p_flags & PF_W) || (phdr[idx].p_filesz != phdr[idx].
p_memsz) ) {
152             while(vptr < vend) {
153                 /* start of this page and next page */
154                 vpage = (addr_t)vptr;
155                 vnext = vptr + PAGE_SIZE;
156
157                 /* allocate and map the new page */
158                 page = pfalloc();
159                 vm_map_kernel((addr_t)dest_page, page, VM_FLAG_READ_WRITE);
160
161                 dest = dest_page;
162
163                 /* copy */
164                 stop = vnext;
165                 if(stop > vfind) {
166                     stop = vfind;
167                 }
168
169                 while(vptr < stop) {
170                     *(dest++) = *(file_ptr++);
171                     ++vptr;
172                 }
173
174                 /* pad */
175                 while(vptr < vnext) {
176                     *(dest++) = 0;
177                     ++vptr;
178                 }
179
180                 /* set flags */
181                 if(phdr[idx].p_flags & PF_W) {
182                     flags = VM_FLAG_READ_WRITE;
183                 }
184                 else {
185                     flags = VM_FLAG_READ_ONLY;
186                 }
187
188                 /* undo temporary mapping and map page in proper address
189                    * space */
190                 vm_unmap_kernel((addr_t)dest_page);
191                 vm_map_user(addr_space, (addr_t)vpage, page, flags);
192             }
193         }
194     }
195     else {
196         while(vptr < vend) {
197             /* perform mapping */

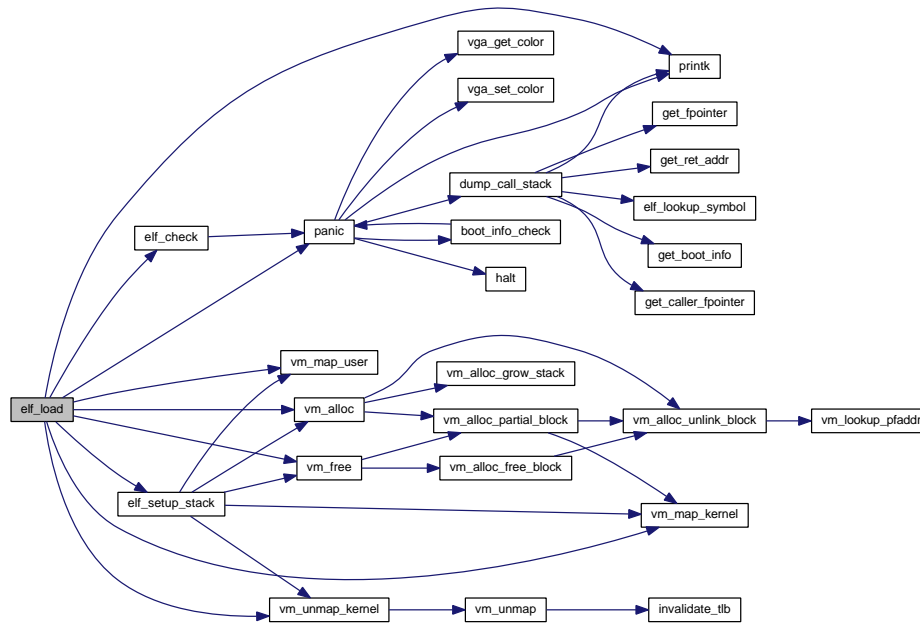
```

```

199         vm_map_user(addr_space, (addr_t)vptr, EARLY_PTR_TO_PFADDR(file_ptr),
VM_FLAG_READ_ONLY);
200
201         vptr    += PAGE_SIZE;
202         file_ptr += PAGE_SIZE;
203     }
204 }
205 }
206
207 vm_free(global_page_allocator, (addr_t)dest_page);
208
209 elf_setup_stack(info);
210
211 printk("ELF binary loaded.\n");
212 }

```

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()`.

```

288     {
289
290     int    idx;
291     size_t symbol_entry_size;
292     size_t symbol_table_size;
293
294     const char *elf_file      = elf_file_bytes(elf_header);
295     const char *symbols_table = NULL;
296     const char *string_table  = NULL;
297
298     for(idx = 0; idx < elf_header->e_shnum; ++idx) {
299         const Elf32_Shdr *section_header = elf_get_section_header(elf_header, idx);

```

```

300
301     if(section_header->sh_type == SHT_SYMTAB) {
302         symbols_table = &elf_file[section_header->sh_offset];
303         symbol_entry_size = section_header->sh_entsize;
304         symbol_table_size = section_header->sh_size;
305
306         const Elf32_Shdr *string_section_header = elf_get_section_header(
307             elf_header,
308             section_header->sh_link);
309
310         string_table = &elf_file[string_section_header->sh_offset];
311
312         break;
313     }
314 }
315
316 if(symbols_table == NULL) {
317     /* no symbol table */
318     return -1;
319 }
320
321 const char *symbol = symbols_table;
322
323 while(symbol < symbols_table + symbol_table_size) {
324     const Elf32_Sym *symbol_header = (const Elf32_Sym *)symbol;
325
326     if(ELF32_ST_TYPE(symbol_header->st_info) == type) {
327         Elf32_Addr lookup_addr = (Elf32_Addr)addr;
328         Elf32_Addr start = symbol_header->st_value;
329         Elf32_Addr end = start + symbol_header->st_size;
330
331         if(lookup_addr >= start && lookup_addr < end) {
332             result->addr = symbol_header->st_value;
333             result->name = &string_table[symbol_header->st_name];
334
335             return 0;
336         }
337     }
338
339     symbol += symbol_entry_size;
340 }
341
342 /* not found */
343 return -1;
344 }

```

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_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
218     /* initial stack allocation */
219     for(vpage = (addr_t)STACK_START; vpage < (addr_t)STACK_BASE; vpage +=
220     PAGE_SIZE) {
221         page = pfalloc();
222         vm_map_user(info->addr_space, vpage, page, VM_FLAG_READ_WRITE);
223     }
224
225     /* At this point, page has the address of the stack's top-most page frame,
226     * which is the one in which we are about to copy the auxiliary vectors. Map

```

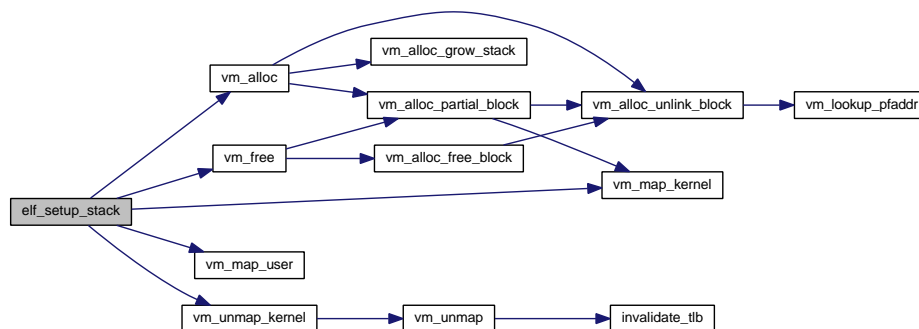


```

228     * it temporarily in this address space so we can write to it. */
229     addr_t top_page = vm_alloc(global_page_allocator);
230     vm_map_kernel(top_page, page, VM_FLAG_READ_WRITE);
231
232     /* start at the top */
233     uint32_t *sp = (uint32_t *) (top_page + PAGE_SIZE);
234
235     /* Program name string: "proc", null-terminated */
236     * (--sp) = 0;
237     * (--sp) = 0x636f7270;
238
239     char *argv0 = (char *) STACK_BASE - 2 * sizeof(uint32_t);
240
241     /* auxiliary vectors */
242     Elf32_auxv_t *auxvp = (Elf32_auxv_t *) sp - 7;
243
244     auxvp[0].a_type = AT_PHDR;
245     auxvp[0].a_un.a_val = (int32_t) info->at_phdr;
246
247     auxvp[1].a_type = AT_PHENT;
248     auxvp[1].a_un.a_val = (int32_t) info->at_phent;
249
250     auxvp[2].a_type = AT_PHNUM;
251     auxvp[2].a_un.a_val = (int32_t) info->at_phnum;
252
253     auxvp[3].a_type = AT_PAGESZ;
254     auxvp[3].a_un.a_val = PAGE_SIZE;
255
256     auxvp[4].a_type = AT_ENTRY;
257     auxvp[4].a_un.a_val = (int32_t) info->entry;
258
259     auxvp[5].a_type = AT_STACKBASE;
260     auxvp[5].a_un.a_val = STACK_BASE;
261
262     auxvp[6].a_type = AT_NULL;
263     auxvp[6].a_un.a_val = 0;
264
265     sp = (uint32_t *) auxvp;
266
267     /* empty environment variables */
268     * (--sp) = 0;
269
270     /* argv with only program name */
271     * (--sp) = 0;
272     * (--sp) = (uint32_t) argv0;
273
274     /* argc */
275     * (--sp) = 1;
276
277     info->stack_addr = (addr_t) STACK_BASE - PAGE_SIZE + ((addr_t) sp - top_page);
278
279     /* unmap and free temporary page */
280     vm_unmap_kernel(top_page);
281     vm_free(global_page_allocator, top_page);
282 }

```

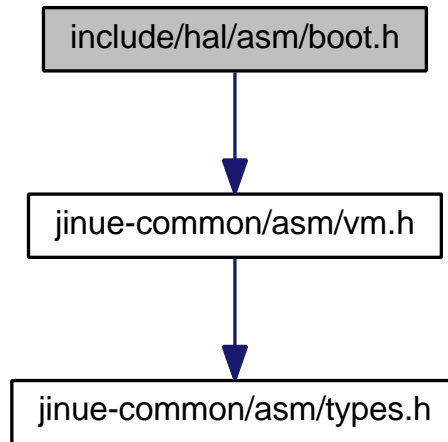
Here is the call graph for this function:



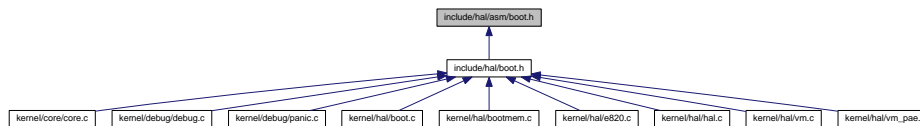
4.10 include/hal/asm/boot.h File Reference

```
#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`

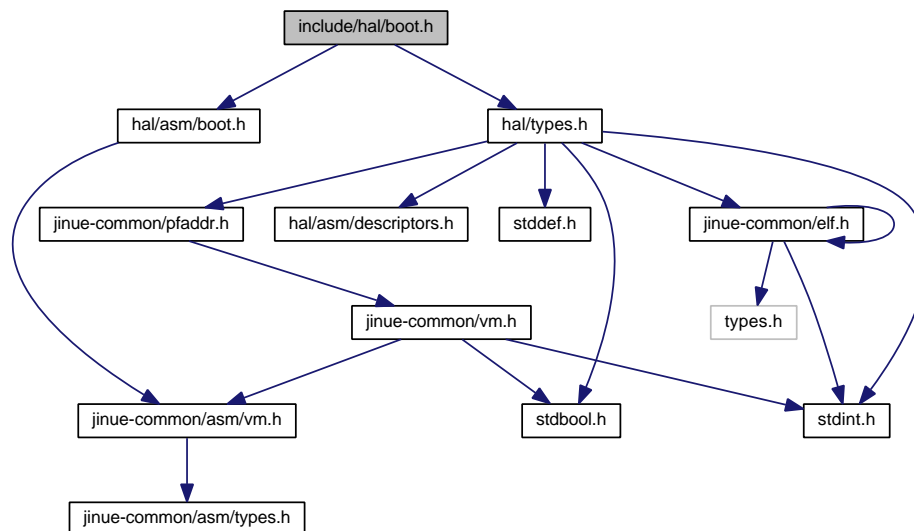
Definition at line 42 of file boot.h.

4.11 include/hal/boot.h File Reference

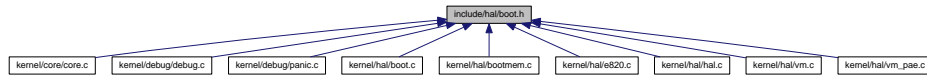
```
#include <hal/asm/boot.h>
```

```
#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.

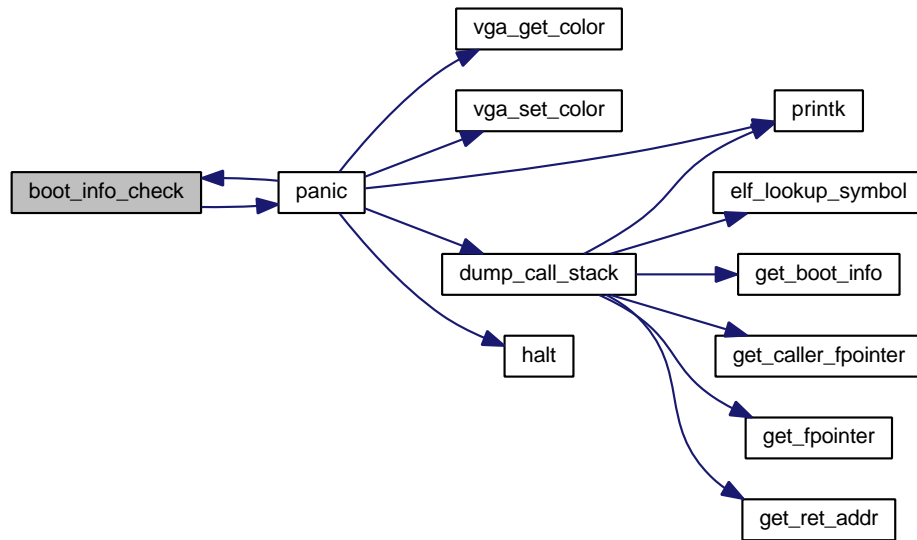
Referenced by hal_init(), and panic().

```

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         return false;
51     }
52 }
53
54 if (boot_info->setup_signature != BOOT_SETUP_MAGIC) {
55     if (panic_on_failure) {
56         panic("Bad setup header signature.");
57     }
58     return false;
59 }
60
61 return true;
62 }
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`.

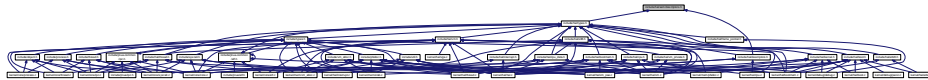
Referenced by `bootmem_init()`, `dump_call_stack()`, `e820_dump()`, `hal_init()`, and `vm_boot_init()`.

```

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 | SEG_FLAG_PRESENT)
 - commonly used segment flags*
- #define **SEG_FLAG_NORMAL_GATE** (SEG_FLAG_32BIT_GATE | SEG_FLAG_SYSTEM | SEG_FLAG_PRESENT)
 - commonly used gate flags*
- #define **SEG_FLAG_TSS** (SEG_FLAG_IN_BYTES | SEG_FLAG_SYSTEM | SEG_FLAG_PRESENT)
 - commonly used flags for task-state segment*
- #define **SEG_TYPE_READ_ONLY** 0
 - read-only data segment*
- #define **SEG_TYPE_DATA** 2
 - read/write data segment*
- #define **SEG_TYPE_TASK_GATE** 5
 - task gate*
- #define **SEG_TYPE_INTERRUPT_GATE** 6
 - interrupt gate*
- #define **SEG_TYPE_TRAP_GATE** 7
 - trap gate*

- **#define SEG_TYPE_TSS 9**
task-state segment (TSS)
- **#define SEG_TYPE_CODE 10**
code segment
- **#define SEG_TYPE_CALL_GATE 12**
call gate

4.12.1 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.

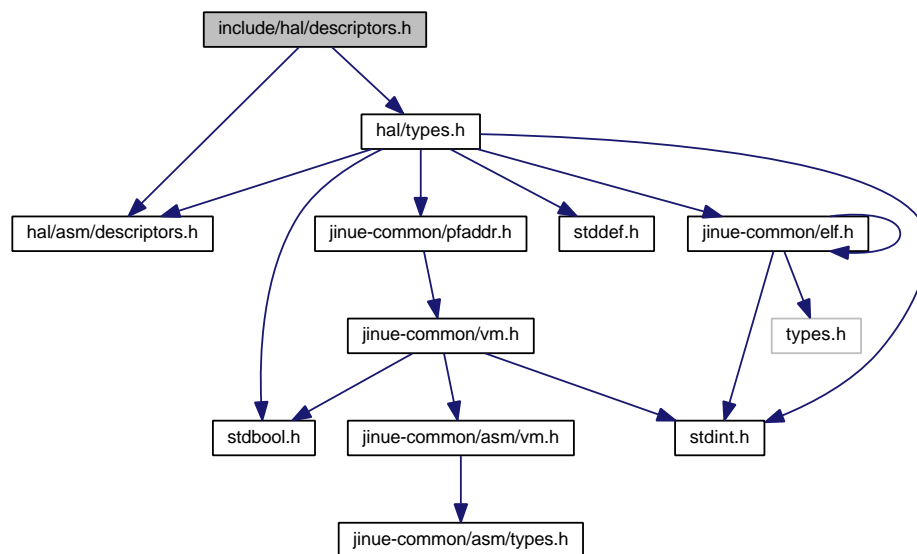
Referenced by cpu_init_data().

4.13 include/hal/descriptors.h File Reference

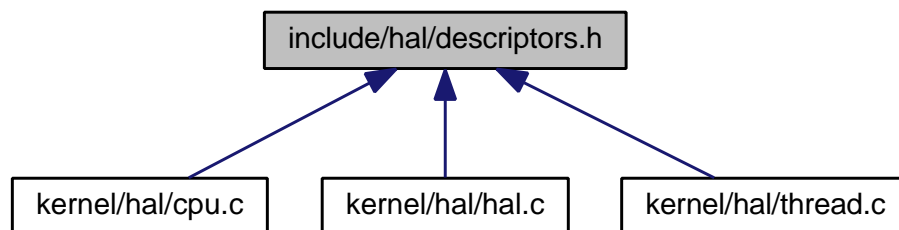
```
#include <hal/asm/descriptors.h>
```

```
#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:

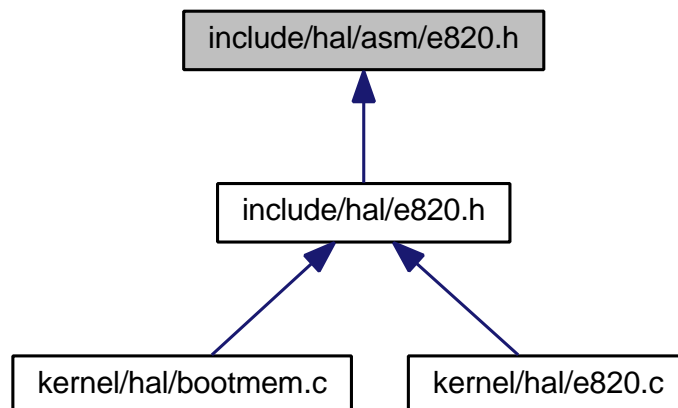
```
(
    PACK_DESCRIPTOR((type),  0xf0ff,  0,  SEG_FLAGS_OFFSET) \
| PACK_DESCRIPTOR((base),   0xff,    24, 56) \
| PACK_DESCRIPTOR((base),   0xff,    16, 32) \
| PACK_DESCRIPTOR((base),   0xffff,  0,  16) \
| PACK_DESCRIPTOR((limit),  0xf,     16, 48) \
| PACK_DESCRIPTOR((limit),  0xffff,  0,   0)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

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()`.

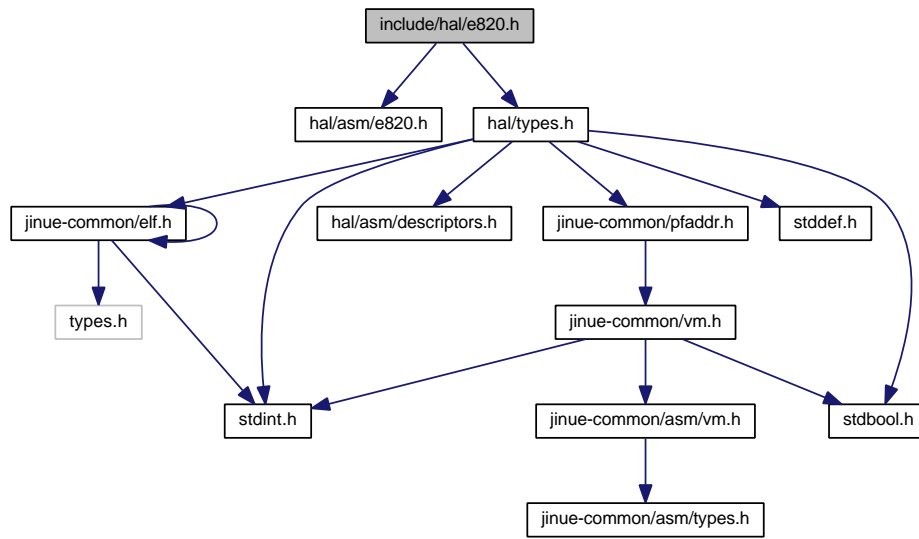
4.14.1.4 `#define E820_SMAP 0x534d4150`

Definition at line 41 of file `e820.h`.

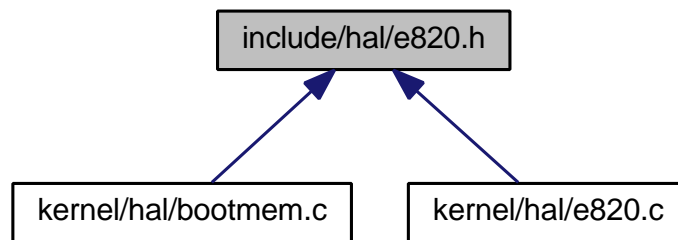
4.15 `include/hal/e820.h` File Reference

```
#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");

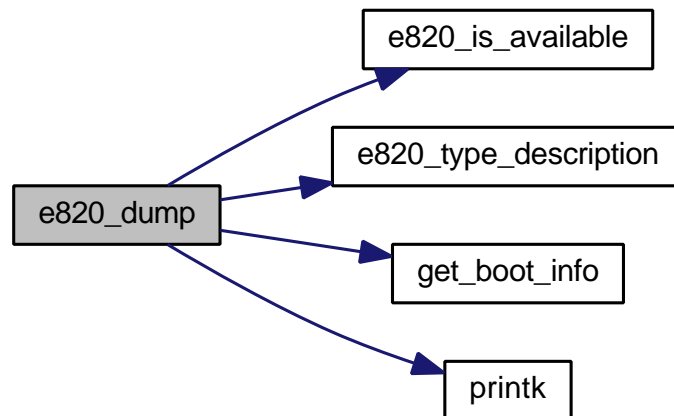
```

```

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:



4.15.1.2 `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.

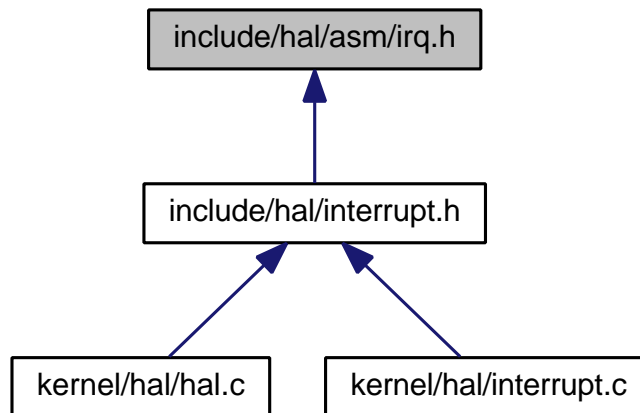
Referenced by e820_dump().

```

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

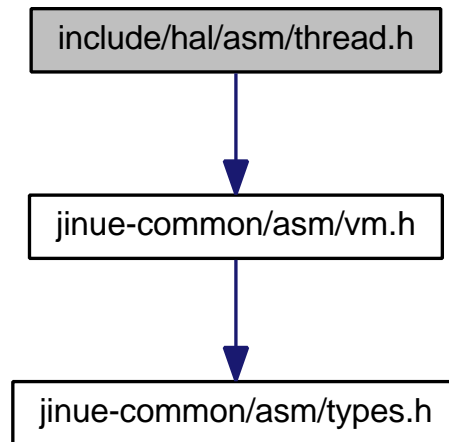
Definition at line 35 of file irq.h.

Referenced by hal_init().

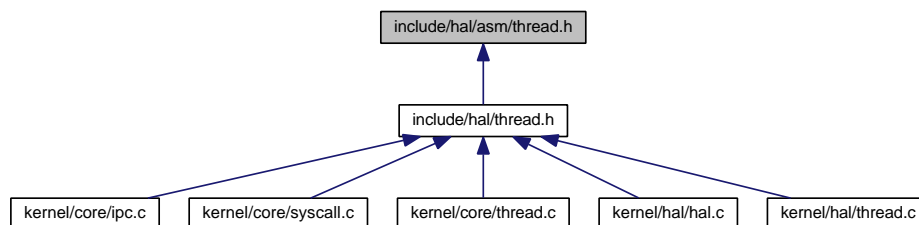
4.17 include/hal/asm/thread.h File Reference

```
#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.

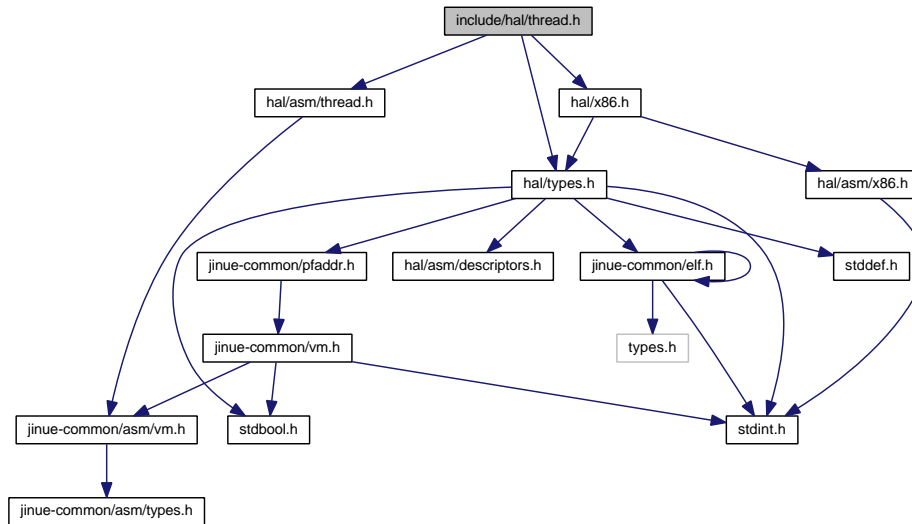
4.17.1.2 `#define THREAD_CONTEXT_SIZE PAGE_SIZE`

Definition at line 38 of file thread.h.

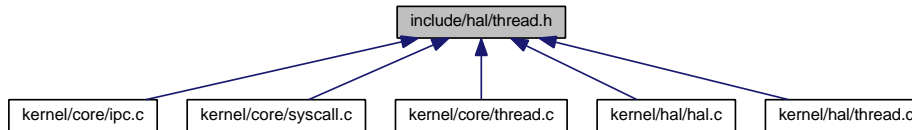
4.18 include/hal/thread.h File Reference

```
#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().

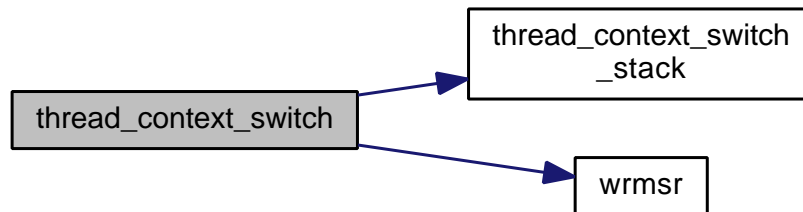
Referenced by thread_switch().


```

148                                     {
149
151     assert(to_ctx != NULL);
152
154     assert(from_ctx != NULL || ! destroy_from);
155
156     /* nothing to do if this is already the current thread */
157     if(from_ctx != to_ctx) {
158         /* setup TSS with kernel stack base for this thread context */
159         addr_t kernel_stack_base = get_kernel_stack_base(to_ctx);
160         tss_t *tss = get_tss();
161
162         tss->esp0 = kernel_stack_base;
163         tss->esp1 = kernel_stack_base;
164         tss->esp2 = kernel_stack_base;
165
166         /* update kernel stack address for SYSENTER instruction */
167         if(cpu_has_feature(CPU_FEATURE_SYSENTER)) {
168             wrmsr(MSR_IA32_SYSENTER_ESP, (uint64_t)(uintptr_t)kernel_stack_base);
169         }
170
171         /* switch thread context stack */
172         thread_context_switch_stack(from_ctx, to_ctx, destroy_from);
173     }
174 }

```

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()`.

```

87                                     {
88
89     /* allocate thread context */
90     thread_t *thread = (thread_t *)vm_alloc( global_page_allocator );
91
92     if(thread != NULL) {
93         pfaddr_t pf = pfalloc();
94
95         if(pf == PFNULL) {
96             vm_free(global_page_allocator, (addr_t)thread);
97             return NULL;
98         }
99
100         vm_map_kernel((addr_t)thread, pf, VM_FLAG_READ_WRITE);
101
102         /* initialize fields */
103         thread_context_t *thread_ctx = &thread->thread_ctx;
104

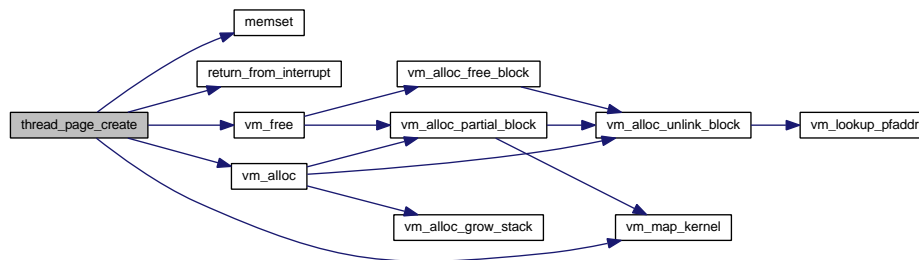
```

```

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->savestack_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.

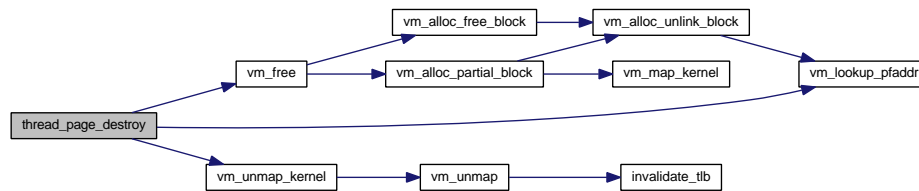
References global_page_allocator, NULL, pffree, vm_free(), vm_lookup_pfaddr(), and vm_unmap_kernel().

```

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 }

```

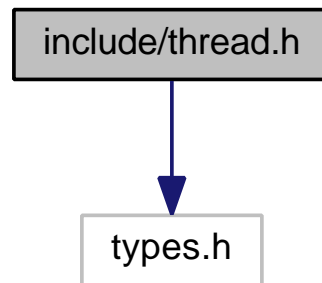
Here is the call graph for this function:



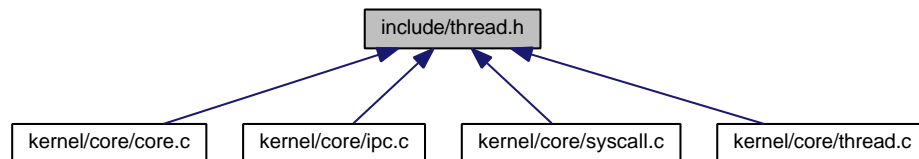
4.19 include/thread.h File Reference

```
#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()**.

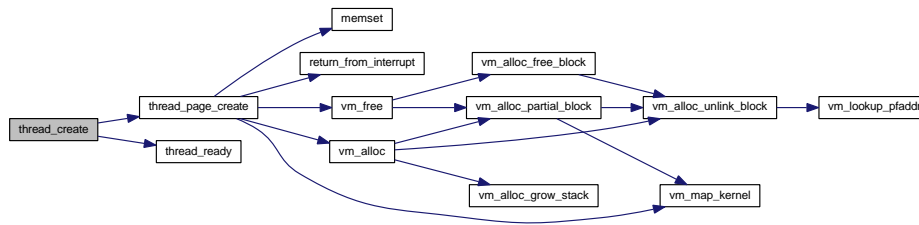
Referenced by `dispatch_syscall()`, and `kmain()`.

```

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         jinue_node_init(&thread->thread_list);
52         thread->process = process;
53         thread->sender = NULL;
54         thread_ready(thread);
55     }
56
57     return thread;
58 }
59
60
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;

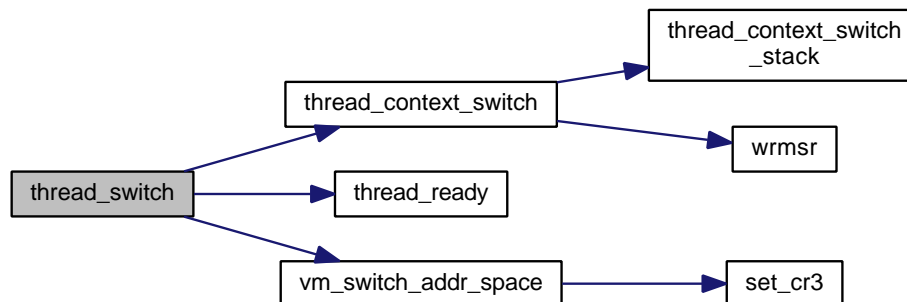
```

```

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:



4.19.1.4 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 }

```


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().

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

```

```

146     addr_t vga_text_base = kernel_vm_top;
147
148     while(addr < (addr_t)VGA_TEXT_VID_TOP) {
149         vm_map_early(kernel_vm_top, ADDR_TO_PFADDR((uintptr_t)addr),
VM_FLAG_KERNEL | VM_FLAG_READ_WRITE);
150         kernel_vm_top += PAGE_SIZE;
151         addr += PAGE_SIZE;
152     }
153
154     /* remap VGA text buffer
155     *
156     * Note: after the call to vga_set_base_addr() below until we switch to the
157     * new address space, VGA output is not possible. Calling printk() will cause
158     * a kernel panic due to a page fault (and the panic handler calls printk()). */
159     printk("Remapping text video memory at 0x%x\n", kernel_vm_top);
160
161     vga_set_base_addr(vga_text_base);
162
163     if(use_pae) {
164         /* If we are enabling PAE, this is where the switch to the new page
165         * tables actually happens instead of at the call to vm_switch_addr_space()
166         * as would be expected.
167         *
168         * From Intel 64 and IA-32 Architectures Software Developer's Manual
169         * Volume 3: System Programming Guide, section 4.4.1 "PDPTE Registers":
170         *
171         * " The logical processor loads [the PDPTE] registers from the PDPTEs
172         *   in memory as part of certain operations:
173         *   * If PAE paging would be in use following an execution of MOV to
174         *     CR0 or MOV to CR4 (see Section 4.1.1) and the instruction is
175         *     modifying any of (...) CR4.PAE, (...); then the PDPTEs are
176         *     loaded from the address in CR3. "
177         *
178         * There are bootstrapping issues when enabling PAE while paging is enabled.
179         * See the comment at the top of the vm_pae_create_initial_addr_space()
180         * function in vm_pae.c for more detail. */
181         vm_pae_enable();
182     }
183
184     /* switch to new address space */
185     vm_switch_addr_space(addr_space);
186
187     /* initialize global page allocator (region starting at KLIMIT)
188     *
189     * TODO Some work needs to be done in the page allocator to support allocating
190     * up to the top of memory (i.e. 0x100000000, which cannot be represented on
191     * 32 bits). In the mean time, we leave a 4MB gap. */
192     global_page_allocator = &__global_page_allocator;
193     vm_alloc_init_allocator(global_page_allocator, (addr_t)KLIMIT, (addr_t)0 - 4 *
MB);
194
195     vm_alloc_add_region(global_page_allocator, (addr_t)KLIMIT, (addr_t)boot_info->
image_start);
196     vm_alloc_add_region(global_page_allocator, (addr_t)kernel_vm_top, (addr_t)0 - 4 *
MB);
197
198     /* create slab cache to allocate PDPTs
199     *
200     * This must be done after the global page allocator has been initialized
201     * because the slab allocator needs to allocate a slab to allocate the new
202     * slab cache on the slab cache cache.
203     *
204     * This must be done before the first time vm_create_addr_space() is called. */
205     if(use_pae) {
206         vm_pae_create_pdpt_cache();
207     }
208 }

```



```

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
462     assert( is_fast_map_pointer(vaddr) );
463
464     assert( page_offset_of(vaddr) == 0 );
465
466     pte = get_pte_with_offset(global_page_tables, page_number_of(
467         EARLY_VIRT_TO_PHYS((uintptr_t)vaddr) ));
468     set_pte(pte, paddr, flags | VM_FLAG_PRESENT);
469 }
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.

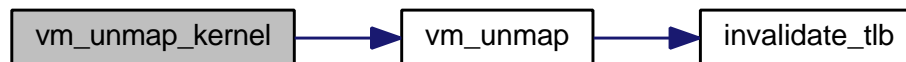
References NULL, and vm_unmap().

Referenced by elf_load(), elf_setup_stack(), thread_page_destroy(), vm_clone_page_directory(), and vm_destroy_page_directory().

```

423     {
424         vm_unmap(NULL, addr);
425     }
  
```

Here is the call graph for this function:



4.21.2.12 void vm_unmap_user (addr_space_t* addr_space, addr_t addr)

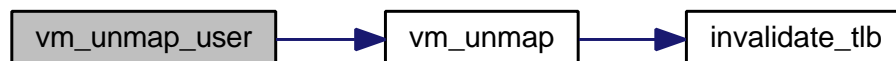
Definition at line 427 of file vm.c.

References vm_unmap().

```

427     {
428         vm_unmap(addr_space, addr);
429     }
  
```

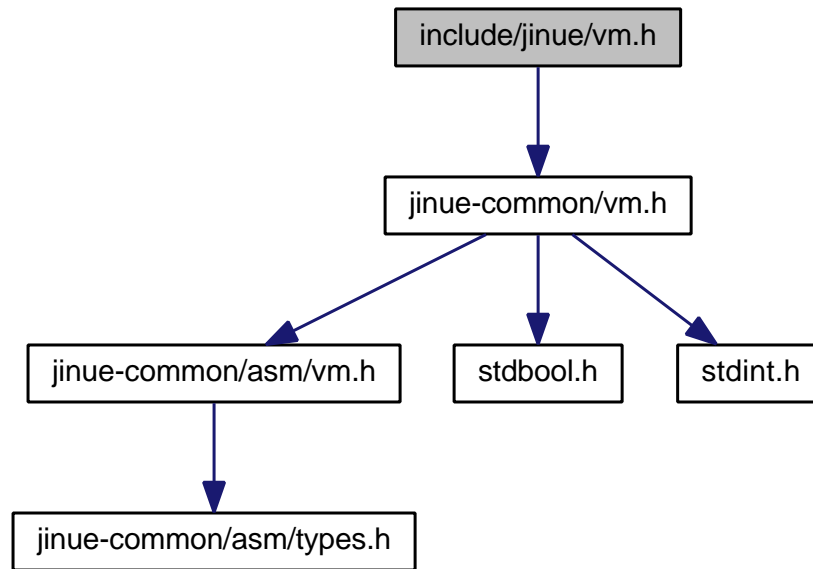
Here is the call graph for this function:



4.22 include/jinue/vm.h File Reference

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

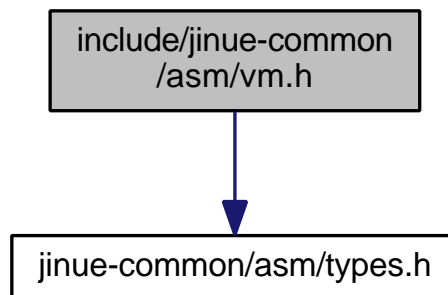
Include dependency graph for vm.h:



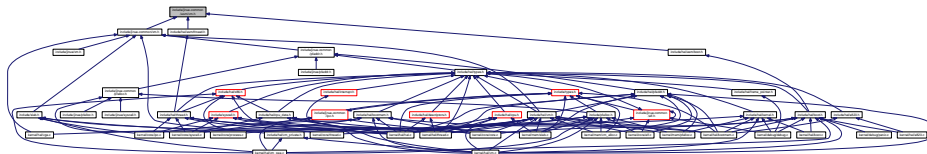
4.23 include/jinue-common/asm/vm.h File Reference

```
#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().

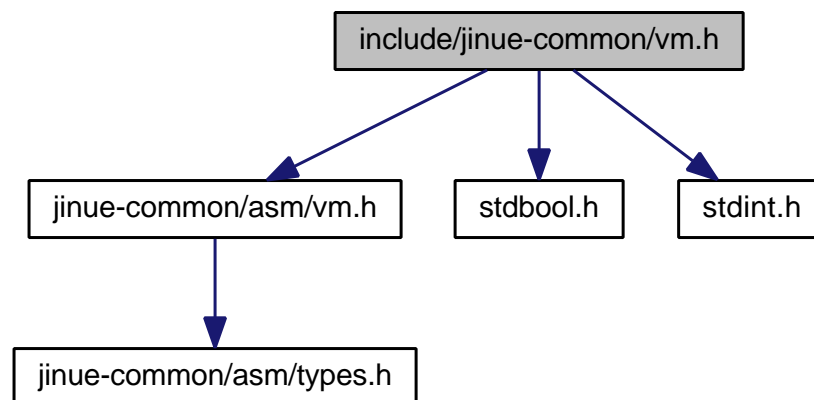
4.24 `include/jinue-common/vm.h` File Reference

```
#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

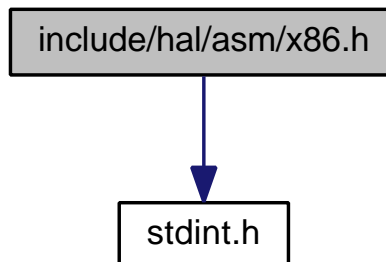
Definition at line 42 of file vm.h.

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:



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)

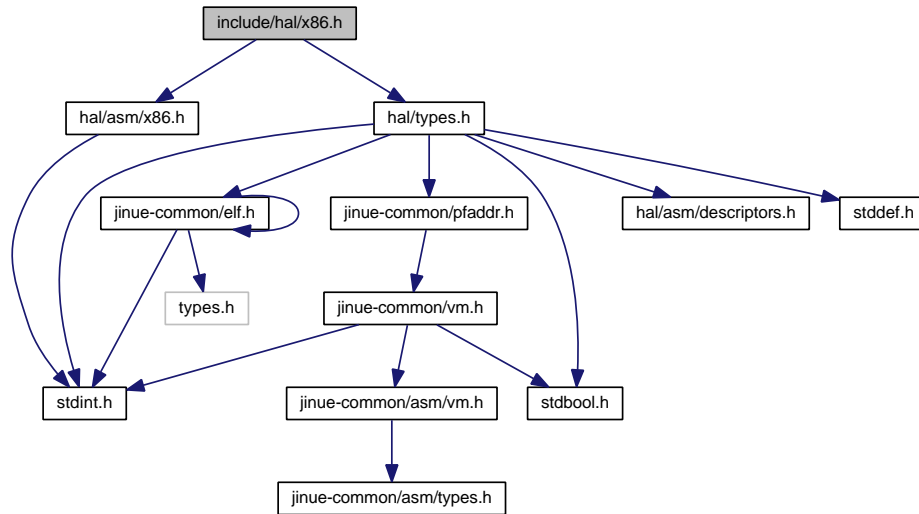
write-through cache policy for page

Definition at line 66 of file x86.h.

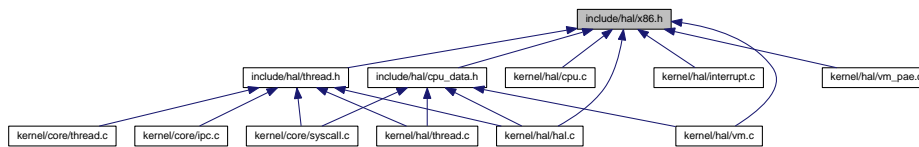
4.26 include/hal/x86.h File Reference

```
#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)

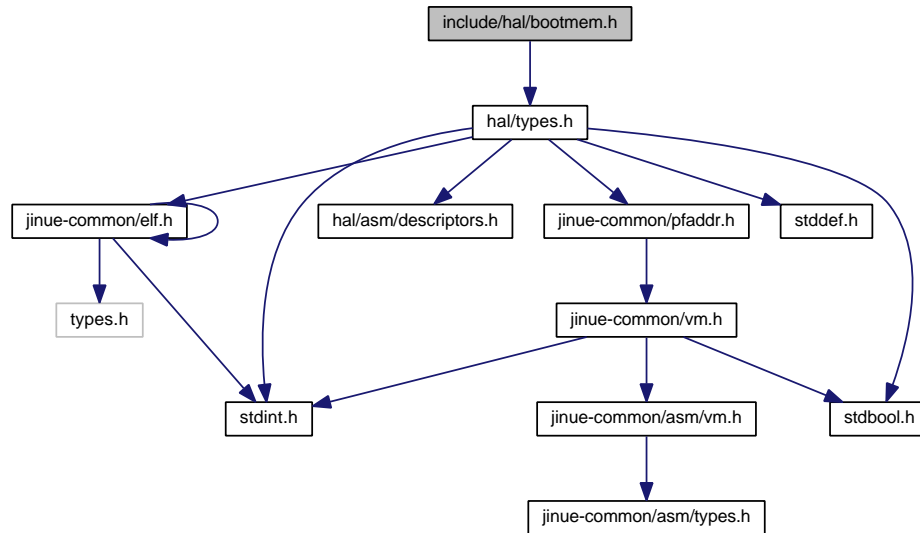
4.26.2.30 void wrmsr (msr_addr_t addr, uint64_t val)

Referenced by hal_init(), and thread_context_switch().

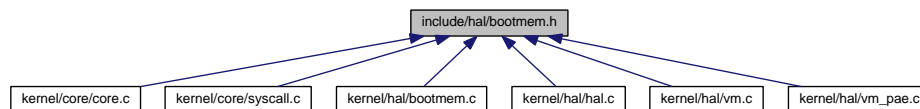
4.27 include/hal/bootmem.h File Reference

```
#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(dp = head, ptr = *head; ptr != NULL; dp = &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             *dp = 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     }

```

```

110      /* fix size or addr if block overlaps hole */
111      if(addr >= hole_addr && addr < hole_top) {
112          ptr->addr = hole_top;
113          ptr->count = (top - hole_top) / (PAGE_SIZE >> PFADDR_SHIFT);
114      }
115      return;
116  }
117
118      if(top > hole_addr && top <= hole_top) {
119          ptr->count = (hole_addr - addr) / (PAGE_SIZE >> PFADDR_SHIFT);
120      }
121  }
122  }

```

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, 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
133  /* copy the available ram entries from the e820 map and insert them
134   * in a linked list */
135  ram_map = NULL;
136
137  for(idx = 0; idx < boot_info->e820_entries; ++idx) {
138      const e820_t *e820_entry = &boot_info->e820_map[idx];
139
140      if(! e820_is_valid(e820_entry)) {
141          continue;
142      }
143
144      if( e820_is_available(e820_entry) ) {
145          /* get memory entry start and end addresses */
146          e820_addr_t start = e820_entry->addr;
147          e820_addr_t end   = start + e820_entry->size;
148
149          /* align on page boundaries */
150          if( OFFSET_OF(start, PAGE_SIZE) != 0 ) {
151              start = (start & (e820_addr_t)~PAGE_MASK) + PAGE_SIZE;
152          }
153
154          if( OFFSET_OF(end, PAGE_SIZE) != 0 ) {
155              end = (end & (e820_addr_t)~PAGE_MASK);
156          }
157
158          /* If Physical Address Extension (PAE) is disabled, memory above the
159           * 4GB mark is not usable. */
160          if(! use_pae) {
161              /* If this memory region is completely above the 4GB mark, exclude it. */
162              if(start >= ADDR_4GB) {
163                  continue;
164              }
165
166              /* If this memory region starts below the 4GB mark but extends
167               * beyond it, crop at 4GB. */
168              if(end > ADDR_4GB) {
169                  end = ADDR_4GB;
170              }
171          }
172
173          /* add entry to linked list */
174          if(end > start) {
175              new_ram_map_entry(ADDR_TO_PFADDR(start), (uint32_t)(end - start) /
176              PAGE_SIZE, &ram_map);
177          }
178      }
179  }
180
181  /* apply every unavailable entries from the e820 map as holes */
182  for(idx = 0; idx < boot_info->e820_entries; ++idx) {
183      const e820_t *e820_entry = &boot_info->e820_map[idx];
184
185      if(! e820_is_valid(e820_entry)) {
186          continue;
187      }
188
189      if( e820_is_available(e820_entry) ) {
190          continue;
191      }
192
193      e820_addr_t start = e820_entry->addr;
194      e820_addr_t end   = start + e820_entry->size;
195
196      apply_mem_hole(start, end, &ram_map);
197  }
198
199  /* Apparently, the first 64k of memory are corrupted by some BIOSes.
200   * It would be nice to try to detect this. In the meantime, let's
201   * assume the problem is present. */
202  apply_mem_hole(0, 0x10000, &ram_map);
203
204  /* the kernel image, its heap and stack, and early-allocated pages */
205  apply_mem_hole((uint32_t)boot_info->image_start, (uint32_t)kernel_region_top, &
206  ram_map);
207
208  /* Entry removal may have left garbage on the heap (bootmem_t
209   * structures which were allocated on the heap but are no longer
210   * linked). Let's clean up. */
211  temp_root = NULL;
212

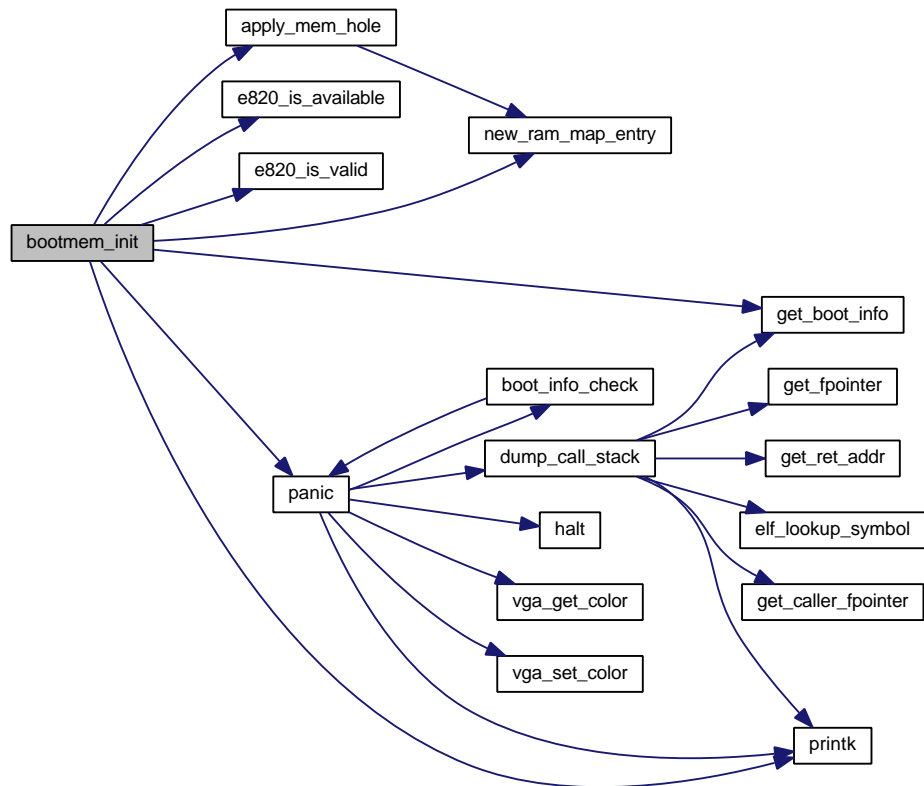
```

```

213     for(ptr = ram_map; ptr != NULL; ptr = ptr->next) {
214         new_ram_map_entry(ptr->addr, ptr->count, &temp_root);
215     }
216     ram_map = NULL;
217     boot_heap = initial_boot_heap;
218
219     for(ptr = temp_root; ptr != NULL; ptr = ptr->next) {
220         new_ram_map_entry(ptr->addr, ptr->count, &ram_map);
221     }
222
223     /* at this point, we should have at least one block of available RAM */
224     if( ram_map == NULL ) {
225         panic("no available memory.");
226     }
227
228     /* Let's count and display the total amount of available memory */
229     uint32_t page_count = 0;
230     for(ptr = ram_map; ptr != NULL; ptr = ptr->next) {
231         page_count += ptr->count;
232     }
233
234     printk("%u kilobytes (%u pages) of memory available.\n",
235           (uint32_t)(page_count * PAGE_SIZE / KB),
236           (uint32_t)(page_count) );
237
238     /* head pointer for bootmem_get_block() */
239     bootmem_root = ram_map;
240 }

```

Here is the call graph for this function:



4.27.2.4 void new_ram_map_entry (paddr_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 }
```

4.27.3 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

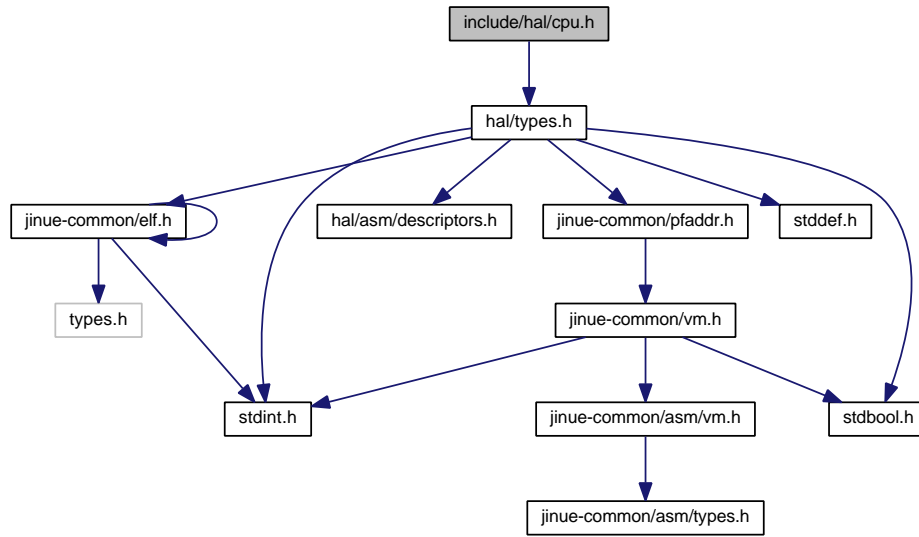
Definition at line 46 of file `bootmem.c`.

Referenced by `bootmem_init()`.

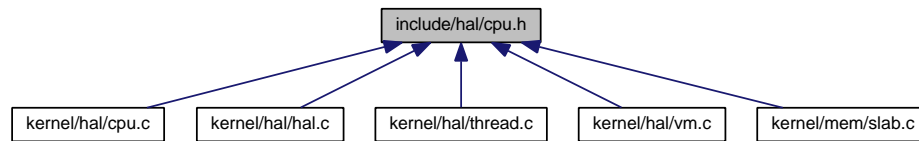
4.28 `include/hal/cpu.h` File Reference

```
#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 /* inel */`
- `#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 /* intel */`

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, CPU_FEATURE_SYSCALL, CPU_FEATURE_SYSENTER, CPU_VENDOR_AMD, CPU_VENDOR_AMD_DW0, CPU_VENDOR_AMD_DW1, CPU_VENDOR_AMD_DW2, CPU_VENDOR_GENERIC, CPU_VENDOR_INTEL, CPU_VENDOR_INTEL_DW0, CPU_VENDOR_INTEL_DW1, CPU_VENDOR_INTEL_DW2, cpuid(), CPUID_EXT_FEATURE_SYSCALL, CPUID_FEATURE_APIC, CPUID_FEATURE_CLFLUSH, CPUID_FEATURE_PAE, CPUID_FEATURE_SEP, 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
137                    && vendor_dw1 == CPU_VENDOR_INTEL_DW1
138                    && vendor_dw2 == CPU_VENDOR_INTEL_DW2) {
139
140                cpu_info.vendor = CPU_VENDOR_INTEL;
141            }
142
143            /* get processor signature (family/model/stepping) and feature flags */
144            if(cpuid_max >= 1) {
145                /* function 1: processor signature and feature flags */
146                regs.eax = 1;
147
148                /* call CPUID instruction */
149                signature = cpuid(&regs);
150
151                /* set processor signature */
152                cpu_info.stepping = signature & 0xf;
153                cpu_info.model    = (signature>>4) & 0xf;
154                cpu_info.family    = (signature>>8) & 0xf;
155
156                /* feature flags */
157                flags = regs.edx;
158
159                /* cache alignment */
160                if(flags & CPUID_FEATURE_CLFLUSH) {
161                    cpu_info.dcache_alignment = ((regs.ebx >> 8) & 0xff) * 8;
162                }
163            }
164
165            /* extended function 0: max value of eax when calling CPUID (extended function) */
166            regs.eax = 0x80000000;
167            cpuid_ext_max = cpuid(&regs);

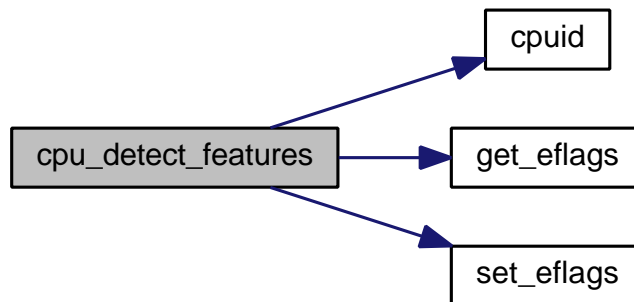
```

```

168
169     /* get extended feature flags */
170     if(cpuinfo_ext_max >= 0x80000001) {
171         /* extended function 1: extended feature flags */
172         regs.eax = 0x80000001;
173         (void)cpuid(&regs);
174
175         /* extended feature flags */
176         ext_flags = regs.edx;
177     }
178
179     /* support for SYSENTER/SYSEXIT instructions */
180     if(flags & CPUID_FEATURE_SEP) {
181         if(cpuinfo.vendor == CPU_VENDOR_AMD) {
182             cpuinfo.features |= CPU_FEATURE_SYSENTER;
183         }
184         else if(cpuinfo.vendor == CPU_VENDOR_INTEL) {
185             if(cpuinfo.family == 6 && cpuinfo.model < 3 && cpuinfo.
stepping < 3) {
186                 /* not supported */
187             }
188             else {
189                 cpuinfo.features |= CPU_FEATURE_SYSENTER;
190             }
191         }
192     }
193
194     /* support for SYSCALL/SYSRET instructions */
195     if(cpuinfo.vendor == CPU_VENDOR_AMD) {
196         if(ext_flags & CPUID_EXT_FEATURE_SYSCALL) {
197             cpuinfo.features |= CPU_FEATURE_SYSCALL;
198         }
199     }
200
201     /* support for local APIC */
202     if(cpuinfo.vendor == CPU_VENDOR_AMD || cpuinfo.vendor ==
CPU_VENDOR_INTEL) {
203         if(flags & CPUID_FEATURE_APIC) {
204             cpuinfo.features |= CPU_FEATURE_LOCAL_APIC;
205         }
206     }
207
208     /* support for physical address extension (PAE) */
209     if(cpuinfo.vendor == CPU_VENDOR_AMD || cpuinfo.vendor ==
CPU_VENDOR_INTEL) {
210         if(flags & CPUID_FEATURE_PAE) {
211             cpuinfo.features |= CPU_FEATURE_PAE;
212         }
213     }
214 }
215 }

```

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()`.

```

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, 0xffff, SEG_TYPE_CODE |
58         SEG_FLAG_KERNEL | SEG_FLAG_NORMAL);
59
60     data->gdt[GDT_KERNEL_DATA] =
61         SEG_DESCRIPTOR( 0, 0xffff, SEG_TYPE_DATA |
62         SEG_FLAG_KERNEL | SEG_FLAG_NORMAL);
63
64     data->gdt[GDT_USER_CODE] =
65         SEG_DESCRIPTOR( 0, 0xffff, SEG_TYPE_CODE |
66         SEG_FLAG_USER | SEG_FLAG_NORMAL);
67
68     data->gdt[GDT_USER_DATA] =
69         SEG_DESCRIPTOR( 0, 0xffff, SEG_TYPE_DATA |
70         SEG_FLAG_USER | SEG_FLAG_NORMAL);
71
72     data->gdt[GDT_TSS] =
73         SEG_DESCRIPTOR( tss, TSS_LIMIT-1, SEG_TYPE_TSS |
74         SEG_FLAG_KERNEL | SEG_FLAG_TSS);
75
76     data->gdt[GDT_PER_CPU_DATA] =
77         SEG_DESCRIPTOR( data, sizeof(cpu_data_t)-1, SEG_TYPE_DATA |
78         SEG_FLAG_KERNEL | SEG_FLAG_32BIT | SEG_FLAG_IN_BYTES | SEG_FLAG_NOSYSTEM |
79         SEG_FLAG_PRESENT);
80
81     data->gdt[GDT_USER_TLS_DATA] = SEG_DESCRIPTOR(0, 0, 0);
82
83     /* setup kernel stack in TSS */
84     tss->ss0 = SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL);
85     tss->ss1 = SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL);
86     tss->ss2 = SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL);
87
88     /* kernel stack address is updated by thread_context_switch() */
89     tss->esp0 = NULL;
90     tss->esp1 = NULL;
91     tss->esp2 = NULL;
92 }

```

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.

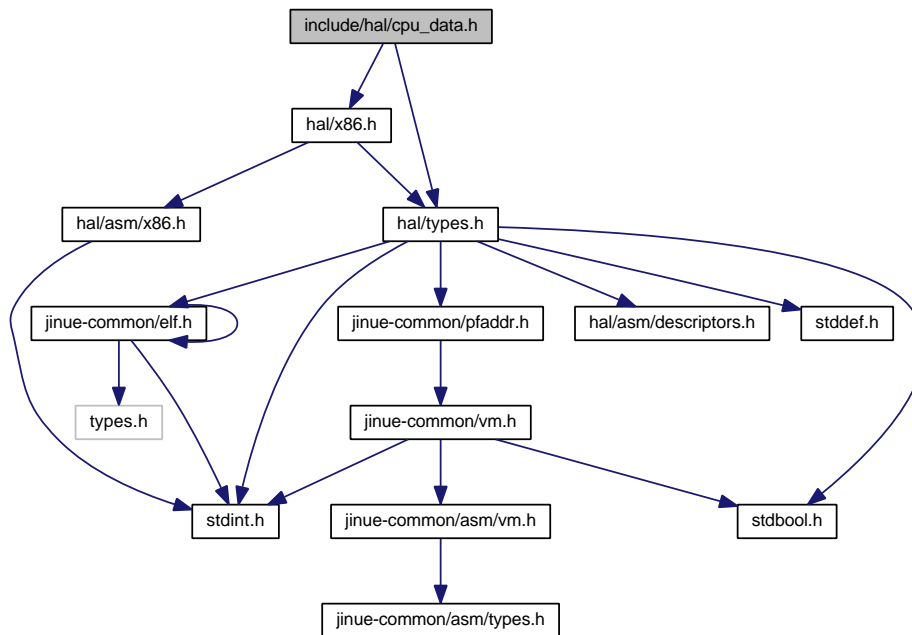
Referenced by slab_cache_create().

4.29 include/hal/cpu_data.h File Reference

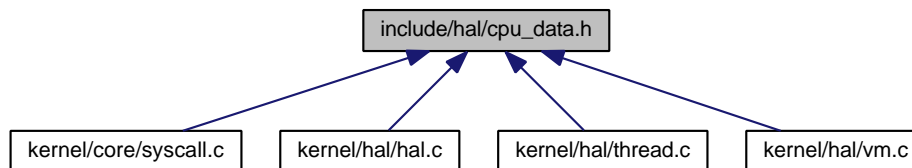
```
#include <hal/types.h>
```

```
#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

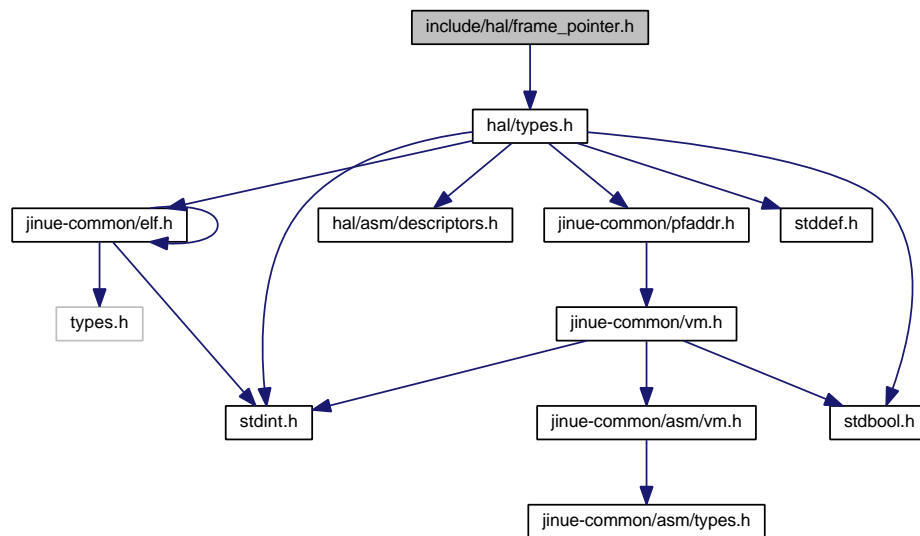
Definition at line 39 of file `cpu_data.h`.

Referenced by `hal_init()`.

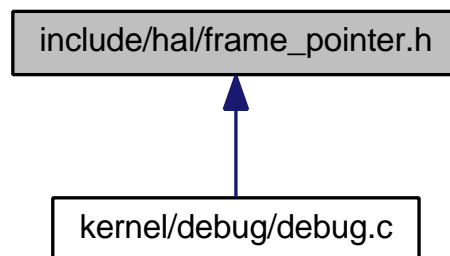
4.30 include/hal/frame_pointer.h File Reference

```
#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()`.

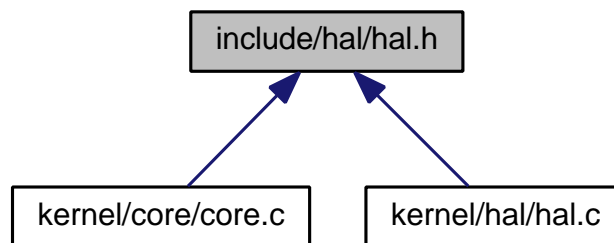
4.30.1.3 `addr_t get_program_counter (void)`

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`, `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
82     assert(page_offset_of(boot_info->image_start) == 0);
83
84     assert(page_offset_of(boot_info->kernel_start) == 0);
85
86     printk("Kernel size is %u bytes.\n", boot_info->kernel_size);
87
88     /* This must be done before any boot heap allocation. */
89     boot_heap = boot_info->boot_heap;
90
91     /* This must be done before any call to pfalloc_early(). */
92     kernel_region_top = boot_info->boot_end;
93
94     /* get cpu info */
95     cpu_detect_features();
96
97     /* allocate new kernel stack */
98     stack = pfalloc_early();
99     stack += PAGE_SIZE;
100
101     /* allocate per-CPU data
102     *
103     * We need to ensure that the Task State Segment (TSS) contained in this
104     * memory block does not cross a page boundary. */
105     assert(sizeof(cpu_data_t) < CPU_DATA_ALIGNMENT);
106
107     boot_heap = ALIGN_END(boot_heap, CPU_DATA_ALIGNMENT);
108
109     cpu_data = boot_heap;
110     boot_heap = cpu_data + 1;
111
112     /* initialize per-CPU data */
113     cpu_init_data(cpu_data, stack);
114
115     /* allocate pseudo-descriptor for GDT and IDT (temporary allocation) */
116     boot_heap_old = boot_heap;
117
118     boot_heap = ALIGN_END(boot_heap, sizeof(pseudo_descriptor_t));
119
120     pseudo = (pseudo_descriptor_t *)boot_heap;
121     boot_heap = (pseudo_descriptor_t *)boot_heap + 1;
122
123     /* load new GDT and TSS */
124     pseudo->addr = (addr_t)&cpu_data->gdt;
125     pseudo->limit = GDT_LENGTH * 8 - 1;
126
127     lgdt(pseudo);
128
129     set_cs( SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL) );
130     set_ss( SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL) );
131     set_data_segments( SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL) );
132     set_gs( SEG_SELECTOR(GDT_PER_CPU_DATA, RPL_KERNEL) );
133
134
135

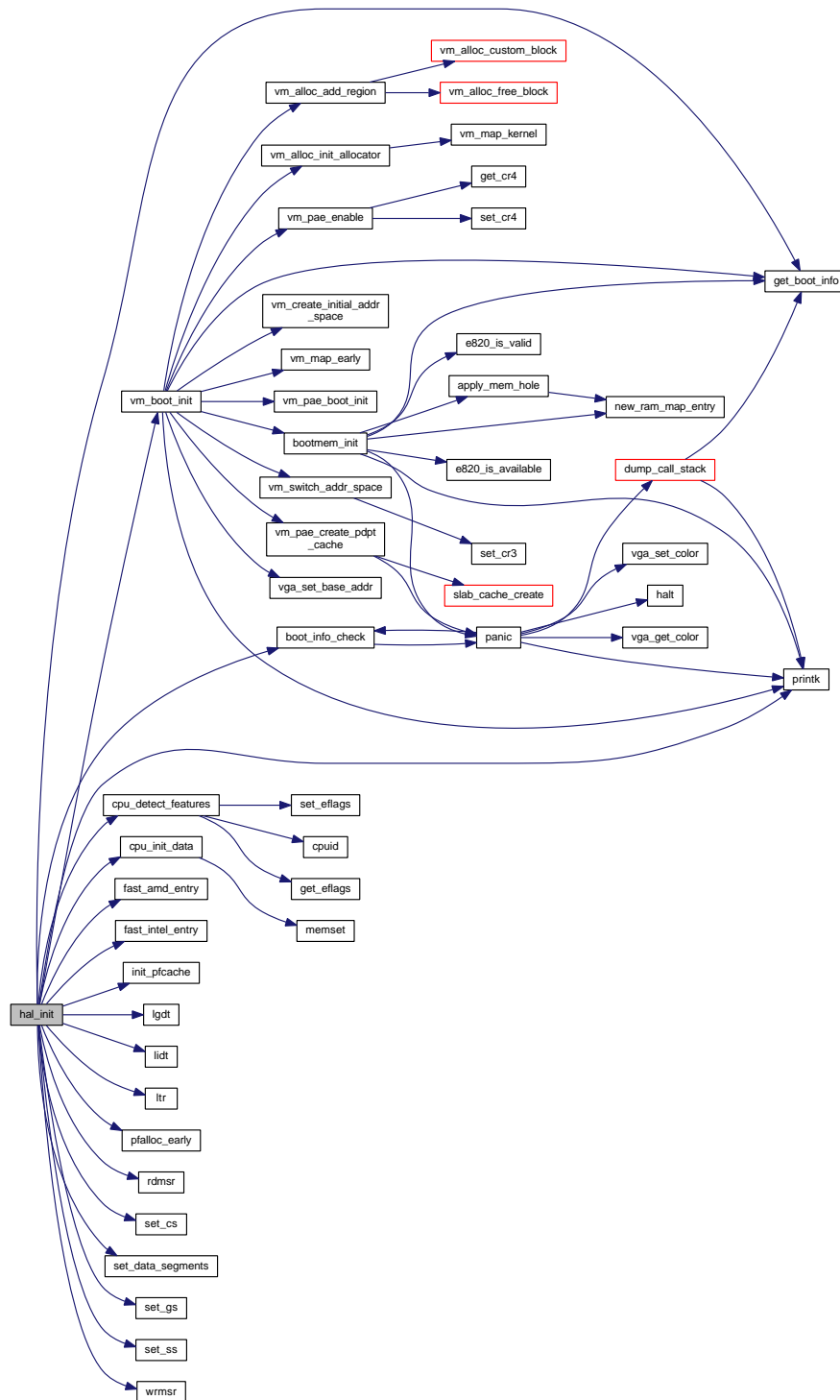
```

```

136     ltr( SEG_SELECTOR(GDT_TSS, RPL_KERNEL) );
137
138     /* initialize IDT */
139     for(idx = 0; idx < IDT_VECTOR_COUNT; ++idx) {
140         /* get address, which is already stored in the IDT entry */
141         addr = (addr_t)(uintptr_t)idt[idx];
142
143         /* set interrupt gate flags */
144         flags = SEG_TYPE_INTERRUPT_GATE | SEG_FLAG_NORMAL_GATE;
145
146         if(idx == SYSCALL_IRQ) {
147             flags |= SEG_FLAG_USER;
148         }
149         else {
150             flags |= SEG_FLAG_KERNEL;
151         }
152
153         /* create interrupt gate descriptor */
154         idt[idx] = GATE_DESCRIPTOR(
155             SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL),
156             addr,
157             flags,
158             NULL );
159     }
160
161     pseudo->addr = (addr_t)idt;
162     pseudo->limit = IDT_VECTOR_COUNT * sizeof(seg_descriptor_t) - 1;
163     lidt(pseudo);
164
165     /* de-allocate pseudo-descriptor */
166     boot_heap = boot_heap_old;
167
168     /* initialize the page frame allocator */
169     page_stack_buffer = (pfaddr_t *)pfalloc_early();
170     init_pfcache(&global_pfcache, page_stack_buffer);
171
172     for(idx = 0; idx < KERNEL_PAGE_STACK_INIT; ++idx) {
173         pffree( EARLY_PTR_TO_PFADDR( pfalloc_early() ) );
174     }
175
176     /* initialize virtual memory management, enable paging
177     *
178     * below this point, it is no longer safe to call pfalloc_early() */
179     vm_boot_init();
180
181     /* choose system call method */
182     syscall_method = SYSCALL_METHOD_INTR;
183
184     if(cpu_has_feature(CPU_FEATURE_SYSENTER)) {
185         syscall_method = SYSCALL_METHOD_FAST_INTEL;
186
187         wrmsr(MSR_IA32_SYSENTER_CS, SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL));
188         wrmsr(MSR_IA32_SYSENTER_EIP, (uint64_t)(uintptr_t)fast_intel_entry);
189
190         /* kernel stack address is set when switching thread context */
191         wrmsr(MSR_IA32_SYSENTER_ESP, (uint64_t)(uintptr_t)NULL);
192     }
193
194     if(cpu_has_feature(CPU_FEATURE_SYSCALL)) {
195         syscall_method = SYSCALL_METHOD_FAST_AMD;
196
197         msrval = rdmsr(MSR_EFER);
198         msrval |= MSR_FLAG_STAR_SCE;
199         wrmsr(MSR_EFER, msrval);
200
201         msrval = (uint64_t)(uintptr_t)fast_amd_entry;
202         msrval |= (uint64_t)SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL) << 32;
203         msrval |= (uint64_t)SEG_SELECTOR(GDT_USER_CODE, RPL_USER) << 48;
204
205         wrmsr(MSR_STAR, msrval);
206     }
207 }

```

Here is the call graph for this function:



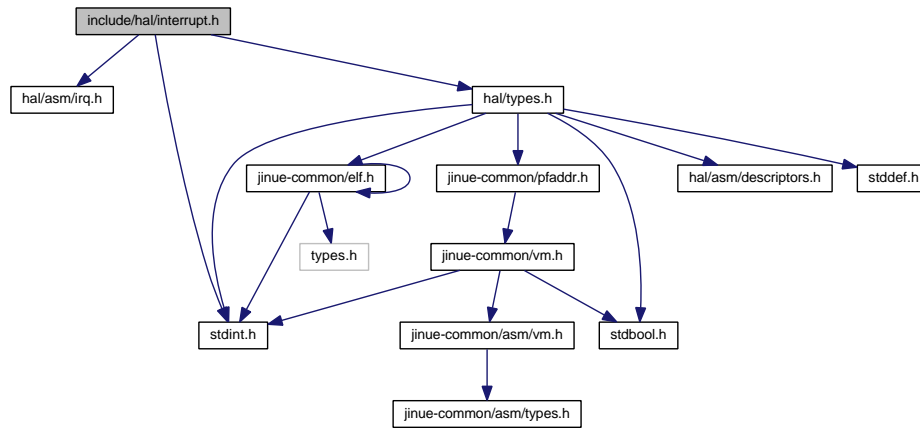
4.32 include/hal/interrupt.h File Reference

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

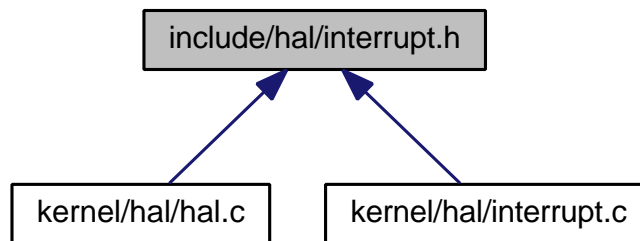
```
#include <hal/types.h>
```

```
#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;

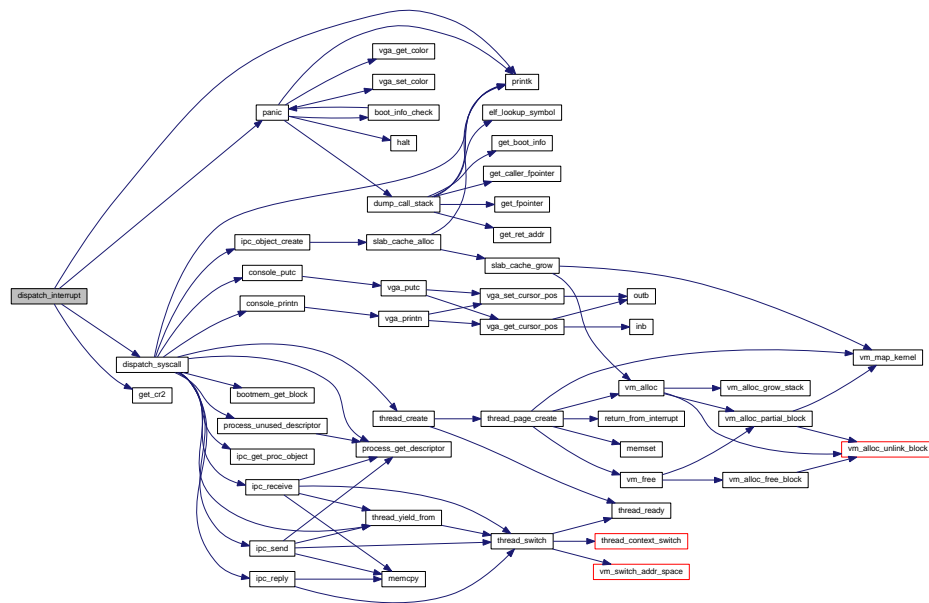
```

```

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:



4.32.2 Variable Documentation

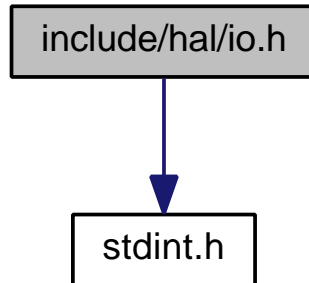
4.32.2.1 seg_descriptor_t idt[]

Referenced by hal_init().

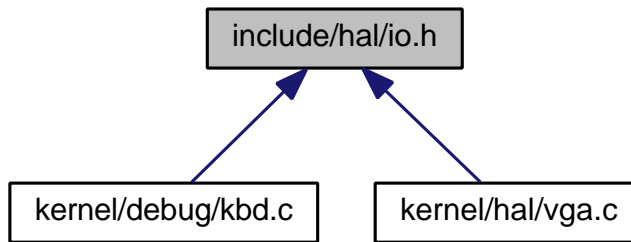
4.33 include/hal/io.h File Reference

```
#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().

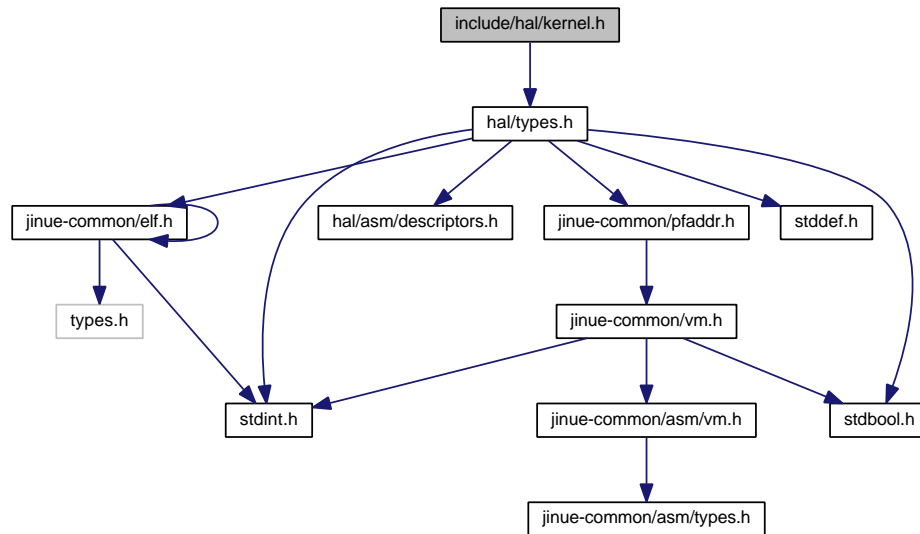
4.33.1.5 void outl (uint16_t port, uint32_t value)

4.33.1.6 void outw (uint16_t port, uint16_t value)

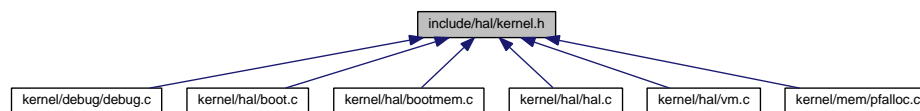
4.34 include/hal/kernel.h File Reference

```
#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

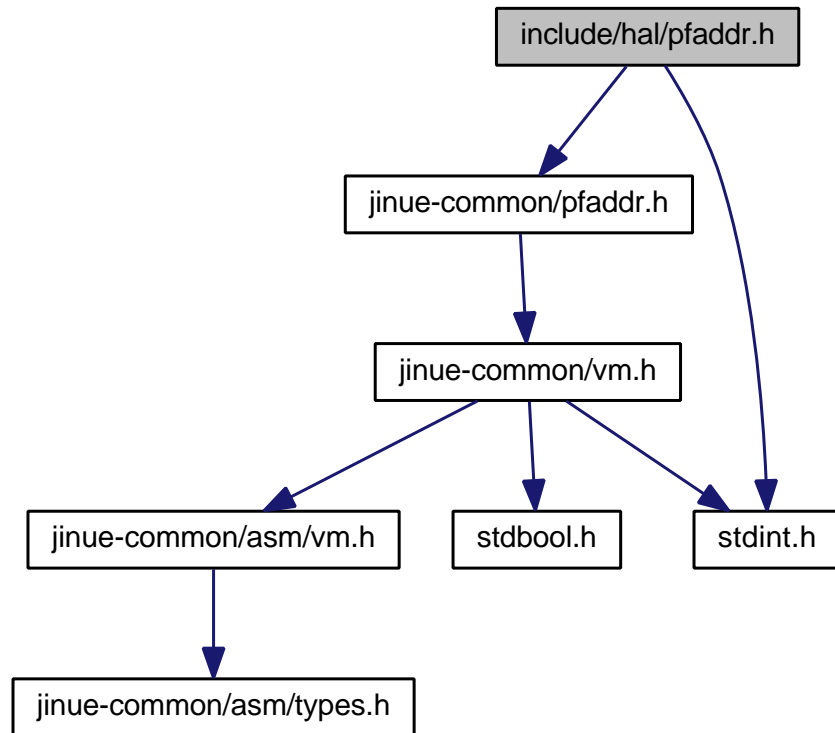
4.34.1.1 addr_t kernel_region_top

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()`.

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



- **#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)`

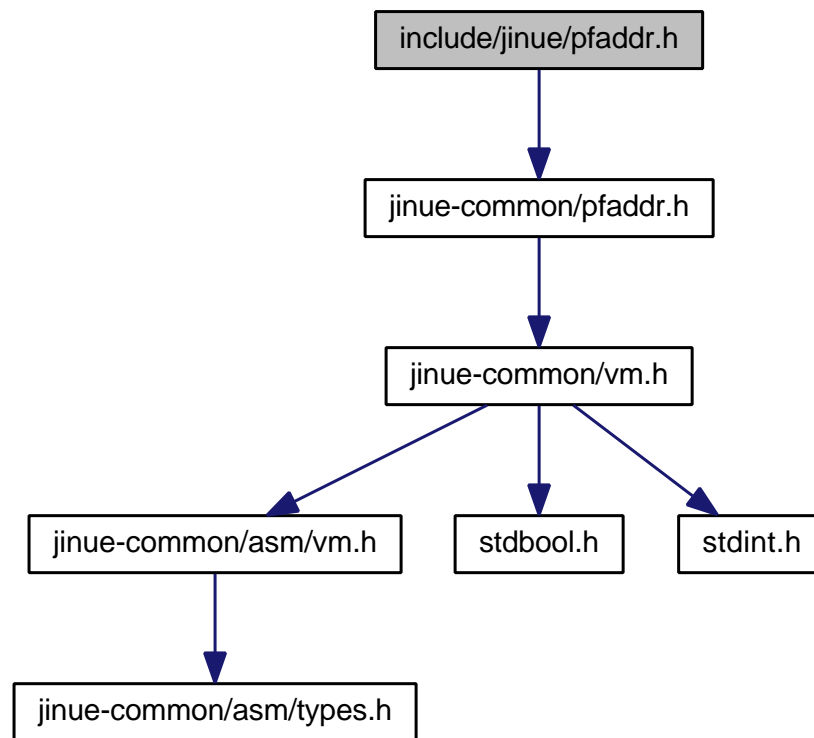
convert a page frame address to an address in an integer

Definition at line 42 of file pfaddr.h.

4.36 `include/jinue/pfaddr.h` File Reference

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

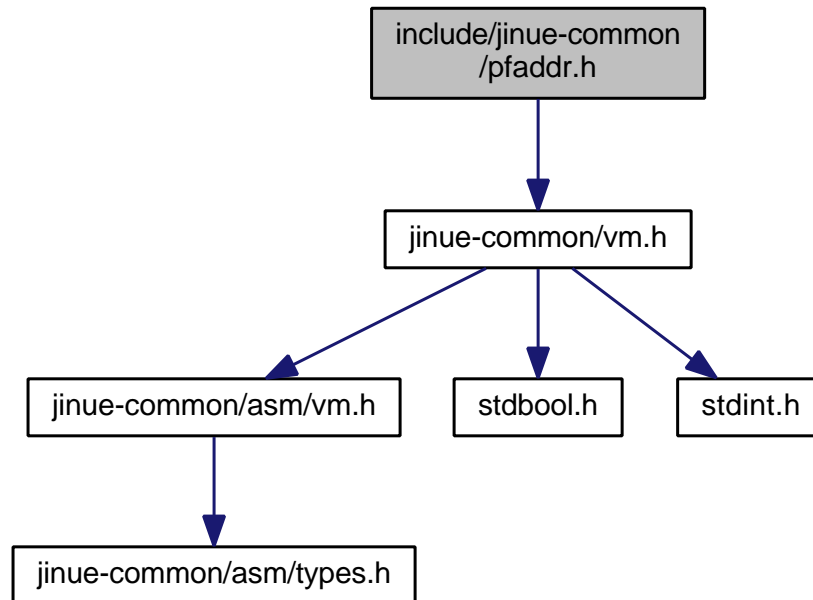
Include dependency graph for pfaddr.h:



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

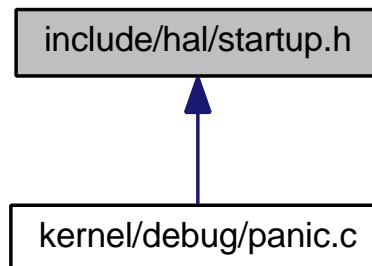
4.37.2.1 typedef uint32_t pfaddr_t

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)

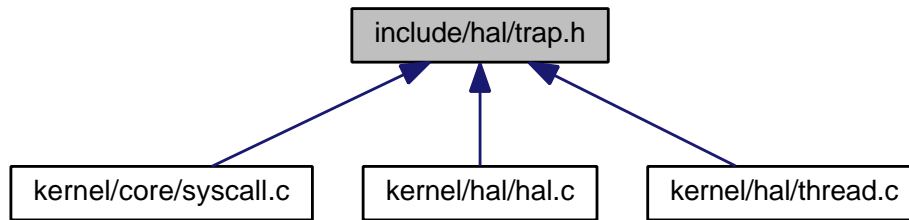
4.38.1 Function Documentation

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.

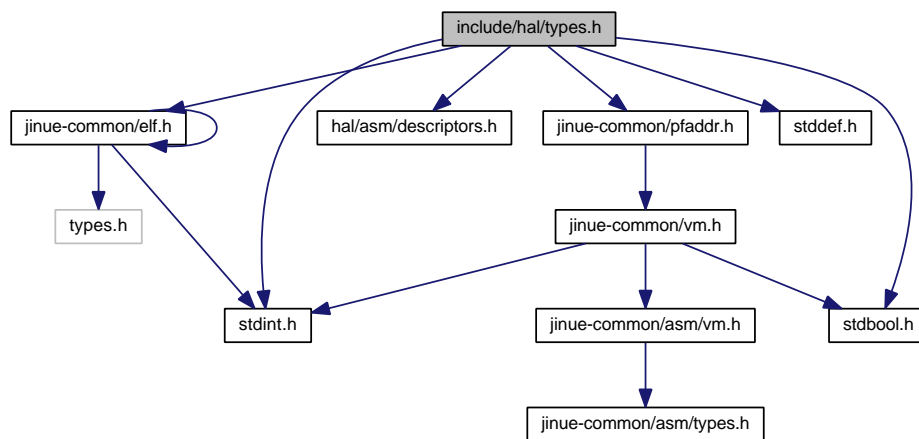
Definition at line 62 of file hal.c.

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.

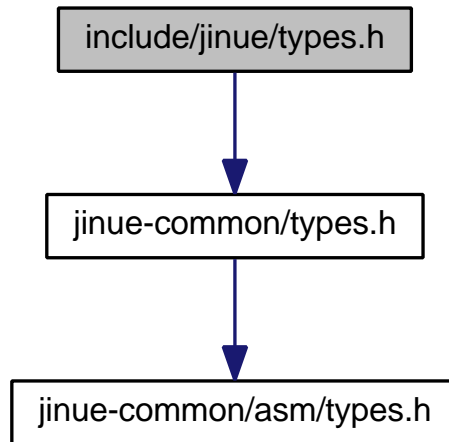
4.40.2.9 `typedef uint32_t seg_selector_t`

Definition at line 112 of file types.h.

4.41 include/jinue/types.h File Reference

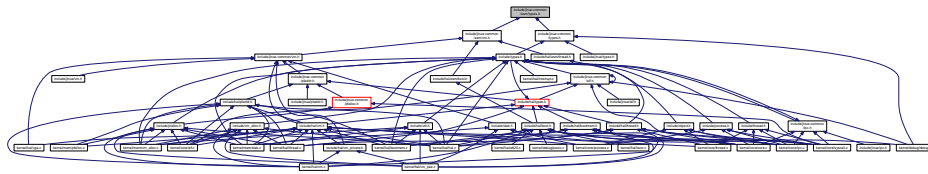
```
#include <jinue-common/types.h>
```

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)`

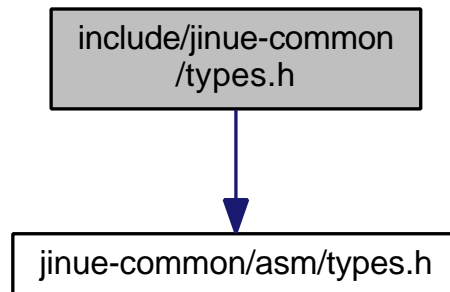
Definition at line 37 of file types.h.

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:



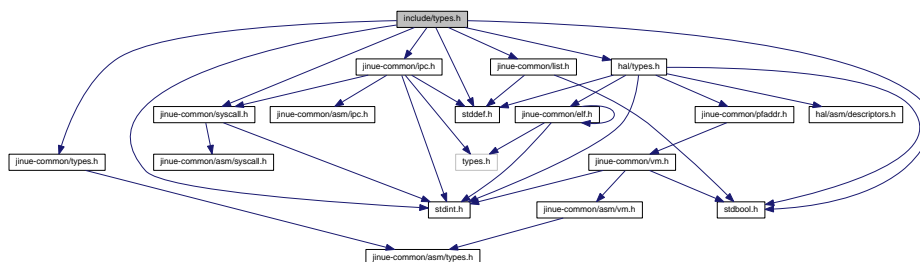
This graph shows which files directly or indirectly include this file:



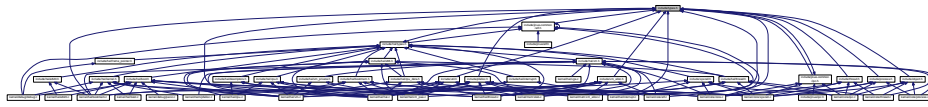
4.44 include/types.h File Reference

```
#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_DESCRIPTOR** 12

Typedefs

- `typedef struct` **thread_t** **thread_t**

4.44.1 Macro Definition Documentation

4.44.1.1 `#define` **PROCESS_MAX_DESCRIPTOR** 12

Definition at line 57 of file `types.h`.

Referenced by `process_get_descriptor()`, and `process_unused_descriptor()`.

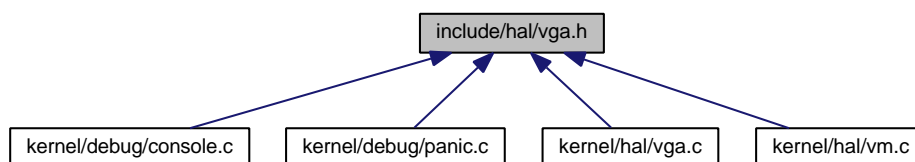
4.44.2 Typedef Documentation

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().

```
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 }
```

4.45.3.2 unsigned int vga_get_color (void)

Definition at line 95 of file vga.c.

Referenced by panic().

```
95     {
96     return vga_text_color;
97 }
```

4.45.3.3 vga_pos_t vga_get_cursor_pos (void)

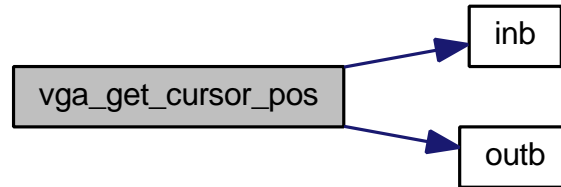
Definition at line 103 of file vga.c.

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

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

```
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:



4.45.3.4 void vga_init (void)

Definition at line 46 of file vga.c.

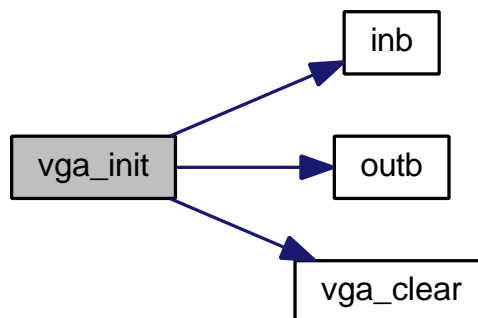
References `inb()`, `outb()`, `vga_clear()`, `VGA_COLOR_DEFAULT`, `VGA_CRTC_ADDR`, `VGA_CRTC_DATA`, `VGA_MISC_OUT_RD`, and `VGA_MISC_OUT_WR`.

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)

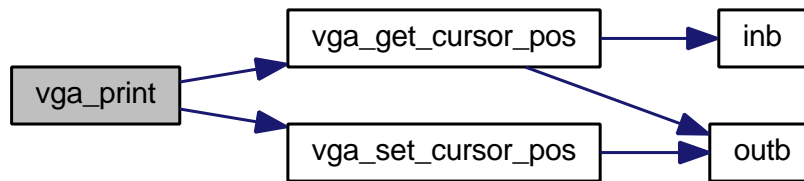
Definition at line 125 of file vga.c.

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

```

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:



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

Definition at line 136 of file `vga.c`.

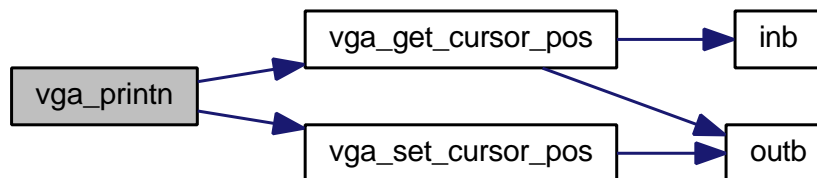
References `vga_get_cursor_pos()`, and `vga_set_cursor_pos()`.

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`.

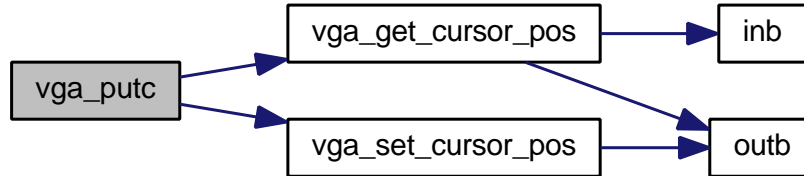
References `vga_get_cursor_pos()`, and `vga_set_cursor_pos()`.

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()`.

```

99                                     {
100     vga_text_color = color;
101 }

```

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.

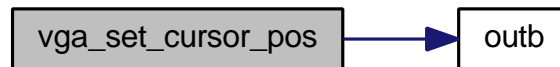
Referenced by vga_print(), vga_printn(), and vga_putc().

```

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 }

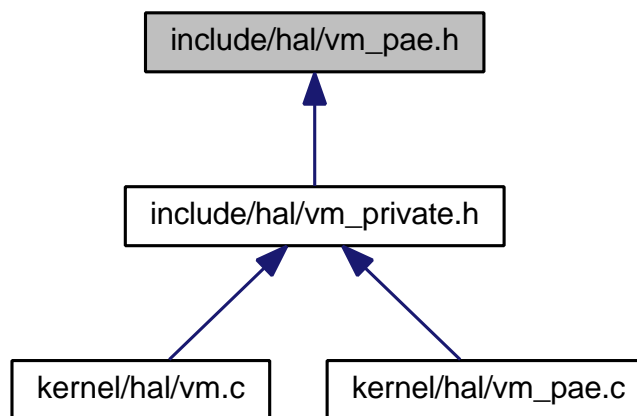
```

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.

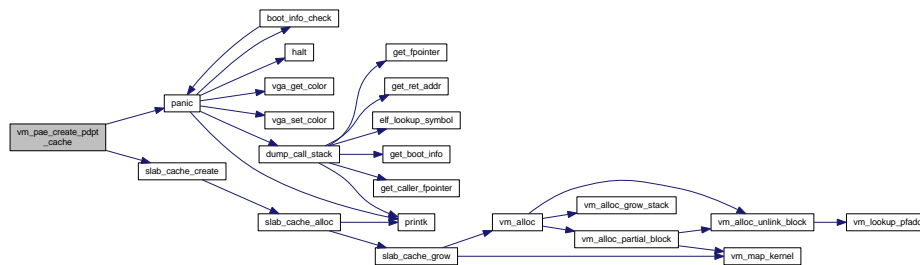
References NULL, panic(), slab_cache_create(), and SLAB_DEFAULTS.

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`.

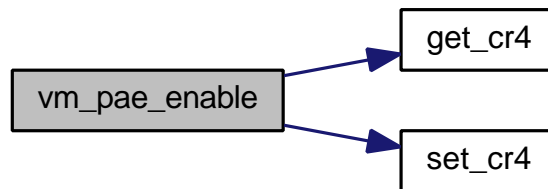
Referenced by `vm_boot_init()`.

```

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

```

Here is the call graph for this function:



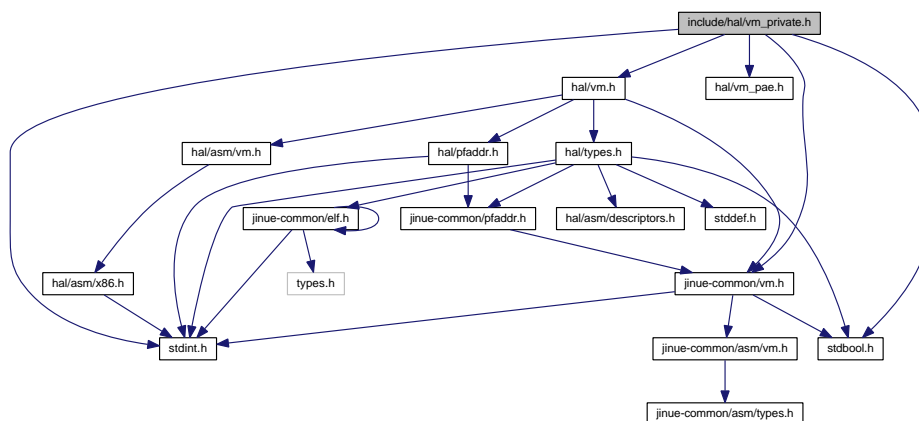
4.47 include/hal/vm_private.h File Reference

```

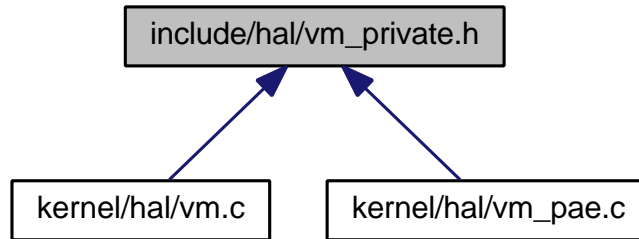
#include <jinue-common/vm.h>
#include <hal/vm.h>
#include <hal/vm_pae.h>
#include <stdbool.h>
#include <stdint.h>

```

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

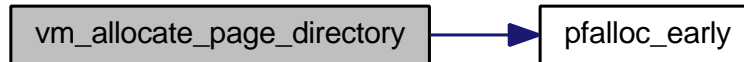
```

```

543     * same address once paging is enabled. */
544     page_table = (pte_t *)pfalloc_early();
545
546     if(first_pd && idx == start_index) {
547         /* remember the address of the first page table for use by
548          * vm_map() later */
549         global_page_tables = page_table;
550     }
551
552     set_pte(
553         get_pte_with_offset(page_directory, idx),
554         EARLY_PTR_TO_PFADDR(page_table),
555         VM_FLAG_PRESENT | VM_FLAG_READ_WRITE );
556
557     /* clear page table */
558     for(idy = 0; idy < page_table_entries; ++idy) {
559         clear_pte( get_pte_with_offset(page_table, idy) );
560     }
561 }
562
563 return page_directory;
564 }

```

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.

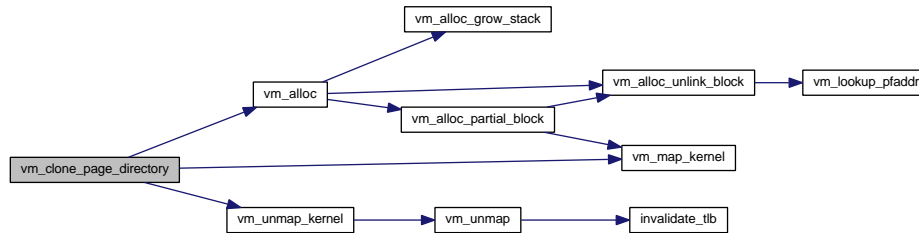
References `clear_pte`, `copy_pte`, `get_pte_with_offset`, `page_table_entries`, `pfalloc`, `vm_alloc()`, `VM_FLAG_READ_WRITE`, `vm_map_kernel()`, and `vm_unmap_kernel()`.

```

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.

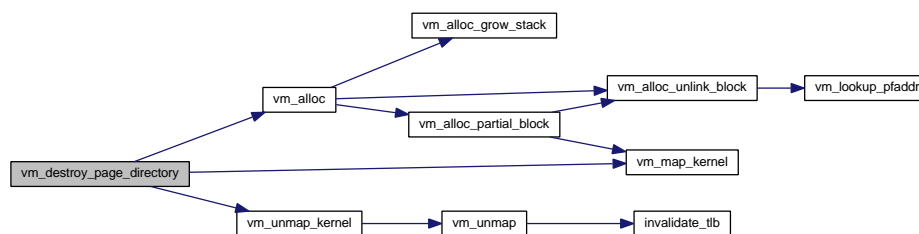
References `get_pte_flags`, `get_pte_pfaddr`, `get_pte_with_offset`, `pffree`, `vm_alloc()`, `VM_FLAG_READ_WRITE`, `vm_map_kernel()`, and `vm_unmap_kernel()`.

```

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)

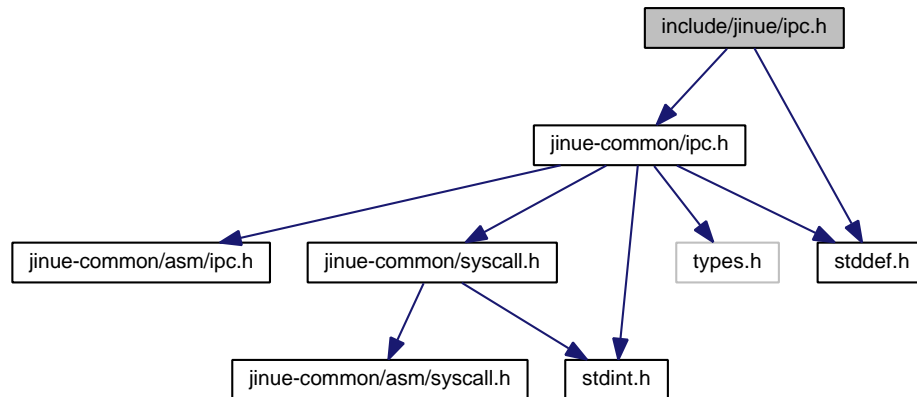
Definition at line 697 of file vm.c.

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*)

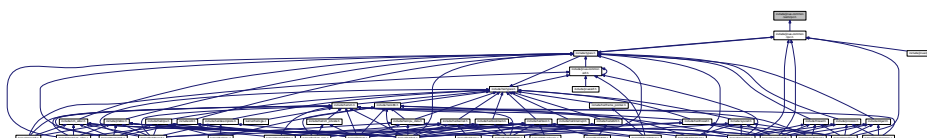
4.49.1.2 int `jinue_receive` (int *fd*, char * *buffer*, **size_t** *buffer_size*, `jinue_message_t` * *message*, int * *perrno*)

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

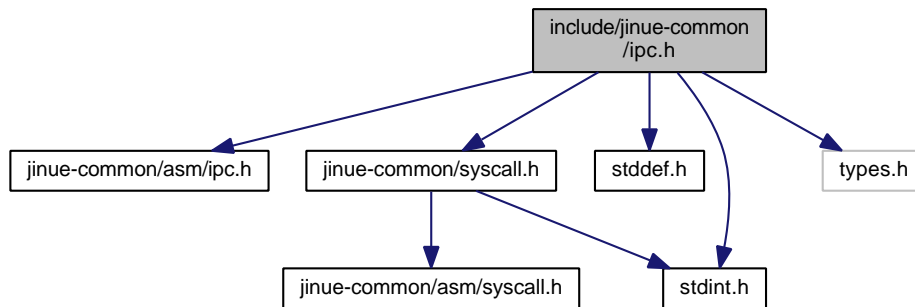
Definition at line 53 of file ipc.h.

Referenced by ipc_reply().

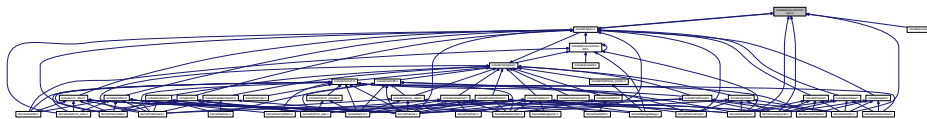
4.51 include/jinue-common/ipc.h File Reference

```
#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()`.

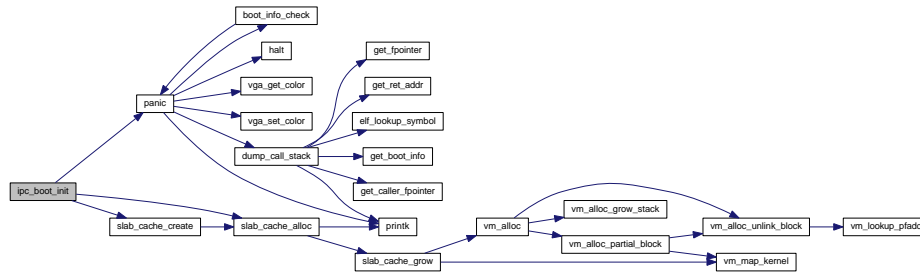
```
58     {
59         ipc_object_cache = slab_cache_create(
60             "ipc_object_cache",
61             sizeof(ipc_t),
62             0,
63             ipc_object_ctor,
64             NULL,
65             SLAB_DEFAULTS );
66     }
```

```

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().

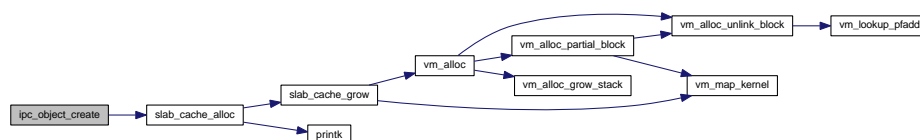
Referenced by dispatch_syscall().

```

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 */

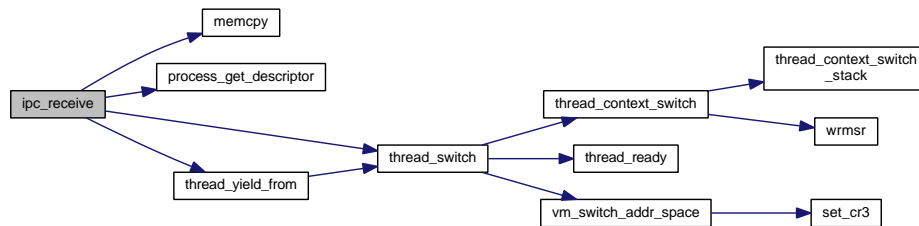
```

```

266     send_thread = thread->sender;
267 }
268 else {
269     object_addrref(&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

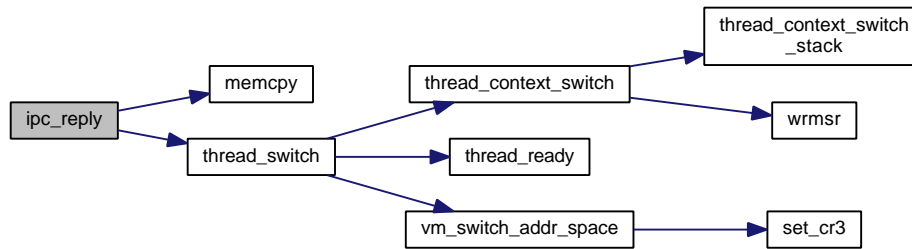
{

```

303     thread_t *thread      = get_current_thread();
304     thread_t *send_thread = thread->sender;
305
306     if(send_thread == NULL) {
307         syscall_args_set_error(args, JINUE_EINVAL);
308         return;
309     }
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
340     if(desc_n > 0) {
341         syscall_args_set_error(args, JINUE_ENOSYS);
342         return;
343     }
344
345     const char *user_ptr = (const char *)args->arg2;
346
347     if(! user_buffer_check(user_ptr, buffer_size)) {
348         syscall_args_set_error(args, JINUE_EINVAL);
349         return;
350     }
351 }
352
353     memcpy(&send_thread->message_buffer, user_ptr, data_size);
354
355     syscall_args_set_return(send_thread->message_args, 0);
356     send_thread->message_args->arg3 =
357         args->arg3 & ~(JINUE_SEND_SIZE_MASK << JINUE_SEND_BUFFER_SIZE_OFFSET);
358
359     send_thread->message_info.data_size = data_size;
360     send_thread->message_info.desc_n   = desc_n;
361
362     object_subref(&send_thread->header);
363     thread->sender = NULL;
364
365     syscall_args_set_return(args, 0);
366
367     /* switch back to sender thread to return from call immediately */
368     thread_switch(
369         thread,
370         send_thread,
371         false, /* don't block (put this thread back in ready queue) */
372         false); /* don't destroy */
373 }

```


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::rcv_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
116         if(message_info->desc_n > 0) {
117             syscall_args_set_error(args, JINUE_ENOSYS);
118             return;
119         }
120     }
121
122     int fd = (int)args->arg1;

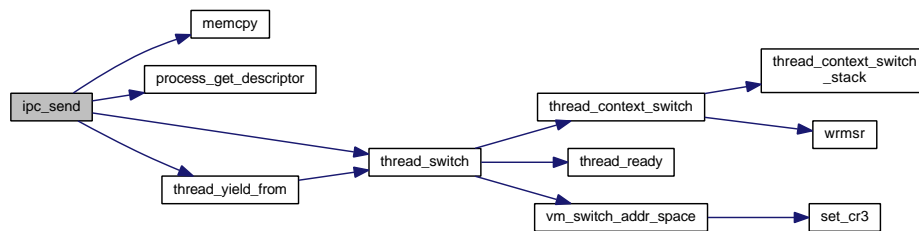
```

```

123
124 object_ref_t *ref = process_get_descriptor(thread->process, fd);
125
126 if(! object_ref_is_valid(ref)) {
127     syscall_args_set_error(args, JINUE_EBADF);
128     return;
129 }
130
131 if(object_ref_is_closed(ref)) {
132     syscall_args_set_error(args, JINUE_EIO);
133     return;
134 }
135
136 message_info->cookie = ref->cookie;
137
138 object_header_t *header = ref->object;
139
140 if(object_is_destroyed(header)) {
141     ref->flags |= OBJECT_REF_FLAG_CLOSED;
142     object_subref(header);
143
144     syscall_args_set_error(args, JINUE_EIO);
145     return;
146 }
147
148 if(header->type != OBJECT_TYPE_IPC) {
149     syscall_args_set_error(args, JINUE_EBADF);
150     return;
151 }
152
153 ipc_t *ipc = (ipc_t *)header;
154
155 char *user_ptr = (char *)args->arg2;
156
157 if(! user_buffer_check(user_ptr, message_info->buffer_size)) {
158     syscall_args_set_error(args, JINUE_EINVAL);
159     return;
160 }
161
162 memcpy(&thread->message_buffer, user_ptr, message_info->data_size);
163
164 /* return values are set by ipc_reply() (or by ipc_receive() if the call
165  * fails because the message is too big for the receiver's buffer) */
166 thread->message_args = args;
167
168 thread_t *recv_thread = jinue_node_entry(
169     jinue_list_dequeue(&ipc->recv_list),
170     thread_t,
171     thread_list);
172
173 if(recv_thread == NULL) {
174     /* No thread is waiting to receive this message, so we must wait on the
175      * sender list. */
176     jinue_list_enqueue(&ipc->send_list, &thread->thread_list);
177
178     thread_yield_from(
179         thread,
180         true,      /* make thread block */
181         false);   /* don't destroy */
182 }
183 else {
184     object_addrref(&thread->header);
185     recv_thread->sender = thread;
186
187     /* switch to receiver thread, which will resume inside syscall_receive() */
188     thread_switch(
189         thread,
190         recv_thread,
191         true,      /* block sender thread */
192         false);   /* don't destroy sender */
193 }
194
195 /* copy reply to user space buffer */
196 memcpy(user_ptr, &thread->message_buffer, message_info->data_size);
197
198
199
200 }

```

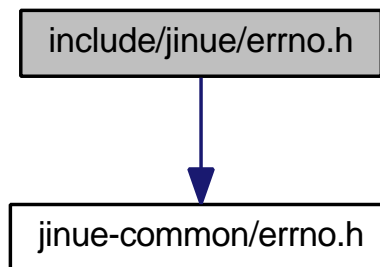
Here is the call graph for this function:



4.52 include/jinue/errno.h File Reference

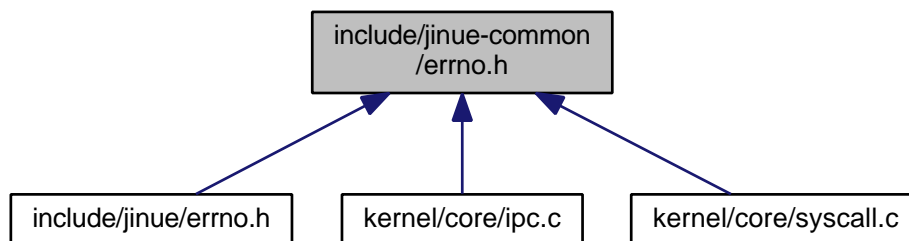
```
#include <jinue-common/errno.h>
```

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

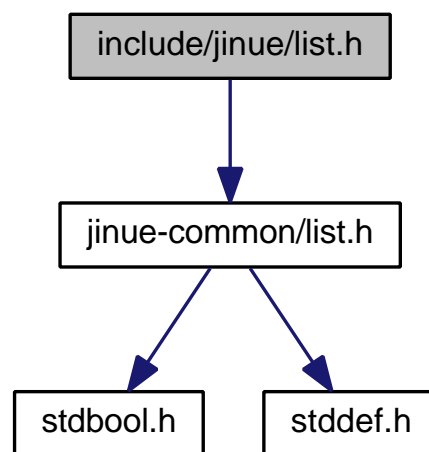
Definition at line 49 of file errno.h.

Referenced by ipc_receive().

4.54 include/jinue/list.h File Reference

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

Include dependency graph for list.h:

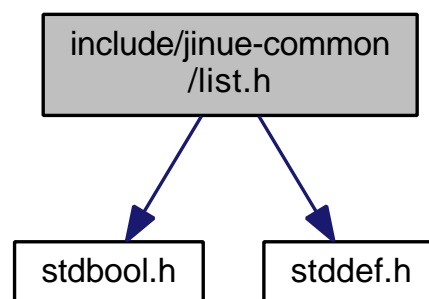


4.55 include/jinue-common/list.h File Reference

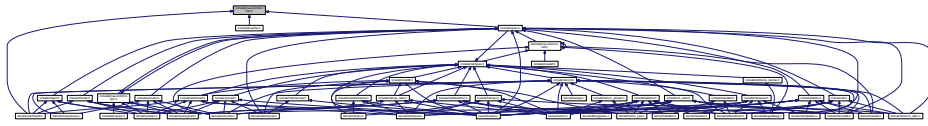
```
#include <stdbool.h>
```

```
#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.

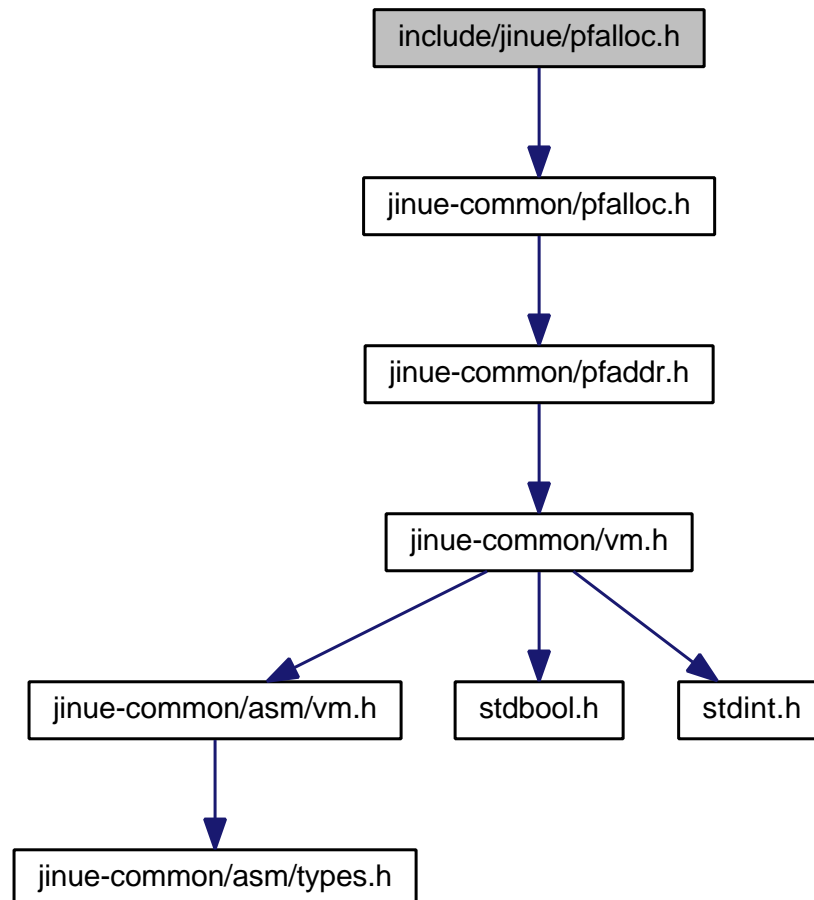
4.55.2.2 `typedef struct jinue_node_t jinue_node_t`

Definition at line 42 of file list.h.

4.56 include/jinue/pfalloc.h File Reference

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

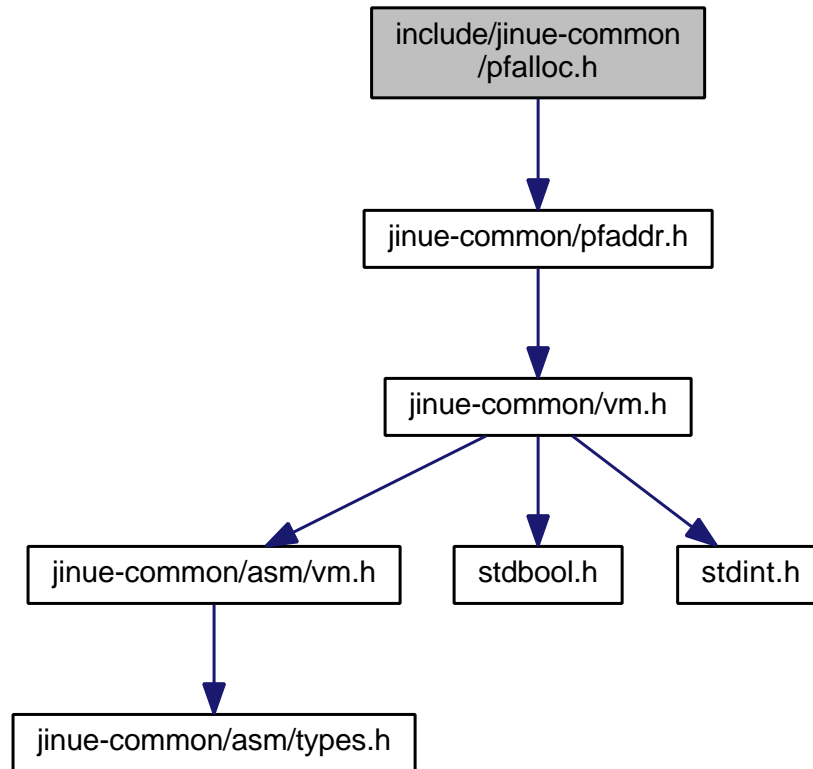
Include dependency graph for pfalloc.h:



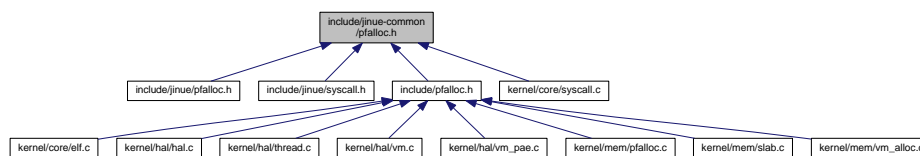
4.57 include/jinue-common/pfalloc.h File Reference

```
#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.

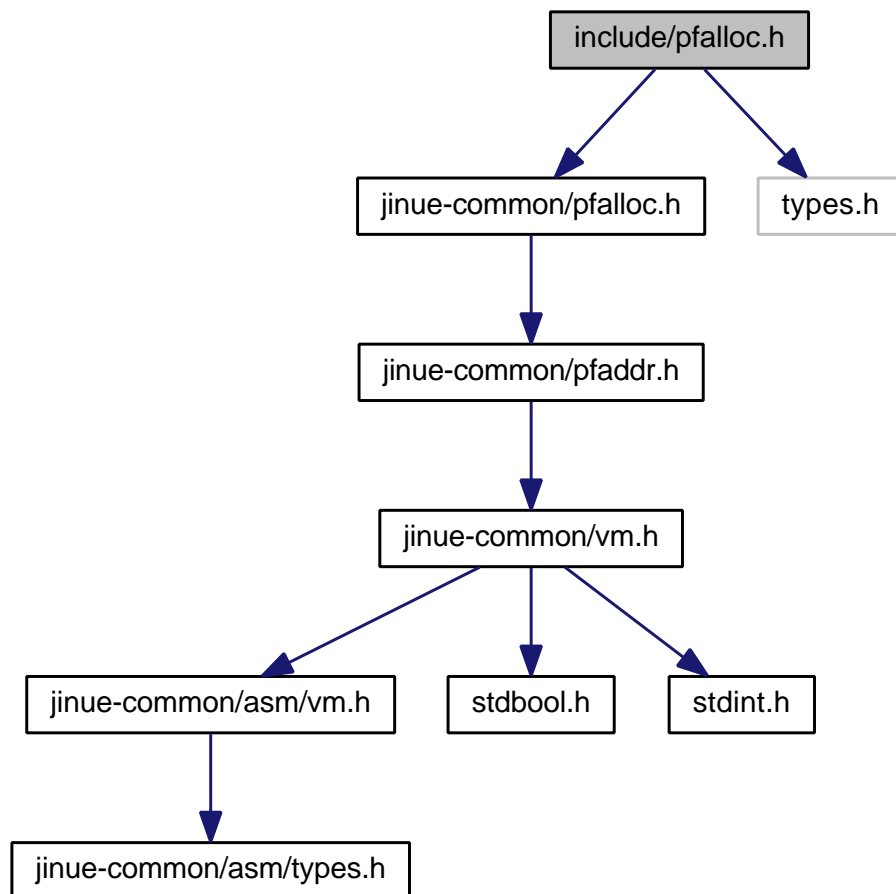
Referenced by init_pfcache(), and pffree_to().

4.58 include/pfalloc.h File Reference

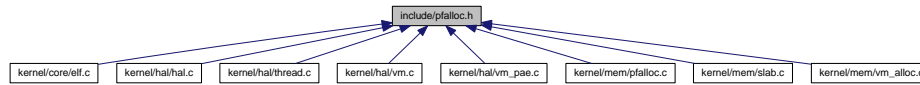
```
#include <jinue-common/pfalloc.h>
```

```
#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 **pf-free(p)** **pf-free_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 **pf-free_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 pf-free(p) pf-free_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
49     assert(use_pfalloc_early);
50
51     page = kernel_region_top;
52     kernel_region_top += PAGE_SIZE;
53
54     return page;
55 }
56 }
```

4.58.2.3 `pfaddr_t pfalloc_from (pfcache_t * pfcache)`

ASSERTION: `pfalloc_early` must be used early

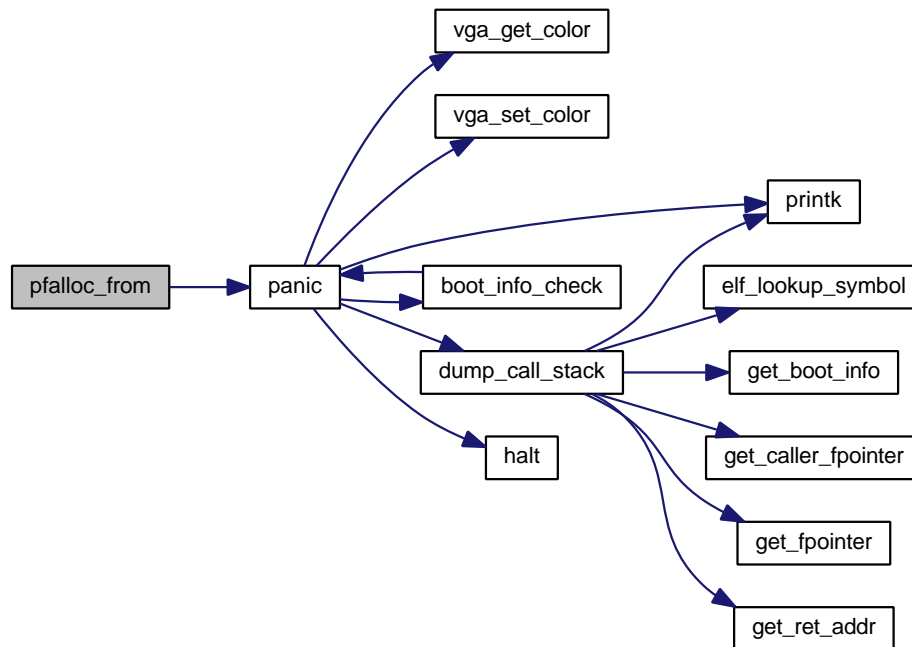
Definition at line 72 of file `pfalloc.c`.

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

```

72                                     {
73     assert( ! use_pfalloc_early );
74
75     if(pfcache->count == 0) {
76         panic("pfalloc_from(): no more pages to allocate");
77     }
78
79     --pfcache->count;
80
81     return *(--pfcache->ptr);
82 }
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) {
87             return;
88         }
89     }
90     ++pfcache->count;
91     (pfcache->ptr++)[0] = pf;
92 }

```

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

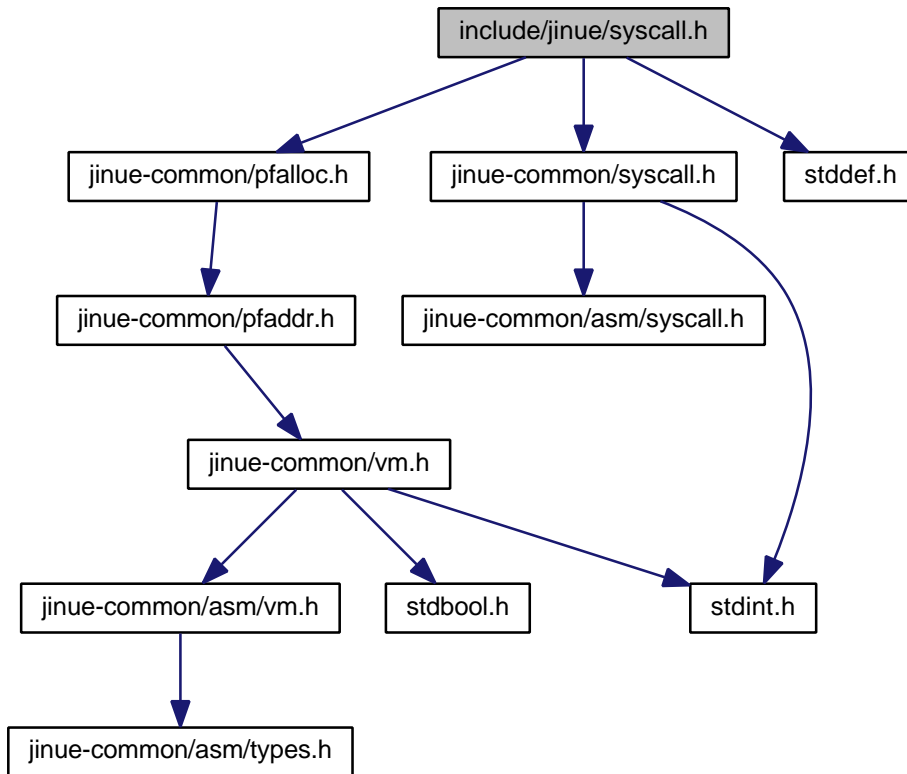
Definition at line 41 of file pfalloc.c.

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)

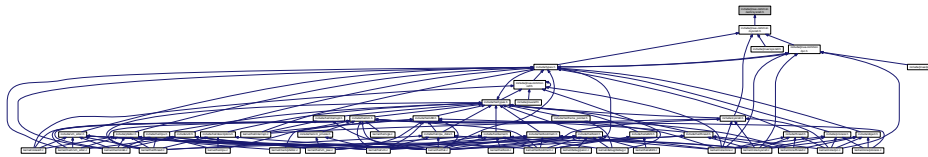
4.59.1 Function Documentation

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.

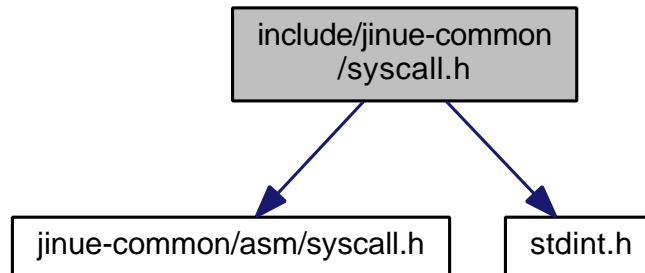
Referenced by hal_init().

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

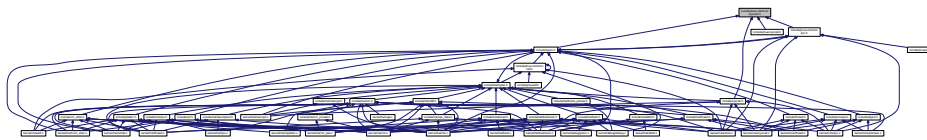
```
#include <jinue-common/asm/syscall.h>
```

```
#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`

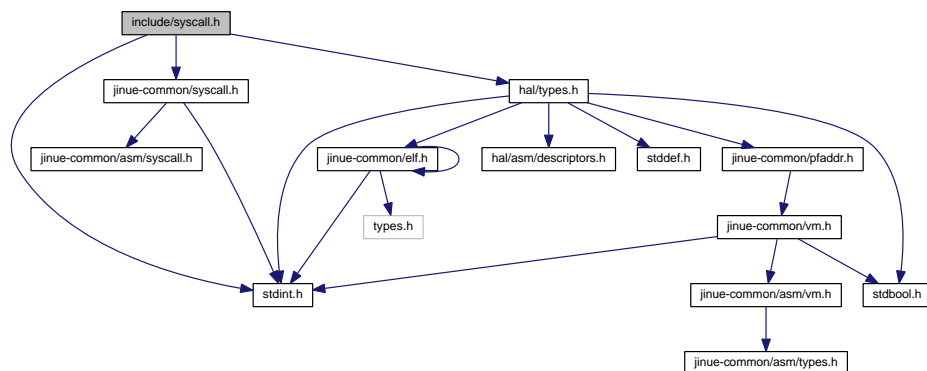
4.62 include/syscall.h File Reference

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

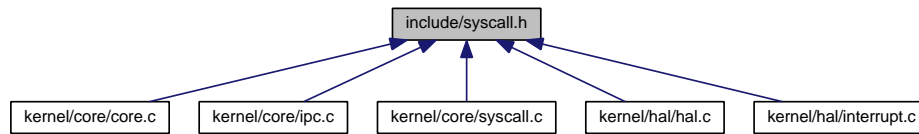
```
#include <hal/types.h>
```

```
#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
52     uintptr_t function_number = args->arg0;
53
54     if(function_number < SYSCALL_FUNCT_PROC_BASE) {
55         /* microkernel system calls */
56         switch(function_number) {
57
58             case SYSCALL_FUNCT_SYSCALL_METHOD:
59                 syscall_args_set_return(args, syscall_method);
60                 break;
61
62             case SYSCALL_FUNCT_CONSOLE_PUTC:
63                 console_putc((char)args->arg1);
64                 syscall_args_set_return(args, 0);
65                 break;
66
67             case SYSCALL_FUNCT_CONSOLE_PUTS:
68                 console_printn((char *)args->arg2, jinue_args_get_data_size(args));
69                 syscall_args_set_return(args, 0);
70         }
71     }
72 }

```

```

73         break;
74
75     case SYSCALL_FUNCT_THREAD_CREATE:
76     {
77         thread_t *thread = thread_create(
78             /* TODO use arg1 as an address space reference if specified */
79             get_current_thread()->process,
80             (addr_t)args->arg2,
81             (addr_t)args->arg3);
82
83         if(thread == NULL) {
84             syscall_args_set_error(args, JINUE_EAGAIN);
85         }
86         else {
87             syscall_args_set_return(args, 0);
88         }
89     }
90     break;
91
92     case SYSCALL_FUNCT_THREAD_YIELD:
93         thread_yield_from(
94             get_current_thread(),
95             false, /* don't block */
96             args->arg1); /* destroy (aka. exit) thread if true */
97         syscall_args_set_return(args, 0);
98         break;
99
100    case SYSCALL_FUNCT_SET_THREAD_LOCAL_ADDR:
101        thread_context_set_local_storage(
102            &get_current_thread()->thread_ctx,
103            (addr_t)args->arg1,
104            (size_t)args->arg2);
105        syscall_args_set_return(args, 0);
106        break;
107
108    case SYSCALL_FUNCT_GET_THREAD_LOCAL_ADDR:
109        syscall_args_set_return_ptr(
110            args,
111            thread_context_get_local_storage(
112                &get_current_thread()->thread_ctx));
113        break;
114
115    case SYSCALL_FUNCT_GET_FREE_MEMORY:
116    {
117        bootmem_t *block;
118        memory_block_t *block_dest;
119        unsigned int count, count_max;
120
121        size_t buffer_size = jinue_args_get_buffer_size(args);
122        block_dest = (memory_block_t *)jinue_args_get_buffer_ptr(args);
123
124        count_max = buffer_size / sizeof(memory_block_t);
125
126        for(count = 0; count < count_max; ++count) {
127            block = bootmem_get_block();
128
129            if(block == NULL) {
130                break;
131            }
132
133            block_dest->addr = block->addr;
134            block_dest->count = block->count;
135
136            ++block_dest;
137        }
138
139        args->arg0 = (uintptr_t)count;
140
141        if(count == count_max && bootmem_root != NULL) {
142            args->arg1 = JINUE_EMORE;
143        }
144        else {
145            args->arg1 = 0;
146        }
147
148        args->arg2 = 0;
149        args->arg3 = 0;
150    }
151    break;
152
153    case SYSCALL_FUNCT_CREATE_IPC:

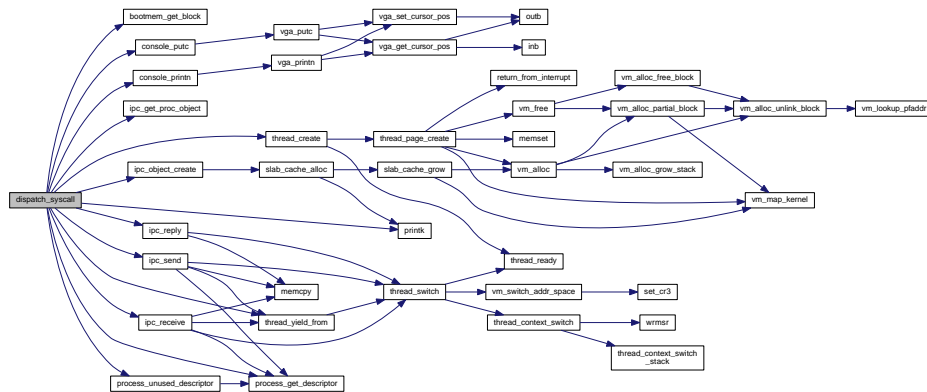
```

```

155     {
156         ipc_t *ipc;
157
158         thread_t *thread = get_current_thread();
159
160         int fd = process_unused_descriptor(thread->process);
161
162         if(fd < 0) {
163             syscall_args_set_error(args, JINUE_EAGAIN);
164             break;
165         }
166
167         if(args->arg1 & JINUE_IPC_PROC) {
168             ipc = ipc_get_proc_object();
169         }
170         else {
171             int flags = IPC_FLAG_NONE;
172
173             if(args->arg1 & JINUE_IPC_SYSTEM) {
174                 flags |= IPC_FLAG_SYSTEM;
175             }
176
177             ipc = ipc_object_create(flags);
178
179             if(ipc == NULL) {
180                 syscall_args_set_error(args, JINUE_EAGAIN);
181                 break;
182             }
183         }
184
185         object_ref_t *ref = process_get_descriptor(thread->process, fd);
186
187         object_addrref(&ipc->header);
188
189         ref->object = &ipc->header;
190         ref->flags = OBJECT_REF_FLAG_VALID | OBJECT_REF_FLAG_OWNER;
191         ref->cookie = 0;
192
193         syscall_args_set_return(args, fd);
194     }
195
196     break;
197 case SYSCALL_FUNCT_RECEIVE:
198     ipc_receive(args);
199     break;
200
201 case SYSCALL_FUNCT_REPLY:
202     ipc_reply(args);
203     break;
204
205 default:
206     printk("SYSCALL: function %u arg1=%u(0x%x) arg2=%u(0x%x) arg3=%u(0x%x)\n",
207           function_number,
208           args->arg1, args->arg1,
209           args->arg2, args->arg2,
210           args->arg3, args->arg3 );
211
212     syscall_args_set_error(args, JINUE_ENOSYS);
213 }
214 }
215 else if(function_number < SYSCALL_FUNCT_SYSTEM_BASE) {
216     /* process manager system calls */
217     printk("PROC SYSCALL: function %u arg1=%u(0x%x) arg2=%u(0x%x) arg3=%u(0x%x)\n",
218           function_number,
219           args->arg1, args->arg1,
220           args->arg2, args->arg2,
221           args->arg3, args->arg3 );
222
223     syscall_args_set_error(args, JINUE_ENOSYS);
224 }
225 else {
226     /* inter-process message */
227     ipc_send(args);
228 }
229 }

```

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 */

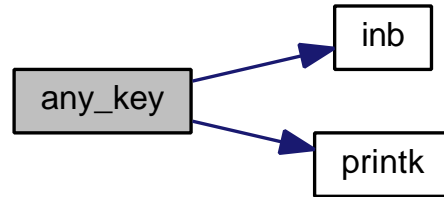
```

```

68     printk("\n");
69 }

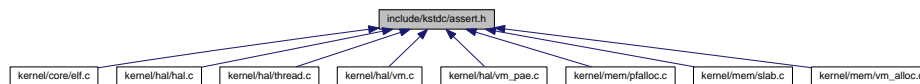
```

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_pfnaddr(), 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*)

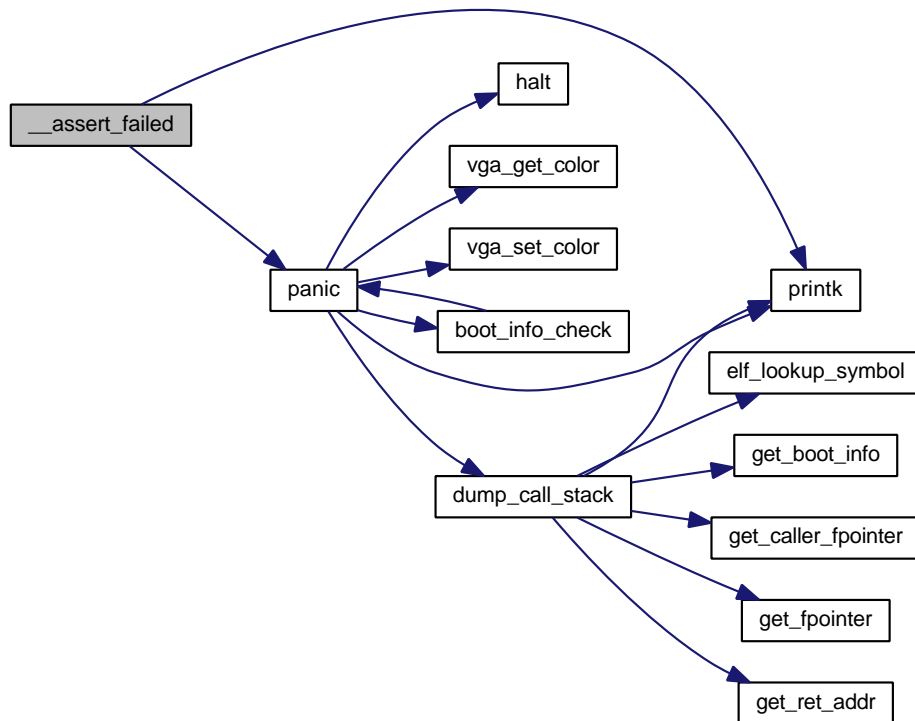
Definition at line 36 of file assert.c.

References panic(), and printk().

```

40         {
41
42     printk(
43         "ASSERTION FAILED [%s]: %s at line %u in function %s.\n",
44         expr, file, line, func );
45
46     panic("Assertion failed.");
47 }
```

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.

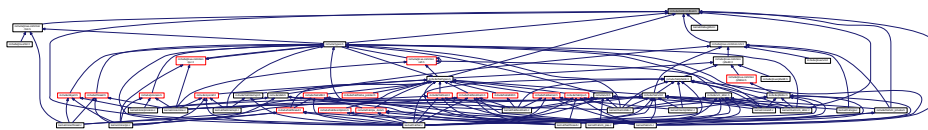
4.65.2 Typedef Documentation

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`

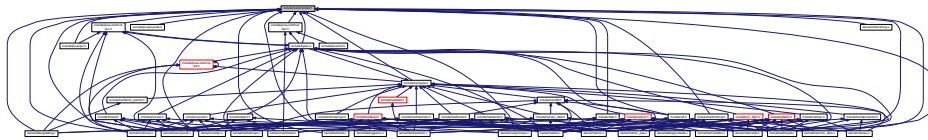
Definition at line 37 of file `stdbool.h`.

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.

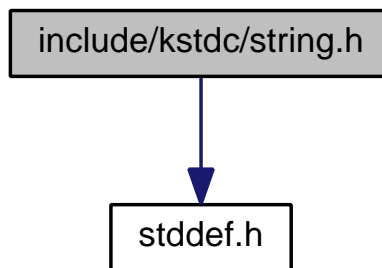
4.68.2.10 typedef unsigned int uintptr_t

Definition at line 59 of file stdint.h.

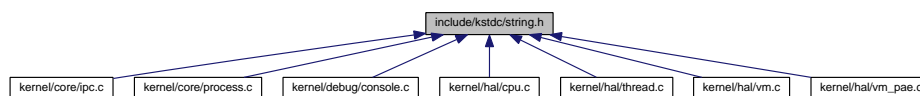
4.69 include/kstdc/string.h File Reference

```
#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().

```

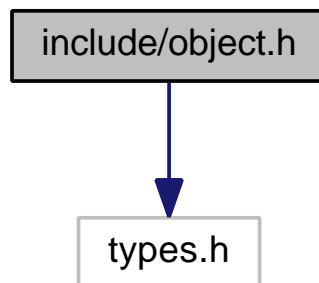
57                                     {
58     size_t count = 0;
59
60     while(*s != 0) {
61         ++s;
62         ++count;
63     }
64
65     return count;
66 }

```

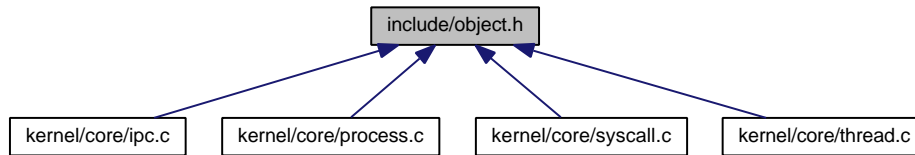
4.70 include/object.h File Reference

```
#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.

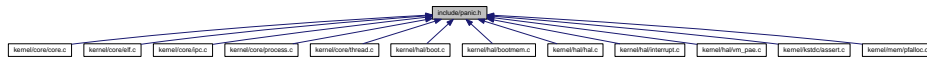
4.70.1.9 #define OBJECT_TYPE_THREAD 1

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.

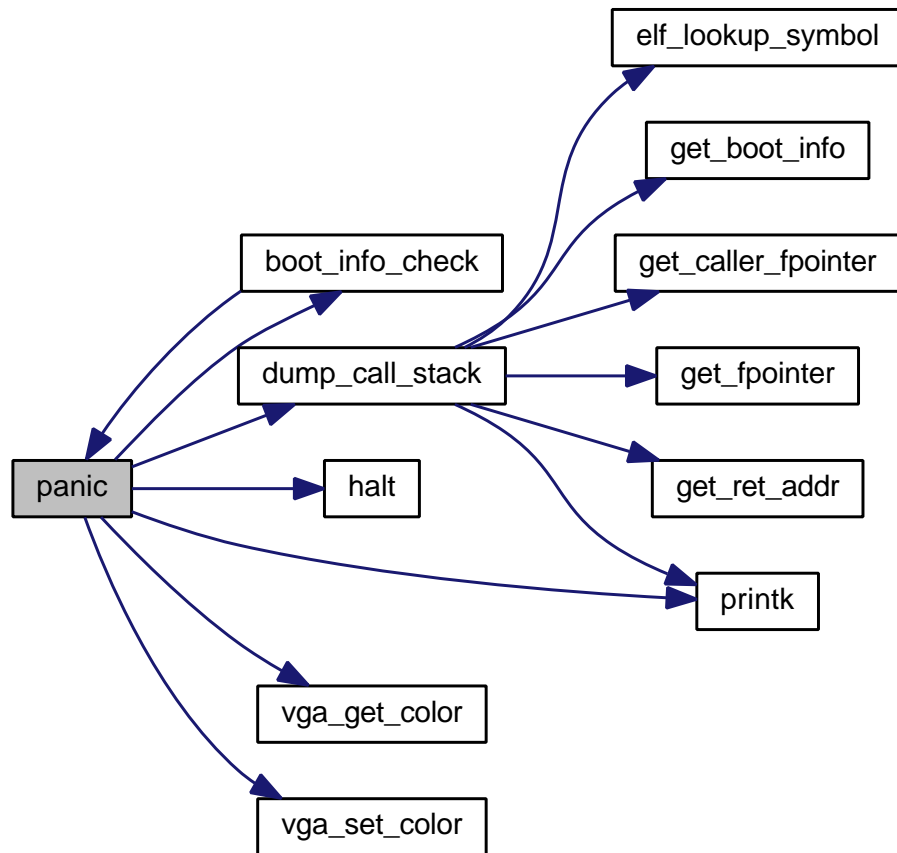
References boot_info_check(), dump_call_stack(), halt(), printk(), VGA_COLOR_RED, vga_get_color(), and vga_set_color().

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);
  
```


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*)

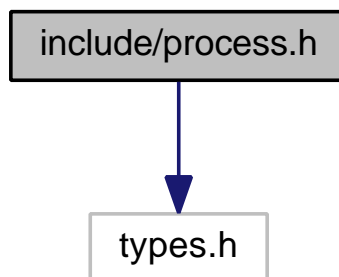
4.72.1.7 void **printk** (const char * *format*, ...)

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()`.

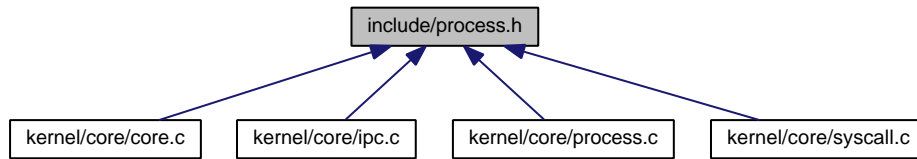
4.73 include/process.h File Reference

```
#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.

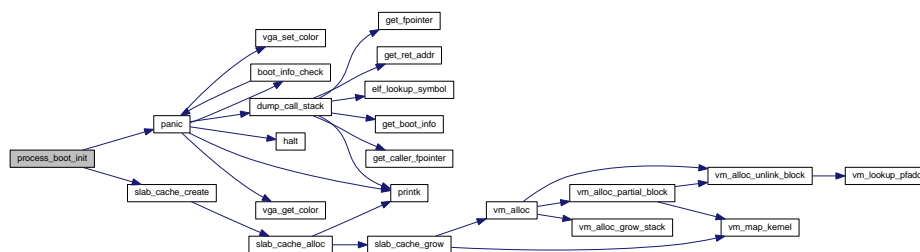
Referenced by kmain().

```

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()`.

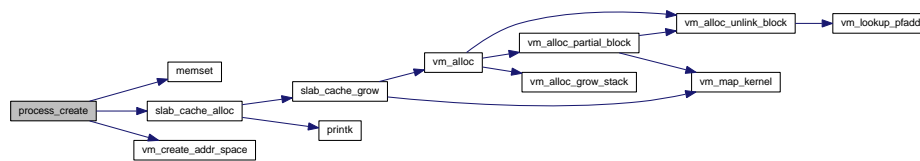
Referenced by `kmain()`.

```

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_DESCRIPTOR`.

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

```

74     {
75     if (fd < 0 || fd > PROCESS_MAX_DESCRIPTOR) {
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_DESCRIPTOR`.

Referenced by `dispatch_syscall()`.

```

82     {
83     int idx;
84
85     for (idx = 0; idx < PROCESS_MAX_DESCRIPTOR; ++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:

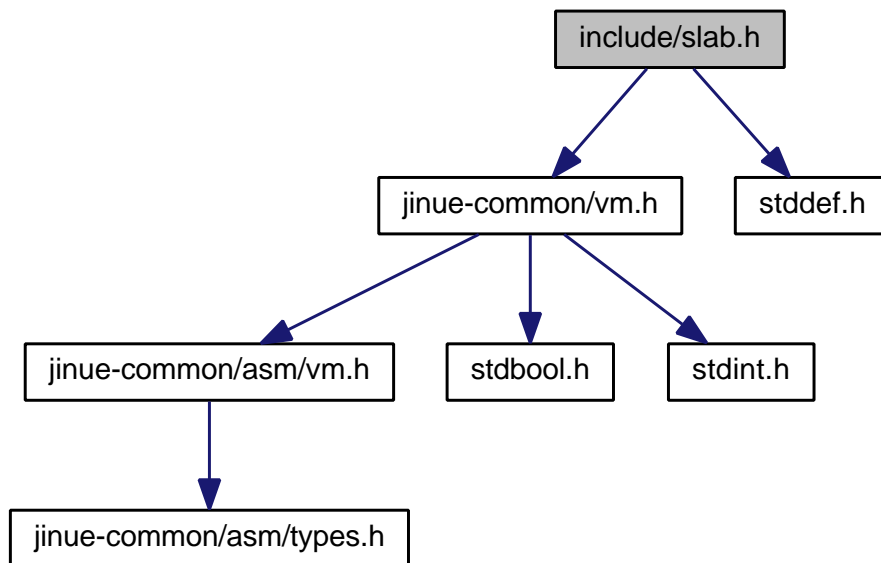


4.74 include/slab.h File Reference

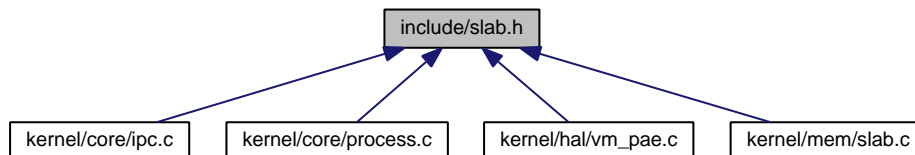
```
#include <jinue-common/vm.h>
```

```
#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`, `printf()`, `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
245         assert(slab != NULL);
246
247         /* We are about to allocate one object from this slab, so it will
248          * not be empty anymore...*/
249         cache->slabs_empty = slab->next;
250
251         --(cache->empty_count);
252
253         slab->next = cache->slabs_partial;
254         if(slab->next != NULL) {
255             slab->next->prev = slab;
256         }
257         cache->slabs_partial = slab;
258     }
259
260     bufctl = slab->free_list;
261
262

```

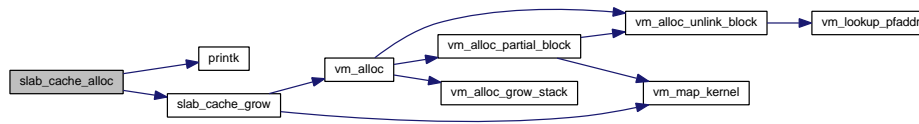


```

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
283         assert(cache->slabs_partial == slab);
284
285         cache->slabs_partial = slab->next;
286
287         if(slab->next != NULL) {
288             slab->next->prev = slab->prev;
289         }
290
291         /* add to the full slabs list */
292         slab->next = cache->slabs_full;
293         cache->slabs_full = slab;
294
295         if(slab->next != NULL) {
296             slab->next->prev = slab;
297         }
298     }
299
300     buffer = (uint32_t *) ( (char *)bufctl - cache->bufctl_offset );
301
302     if(cache->flags & SLAB_POISON) {
303         dump_lines = 0;
304
305         for(idx = 0; idx < cache->obj_size / sizeof(uint32_t); ++idx) {
306             if(buffer[idx] != SLAB_POISON_DEAD_VALUE) {
307                 if(dump_lines == 0) {
308                     printk("detected write to freed object, cache: %s buffer: 0x%x:\n",
309                            cache->name,
310                            (unsigned int)buffer
311                        );
312                 }
313             }
314
315             if(dump_lines < 4) {
316                 printk(" value 0x%x at byte offset %u\n", buffer[idx], idx * sizeof(
uint32_t));
317             }
318
319             ++dump_lines;
320         }
321
322         buffer[idx] = SLAB_POISON_ALIVE_VALUE;
323     }
324
325     /* If both SLAB_POISON and SLAB_RED_ZONE are enabled, we perform
326      * redzone checking even on freed objects. */
327     if(cache->flags & SLAB_RED_ZONE) {
328         if(buffer[idx] != SLAB_RED_ZONE_VALUE) {
329             printk("detected write past the end of freed object, cache: %s buffer: 0x%x value: 0x%x\n",
330                    cache->name,
331                    (unsigned int)buffer,
332                    buffer[idx]
333                );
334         }
335
336         buffer[idx] = SLAB_RED_ZONE_VALUE;
337     }
338
339     if(cache->ctor != NULL) {
340         cache->ctor((void *)buffer, cache->obj_size);
341     }
342 }
343 else if(cache->flags & SLAB_RED_ZONE) {
344     buffer[cache->obj_size / sizeof(uint32_t)] = SLAB_RED_ZONE_VALUE;
345 }
346
347 return (void *)buffer;
348 }

```

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().

```

95         {
96
97     slab_cache_t    *cache;
98     size_t          avail_space;
99     size_t          wasted_space;
100     unsigned int    buffers_per_slab;
101
102     assert( size >= sizeof(void *) );
103
104     assert(name != NULL);
105
106     cache = slab_cache_alloc(&slab_cache_cache);
107
108     cache->name          = name;
109     cache->ctor           = ctor;
110     cache->dtor           = dtor;
111     cache->slabs_empty    = NULL;
112     cache->slabs_partial  = NULL;
113     cache->slabs_full     = NULL;
114     cache->empty_count    = 0;
115     cache->flags          = flags;
116     cache->next_colour    = 0;
117     cache->working_set    = SLAB_DEFAULT_WORKING_SET;
118
119     /* add new cache to cache list */
120     cache->next           = slab_cache_list;
121     slab_cache_list      = cache;
122
123     if(cache->next != NULL) {
124         cache->next->prev = cache;
125     }
126
127     /* compute actual alignment */
128     if(alignment == 0) {
129         cache->alignment = sizeof(uint32_t);
130     }
131     else {
132         cache->alignment = alignment;
133     }
134
135     if((flags & SLAB_HWCACHE_ALIGN) && cache->alignment < cpu_info.
136     dcache_alignment) {
137         cache->alignment = cpu_info.dcache_alignment;
138     }

```

```

139     }
140
141     if(cache->alignment % sizeof(uint32_t) != 0) {
142         cache->alignment += sizeof(uint32_t) - cache->alignment % sizeof(
uint32_t);
143     }
144
145     /* reserve space for bufctl and/or redzone word */
146     cache->obj_size = size;
147
148     if(cache->obj_size % sizeof(uint32_t) != 0) {
149         cache->obj_size += sizeof(uint32_t) - cache->obj_size % sizeof(uint32_t);
150     }
151
152     if((flags & SLAB_POISON) && (flags & SLAB_RED_ZONE)) {
153         /* bufctl and redzone word appended to buffer */
154         cache->alloc_size = cache->obj_size + sizeof(uint32_t) + sizeof(
slab_bufctl_t);
155     }
156     else if((flags & SLAB_POISON) || (flags & SLAB_RED_ZONE)) {
157         /* bufctl and/or redzone word appended to buffer
158          * (can be shared) */
159         cache->alloc_size = cache->obj_size + sizeof(uint32_t);
160     }
161     else if(ctor != NULL && ! (flags & SLAB_COMPACT)) {
162         /* If a constructor is defined, we cannot put the bufctl inside
163          * the object because that could overwrite constructed state,
164          * unless client explicitly says it's ok (SLAB_COMPACT flag). */
165         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:



4.74.3.3 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     {

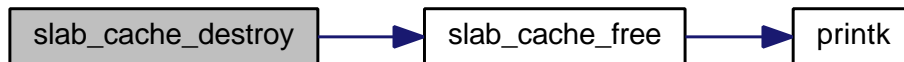
```

```

189     slab_t      *slab;
190     slab_t      *next;
191     unsigned int  empty_count;
192
193     assert(cache->slabs_full == NULL && cache->slabs_partial == NULL);
194
195     /* remove from cache list */
196     if(slab_cache_list == cache) {
197         slab_cache_list = cache->next;
198     }
199     else {
200         cache->prev->next = cache->next;
201     }
202
203     if(cache->next != NULL) {
204         cache->next->prev = cache->prev;
205     }
206
207     /* release all slabs */
208     slab = cache->slabs_empty;
209     empty_count = 0;
210
211     while(slab != NULL) {
212         next = slab->next;
213         destroy_slab(cache, slab);
214         slab = next;
215         ++empty_count;
216     }
217
218     assert(cache->empty_count == empty_count);
219
220     /* free cache structure */
221     slab_cache_free(cache);
222 }

```

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`, `printf()`, `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
391     if(cache->flags & SLAB_RED_ZONE) {
392         rz_word = (uint32_t *) ((char *)buffer + cache->obj_size);
393
394         if(*rz_word != SLAB_RED_ZONE_VALUE) {
395             printk("detected write past the end of object, cache: %s buffer: 0x%x value: 0x%x\n",
396                  cache->name,
397                  (unsigned int)buffer,
398                  *rz_word);
399         }
400     }
401
402     *rz_word = SLAB_RED_ZONE_VALUE;
403 }
404
405 if(cache->flags & SLAB_POISON) {
406     if(cache->dtor != NULL) {
407         cache->dtor(buffer, cache->obj_size);
408     }
409
410     buffer32 = (uint32_t *)buffer;
411
412     for(idx = 0; idx < cache->obj_size / sizeof(uint32_t); ++idx) {
413         buffer32[idx] = SLAB_POISON_DEAD_VALUE;
414     }
415 }
416
417 /* link buffer into slab free list */
418 bufctl->next = slab->free_list;
419 slab->free_list = bufctl;
420 slab->obj_count -= 1;
421
422 /* If we just returned the last object to the slab, move the slab to
423  * the empty list. */
424 if(slab->obj_count == 0) {
425     /* remove from partial slabs list */
426     if(cache->slabs_partial == slab) {
427         cache->slabs_partial = slab->next;
428     }
429     else {
430         slab->prev->next = slab->next;
431     }
432
433     if(slab->next != NULL) {
434         slab->next->prev = slab->prev;
435     }
436
437     /* add to empty slabs list */
438     slab->next = cache->slabs_empty;
439     cache->slabs_empty = slab;
440
441     if(slab->next != NULL) {
442         slab->next->prev = slab;
443     }
444 }

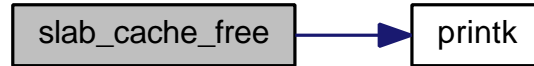
```

```

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
461     assert(slab_addr != NULL);
462
463     vm_map_kernel(slab_addr, pfalloc(), VM_FLAG_READ_WRITE);
464
465     slab = (slab_t *) ( (char *)slab_addr + SLAB_SIZE - sizeof(slab_t) );
466
467     slab->cache = cache;
468
469     /* slab is initially empty */
470     slab->obj_count = 0;
471
472     slab->next      = cache->slabs_empty;
473     cache->slabs_empty = slab;
474
475     if(slab->next != NULL) {
476         slab->next->prev = slab;
477     }
478
479     ++(cache->empty_count);
480
481     /* set slab colour and update cache next colour */
482     slab->colour = cache->next_colour;
483
484     if(cache->next_colour < cache->max_colour) {
485         cache->next_colour += cache->alignment;
486     }
487     else {
488         cache->next_colour = 0;
489     }
490 }

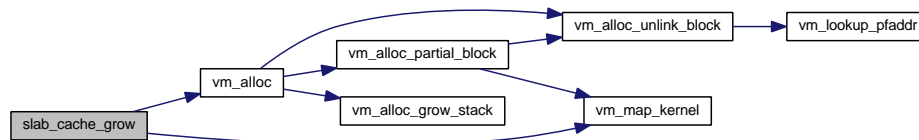
```

```

491
492     /* compute address of first bufctl */
493     bufctl = (slab_bufctl_t *) ( (char *)slab_addr + slab->colour + cache->
bufctl_offset );
494     slab->free_list = bufctl;
495
496     while(1) {
497         buffer = (addr_t)bufctl - cache->bufctl_offset;
498
499         if(cache->flags & SLAB_POISON) {
500             buffer_end = (uint32_t *) (buffer + cache->obj_size);
501
502             for(ptr = (uint32_t *)buffer; ptr < buffer_end; ++ptr) {
503                 *ptr = SLAB_POISON_DEAD_VALUE;
504             }
505
506             /* If both SLAB_POISON and SLAB_RED_ZONE are enabled, we
507              * perform redzone checking even on freed objects. */
508             if(cache->flags & SLAB_RED_ZONE) {
509                 *ptr = SLAB_RED_ZONE_VALUE;
510             }
511         }
512         else if (cache->ctor != NULL) {
513             cache->ctor((void *)buffer, cache->obj_size);
514         }
515
516         next = (slab_bufctl_t *) ( (char *)bufctl + cache->alloc_size );
517
518         if(next >= (slab_bufctl_t *)slab) {
519             bufctl->next = NULL;
520             break;
521         }
522     }
523
524     bufctl->next = next;
525     bufctl = next;
526 }
527 }

```

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

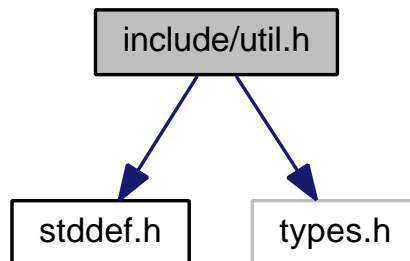
Definition at line 65 of file slab.c.

Referenced by slab_cache_create().

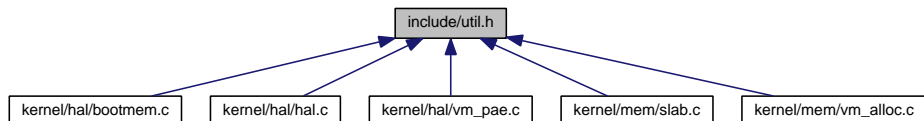
4.75 include/util.h File Reference

```

#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))`

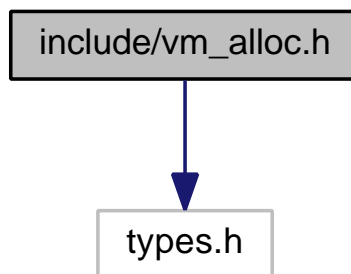
Definition at line 39 of file util.h.

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

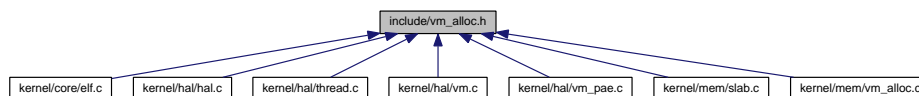
4.76 include/vm_alloc.h File Reference

```
#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

<i>allocator</i>	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()`.

```

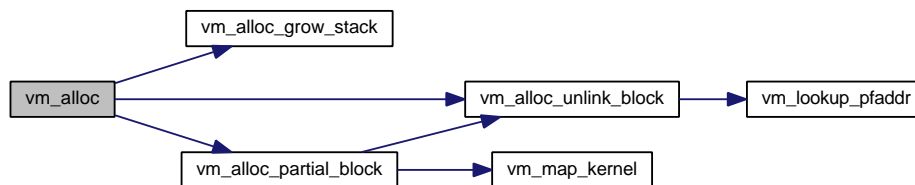
87                                     {
88     vm_block_t *block;
```

```

89     addr_t      page;
90
91     assert(allocator != NULL);
92
93     block = allocator->partial_list;
94
95
96     if(block == NULL) {
97         block = allocator->free_list;
98
99         if(block == NULL) {
100             return (addr_t)NULL;
101         }
102
103         vm_alloc_partial_block(block);
104     }
105
106     assert(block->stack_ptr != NULL);
107
108     /* if the page stack is empty, perform deferred page stack initialization */
109     if( VM_ALLOC_EMPTY_STACK(block) ) {
110         vm_alloc_grow_stack(block);
111     }
112
113     assert( ! VM_ALLOC_EMPTY_STACK(block) );
114
115     page = *(block->stack_ptr++);
116
117     /* if we just emptied the stack, mark the block as used */
118     if( VM_ALLOC_EMPTY_STACK(block) ) {
119         vm_alloc_unlink_block(block);
120     }
121
122     return page;
123 }

```

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

<i>allocator</i>	vm_alloc_t (p. 52) structure for a virtual memory allocator
<i>start_addr</i>	start address of the region
<i>end_addr</i>	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;

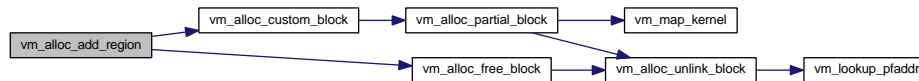
```

```

349     unsigned int idx;
350     addr_t      limit;
351
352     /* skip the block array */
353     if(start_addr >= allocator->start_addr) {
354         start_addr_adjusted = start_addr;
355     }
356     else {
357         start_addr_adjusted = allocator->start_addr;
358     }
359
360     /* start and end block indices */
361     start = ((unsigned int)start_addr_adjusted - (unsigned int)allocator->
base_addr) / VM_ALLOC_BLOCK_SIZE;
362     end   = ((unsigned int)end_addr - (unsigned int)allocator->
base_addr) / VM_ALLOC_BLOCK_SIZE;
363
364     /* check and remember whether last block is partial (last_full < end) or
365      * completely free (last_full == end) */
366     if( OFFSET_OF(end_addr, VM_ALLOC_BLOCK_SIZE) == 0 ) {
367         end_full = end;
368     }
369     else {
370         end_full = end + 1;
371     }
372
373     /* array initialization -- first block (if partial) */
374     idx = start;
375
376     if( OFFSET_OF(start_addr_adjusted, VM_ALLOC_BLOCK_SIZE) != 0 ) {
377         limit = ALIGN_END(start_addr_adjusted, VM_ALLOC_BLOCK_SIZE);
378
379         if(end_addr < limit) {
380             limit = end_addr;
381         }
382
383         vm_alloc_custom_block(&allocator->block_array[idx], start_addr_adjusted, limit);
384
385         ++idx;
386     }
387
388     /* array initialization -- free blocks */
389     for(; idx < end; ++idx) {
390         vm_alloc_free_block(&allocator->block_array[idx]);
391     }
392
393     /* array initialization -- last block (if partial) */
394     if(idx < end_full) {
395         vm_alloc_custom_block(&allocator->block_array[idx], allocator->
block_array[idx].base_addr, end_addr);
396     }
397 }

```

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_PARTIAL`, `VM_ALLOC_IS_USED`, and `vm_alloc_partial_block()`.

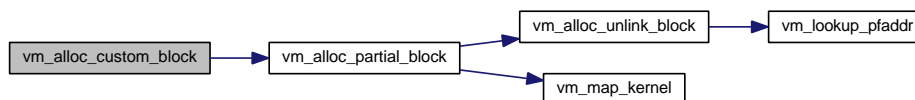
Referenced by `vm_alloc_add_region()`.

```

561
562 #ifndef NDEBBUG
563     addr_t    limit;
564 #endif
565     addr_t    page;
566     addr_t    adjusted_start;
567
568     assert(block != NULL);
569
570     assert(page_offset_of(start_addr) == 0 && page_offset_of(end_addr) == 0);
571
572 #ifndef NDEBBUG
573     limit = block->base_addr + VM_ALLOC_BLOCK_SIZE;
574 #endif
575
576     assert(start_addr >= block->base_addr && end_addr <= limit && start_addr < end_addr );
577
578     assert( ! VM_ALLOC_IS_FREE(block) );
579
580     adjusted_start = start_addr;
581
582     if( VM_ALLOC_IS_USED(block) ) {
583         /* if no stack address is specified at this point, use the first page
584          * of the address range for this purpose */
585         if( block->stack_addr == NULL ) {
586             block->stack_addr = (addr_t *)start_addr;
587             adjusted_start    = start_addr + PAGE_SIZE;
588
589             /* if the address range contained only a single page, there is
590              * nothing left to do here */
591             if(adjusted_start >= end_addr) {
592                 return;
593             }
594         }
595     }
596
597     vm_alloc_partial_block(block);
598
599     assert( VM_ALLOC_IS_PARTIAL(block) );
600
601     /* initialize stack */
602     page = adjusted_start;
603     while(page < end_addr) {
604         assert( ! VM_ALLOC_FULL_STACK(block) );
605
606         *(--block->stack_ptr) = page;
607         page += PAGE_SIZE;
608     }
609 }

```

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

<i>block</i>	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
411     assert(block != NULL);
412
413     /* unlink from partial list if necessary */
414     vm_alloc_unlink_block(block);

```

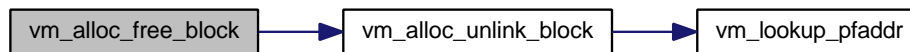


```

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:



4.76.3.6 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

<i>block</i>	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
737     assert(block != NULL);
738
739     assert(block->next != NULL && block->prev != NULL);
740
741     assert(block->stack_ptr != NULL);
742
743 }

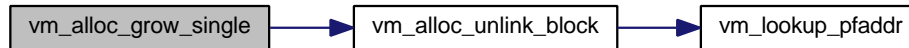
```

```

747     assert( ! VM_ALLOC_CANNOT_GROW(block) );
748
749     page          = block->stack_next;
750     block->stack_next = page + PAGE_SIZE;
751
752     if( VM_ALLOC_CANNOT_GROW(block) ) {
753         /* block is now used up, remove it from the partial list */
754         vm_alloc_unlink_block(block);
755     }
756
757     return page;
758 }

```

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

<i>block</i>	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
700     assert(block != NULL);
701
702     assert(block->next != NULL && block->prev != NULL);
703
704     assert(block->stack_ptr != NULL);
705
706     stack_ptr = block->stack_ptr;
707     page      = block->stack_next;
708     limit     = block->base_addr + VM_ALLOC_BLOCK_SIZE;
709
710     while(page < limit) {
711         assert( ! VM_ALLOC_FULL_STACK(block) );
712
713         *(--stack_ptr) = page;
714
715         page += PAGE_SIZE;
716     }
717
718     block->stack_ptr = stack_ptr;
719     block->stack_next = limit;
720 }
721
722
723
724 }

```

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

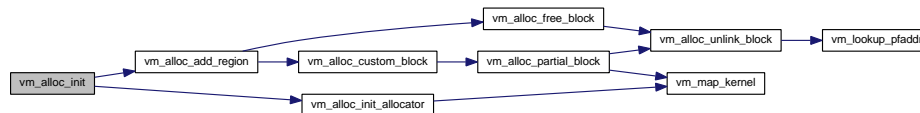
Definition at line 213 of file vm_alloc.c.

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

<i>allocator</i>	vm_alloc_t (p. 52) structure for a virtual memory allocator
<i>start_addr</i>	start address of the region managed by the allocator
<i>size</i>	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
269     assert(allocator != NULL);
270
271     assert( page_offset_of(start_addr) == 0 && page_offset_of(end_addr) == 0 );
272
273     /* align base and end addresses to block size */
```

```

277     base_addr  = (addr_t)ALIGN_START(start_addr, VM_ALLOC_BLOCK_SIZE);
278     aligned_end = (addr_t)ALIGN_END(end_addr, VM_ALLOC_BLOCK_SIZE);
279
280     /* calculate number of memory blocks managed by this allocator */
281     block_count = ( (char *)aligned_end - (char *)base_addr ) /
VM_ALLOC_BLOCK_SIZE;
282
283     /* calculate the number of pages required to store the memory block
284      * descriptor array */
285     array_size = block_count * sizeof(vm_block_t);
286     array_page_count = array_size / PAGE_SIZE;
287     if(array_size % PAGE_SIZE != 0) {
288         ++array_page_count;
289     }
290
291     /* address of the block array */
292     block_array = (vm_block_t *)start_addr;
293
294     /* adjust base address to skip block descriptor array */
295     adjusted_start = start_addr + array_page_count * PAGE_SIZE;
296
297     /* initialize allocator struct */
298     allocator->start_addr  = adjusted_start;
299     allocator->end_addr    = end_addr;
300     allocator->base_addr   = base_addr;
301     allocator->block_count = block_count;
302     allocator->block_array = block_array;
303     allocator->array_pages = array_page_count;
304     allocator->free_list   = NULL;
305     allocator->partial_list = NULL;
306
307     /* allocate block descriptor array pages */
308     addr = (addr_t)block_array;
309     for(idx = 0; idx < array_page_count; ++idx) {
310         /* allocate and map page */
311         paddr = pfalloc();
312         vm_map_kernel(addr, paddr, VM_FLAG_READ_WRITE);
313
314         /* calculate address of next page */
315         addr += PAGE_SIZE;
316     }
317
319     assert(addr == adjusted_start);
320
321     /* basic initialization of array (all blocks unlinked/used) */
322     addr = base_addr;
323     for(idx = 0; idx < block_count; ++idx) {
324         block_array[idx].base_addr = addr;
325         block_array[idx].allocator = allocator;
326
327         /* mark block as unlinked for now */
328         block_array[idx].next = NULL;
329
330         /* a null stack base indicates the block is uninitialized */
331         block_array[idx].stack_addr = NULL;
332
333         /* calculate address of next block */
334         addr += VM_ALLOC_BLOCK_SIZE;
335     }
336 }

```

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

<i>allocator</i>	allocator which manages the memory region from which we wish to obtain a page
------------------	---

ASSERTION: allocator is not null

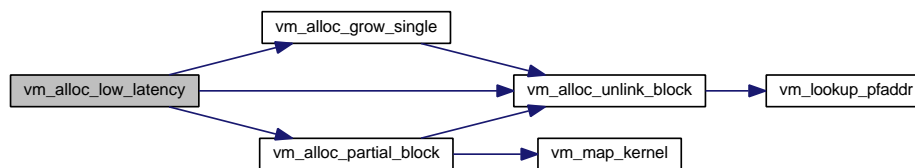
Definition at line 133 of file vm_alloc.c.

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
137     assert(allocator != NULL);
138
139     block = allocator->partial_list;
140
141     if(block == NULL) {
142         block = allocator->free_list;
143
144         if(block == NULL) {
145             return (addr_t) NULL;
146         }
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

<i>block</i>	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     paddr_t         paddr;
459     bool            was_free;
460
461
462     assert(block != NULL);
463
464     /* To keep in mind...
465     *
466     * When the allocator is initialized, some blocks may be created partial
467     * (typical for the first and the last block of the region). If there is a
468     * hole at the start of the block, the page stack will be at the first
469     * available page, not at the start of the block. Since these blocks have
470     * holes, they will never be in the free state.
471     *
472     * So, when a block is free on function entry, we ensure the stack is placed
473     * at the start of the block so that all the remaining pages can be
474     * allocated sequentially (see deferred stack initialization below). However,
475     * if the block is in the used state on function entry, we leave the stack
476     * at its previous location since the first page of the block might not be
477     * available. */
478
479     if(block->next == NULL) {
480         assert(block->stack_addr != NULL);
481
482         /* block was used on function entry */
483         was_free = false;
484     }
485     else {
486         if(block->stack_ptr != NULL) {
487             /* block is already partial, leave it untouched */
488             return;
489         }
490
491         /* block was free on function entry */
492         was_free = true;
493
494         /* unlink from free list */
495         vm_alloc_unlink_block(block);
496
497         /* use first page of block for the stack */
498         block->stack_addr = (addr_t *)block->base_addr;
499     }
500
501     /* allocate the page stack */
502     stack_addr = block->stack_addr;
503     paddr      = pfalloc();
504     vm_map_kernel((addr_t)stack_addr, paddr, VM_FLAG_READ_WRITE);
505
506     assert(block->allocator != NULL);
507
508     /* link block to the partial list */
509     if(block->allocator->partial_list == NULL) {
510         /* special case: partial list is empty */
511         block->allocator->partial_list = block;
512
513         block->next = block;

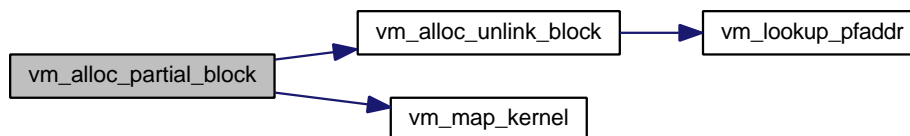
```

```

517     block->prev = block;
518 }
519 else {
520     /* block will be at to the end of the partial block list */
521     next = block->allocator->partial_list;
522     prev = next->prev;
523
524     /* re-link block */
525     block->prev = prev;
526     block->next = next;
527
528     prev->next = block;
529     next->prev = block;
530 }
531
532 /* Ok, here's the deal (deferred stack intialization)...
533  *
534  * We do not want to initialize the page stack right now because this is
535  * a time consuming operation, and we might be in time-critical code
536  * (interrupt handling code for example). Instead, the stack initialization
537  * is deferred until the next page allocations. The first non-time critical
538  * allocation which encounters an empty stack will initialize the whole
539  * stack. In the meantime, time critical ones will just allocate pages
540  * sequentially from the start of the block.
541  *
542  * The stack_next pointer in the vm_block_t structure points to the next
543  * page available for sequential allocation. The memory block is actually
544  * used up (no more pages available) when the page stack is empty AND the
545  * stack_next pointer has reached the end of the block. */
546
547 /* initialize the stack as empty */
548 block->stack_ptr = (addr_t *) ( (char *)stack_addr + PAGE_SIZE );
549
550 if(was_free) {
551     /* free block: we skip the first page as it was allocated for the
552      * stack itself */
553     block->stack_next = block->base_addr + PAGE_SIZE;
554 }
555 else {
556     /* used block: sequential allocation no longer possible */
557     block->stack_next = block->base_addr + VM_ALLOC_BLOCK_SIZE;
558 }
559 }

```

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

<i>block</i>	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

<i>allocator</i>	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()`.

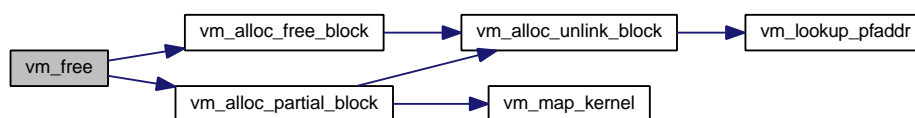
Referenced by `elf_load()`, `elf_setup_stack()`, `thread_page_create()`, and `thread_page_destroy()`.

```

172                                     {
173     vm_block_t    *block;
174     unsigned int  idx;
175
176     assert(allocator != NULL);
177
178     assert(page >= allocator->start_addr && page < allocator->end_addr);
179
180     assert(page_offset_of(page) == 0);
181
182     /* find the block to which the free page belong */
183     idx = ( (unsigned int)page - (unsigned int)allocator->base_addr ) /
184     VM_ALLOC_BLOCK_SIZE;
185     block = &allocator->block_array[idx];
186
187     /* if the block was a used block, make it a partial block */
188     if( VM_ALLOC_IS_USED(block) ) {
189         if(block->stack_addr == NULL) {
190             block->stack_addr = (addr_t *)page;
191             return;
192         }
193     }
194
195     vm_alloc_partial_block(block);
196
197     assert( VM_ALLOC_IS_PARTIAL(block) );
198
199     assert( ! VM_ALLOC_FULL_STACK(block) );
200
201     *(--block->stack_ptr) = page;
202
203     /* check if we just freed the whole block */
204     if( VM_ALLOC_FULL_STACK(block) ) {
205         vm_alloc_free_block(block);
206     }
207 }
208
209
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)

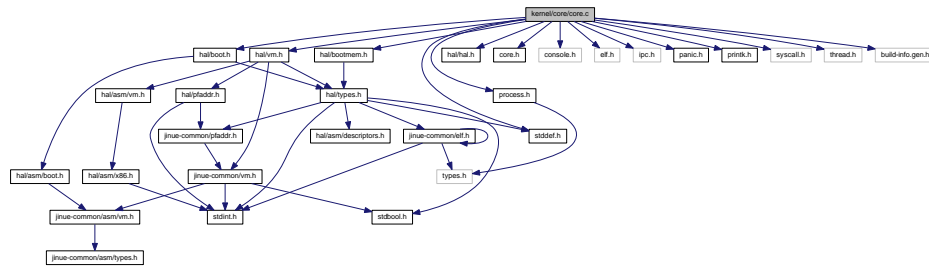
Definition at line 58 of file vm.c.

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 <printf.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

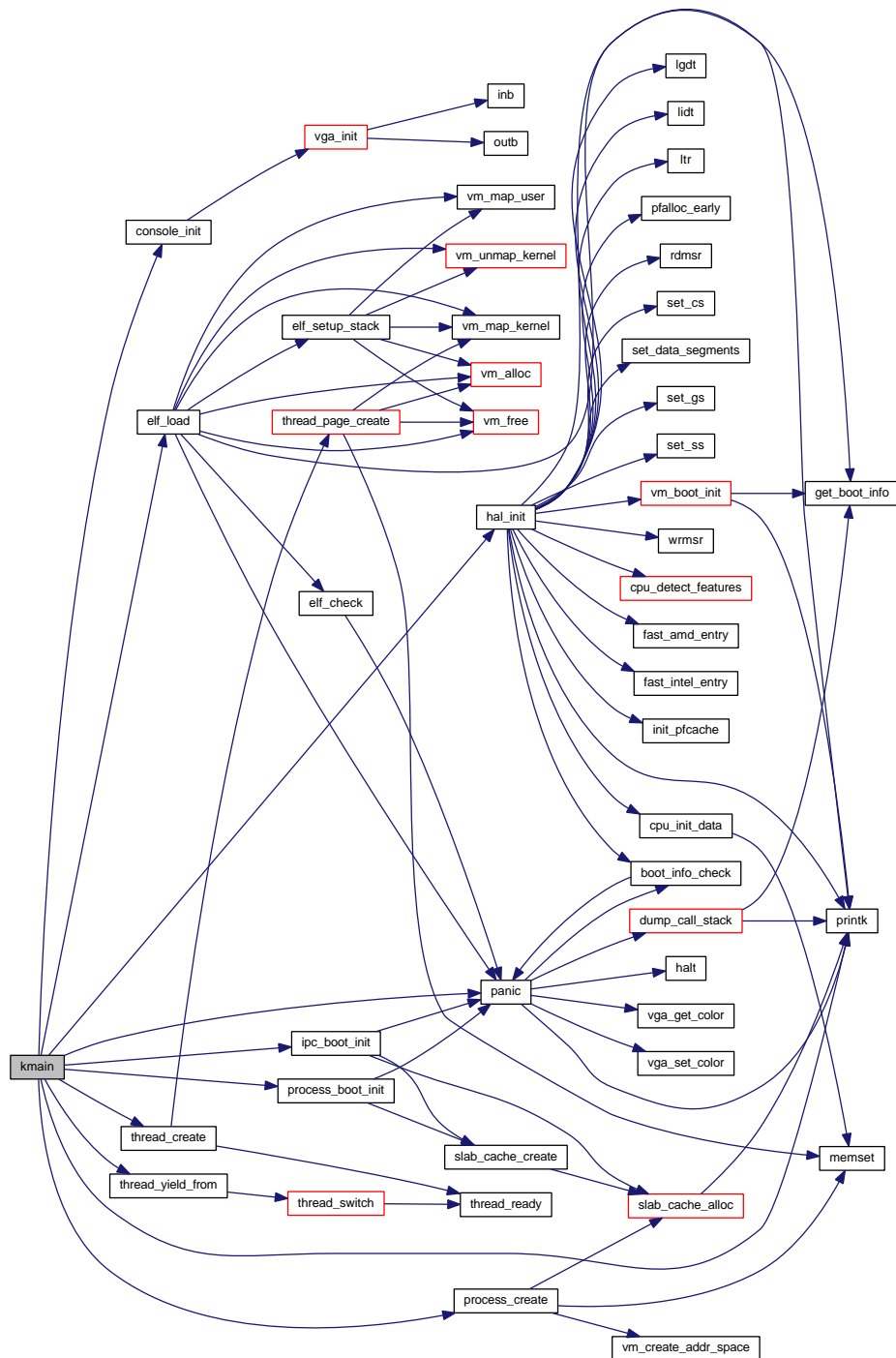
4.77.1.1 void kmain (void)

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(), printf(), 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();
86     elf_load(&elf_info, elf, &process->addr_space);
87
88     /* create initial thread */
89     thread_t *thread = thread_create(
90         process,
91         elf_info.entry,
92         elf_info.stack_addr);
93
94     if(thread == NULL) {
95         panic("Could not create initial thread.");
96     }
97
98     /* start process manager
99     *
100     * We switch from NULL since this is the first thread. */
101     thread_yield_from(
102         NULL,
103         false,      /* don't block */
104         false);     /* don't destroy */
105                 /* just be nice */
106
107     /* should never happen */
108     panic("thread_yield_from() returned in kmain()");
109 }
```

Here is the call graph for this function:

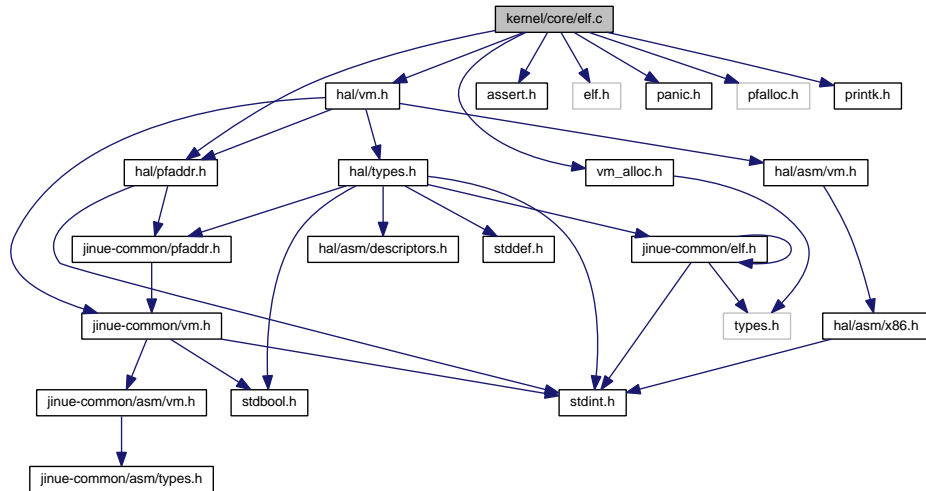


4.78 kernel/core/elf.c File Reference

```
#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.

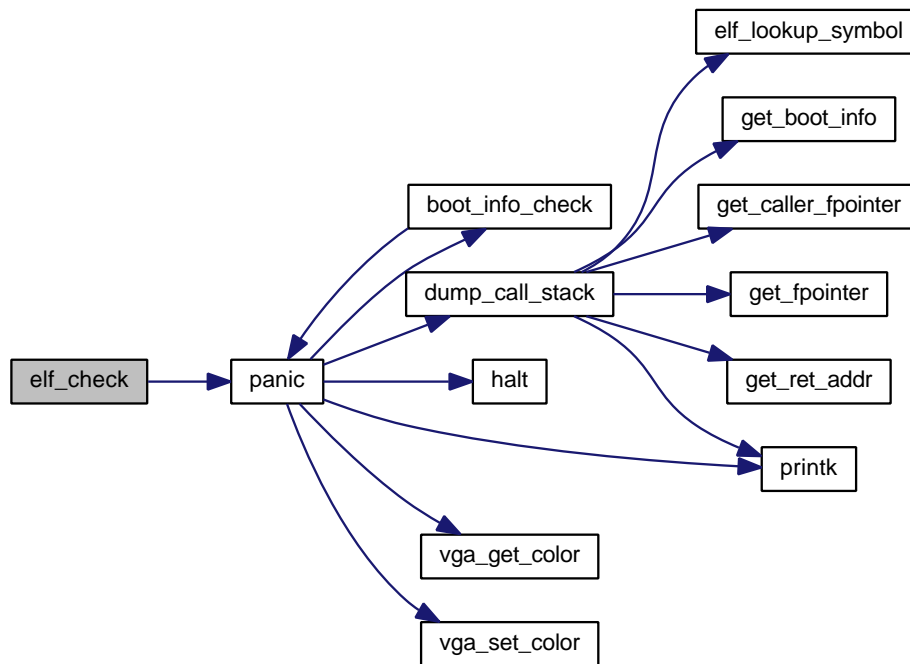
References Elf32_Ehdr::e_entry, Elf32_Ehdr::e_flags, Elf32_Ehdr::e_ident, Elf32_Ehdr::e_machine, Elf32_Ehdr::e_phentsize, Elf32_Ehdr::e_phnum, Elf32_Ehdr::e_phoff, Elf32_Ehdr::e_type, Elf32_Ehdr::e_version, EI_CLASS, EI_DATA, EI_MAG0, EI_MAG1, EI_MAG2, EI_MAG3, EI_VERSION, ELF_MAGIC0, ELF_MAGIC1, ELF_MAGIC2, 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 */
```

```
52     if(elf->e_ident[EI_CLASS] != ELFCLASS32) {
53         panic("Bad file class");
54     }
55
56     /* check: endianness */
57     if(elf->e_ident[EI_DATA] != ELFDATA2LSB) {
58         panic("Bad endianness");
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 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:



4.78.1.2 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;

```

```

117     info->at_phent      = elf->e_phentsize;
118     info->addr_space    = addr_space;
119     info->entry         = (addr_t)elf->e_entry;
120
121     /* temporary page for copies */
122     dest_page = (char *)vm_alloc(global_page_allocator);
123
124     for(idx = 0; idx < elf->e_phnum; ++idx) {
125         if(phdr[idx].p_type != PT_LOAD) {
126             continue;
127         }
128
129         /* check that the segment is not in the region reserved for kernel use */
130         if(! user_buffer_check((void *)phdr[idx].p_vaddr, phdr[idx].p_memsz)) {
131             panic("process manager memory layout -- address of segment too low");
132         }
133
134         /* set start and end addresses for mapping and copying */
135         file_ptr = (char *)elf + phdr[idx].p_offset;
136         vptr     = (char *)phdr[idx].p_vaddr;
137         vend     = vptr + phdr[idx].p_memsz; /* limit for padding */
138         v fend   = vptr + phdr[idx].p_filesz; /* limit for copy */
139
140         /* align on page boundaries, be inclusive,
141          note that v fend is not aligned */
142         file_ptr = (char *) ( (uintptr_t)file_ptr & ~PAGE_MASK );
143         vptr     = (char *) ( (uintptr_t)vptr & ~PAGE_MASK );
144
145         if(page_offset_of(vend) != 0) {
146             vend = (char *) ( (uintptr_t)vend & ~PAGE_MASK );
147             vend += PAGE_SIZE;
148         }
149
150         /* copy if we have to */
151         if( (phdr[idx].p_flags & PF_W) || (phdr[idx].p_filesz != phdr[idx].
p_memsz) ) {
152             while(vptr < vend) {
153                 /* start of this page and next page */
154                 vpage = (addr_t)vptr;
155                 vnext = vptr + PAGE_SIZE;
156
157                 /* allocate and map the new page */
158                 page = pfalloc();
159                 vm_map_kernel((addr_t)dest_page, page, VM_FLAG_READ_WRITE);
160
161                 dest = dest_page;
162
163                 /* copy */
164                 stop = vnext;
165                 if(stop > v fend) {
166                     stop = v fend;
167                 }
168
169                 while(vptr < stop) {
170                     *(dest++) = *(file_ptr++);
171                     ++vptr;
172                 }
173
174                 /* pad */
175                 while(vptr < vnext) {
176                     *(dest++) = 0;
177                     ++vptr;
178                 }
179
180                 /* set flags */
181                 if(phdr[idx].p_flags & PF_W) {
182                     flags = VM_FLAG_READ_WRITE;
183                 }
184                 else {
185                     flags = VM_FLAG_READ_ONLY;
186                 }
187
188                 /* undo temporary mapping and map page in proper address
189                  * space */
190                 vm_unmap_kernel((addr_t)dest_page);
191                 vm_map_user(addr_space, (addr_t)vpage, page, flags);
192             }
193         }
194     }
195     else {
196         while(vptr < vend) {
197             /* perform mapping */

```


Here is the call graph for this function:



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`.

```

288                                     {
289
290     int      idx;
291     size_t   symbol_entry_size;
292     size_t   symbol_table_size;
293
294     const char *elf_file      = elf_file_bytes(elf_header);
295     const char *symbols_table = NULL;

```

```

296     const char *string_table    = NULL;
297
298     for(idx = 0; idx < elf_header->e_shnum; ++idx) {
299         const Elf32_Shdr *section_header = elf_get_section_header(elf_header, idx);
300
301         if(section_header->sh_type == SHT_SYMTAB) {
302             symbols_table        = &elf_file[section_header->sh_offset];
303             symbol_entry_size     = section_header->sh_entsize;
304             symbol_table_size     = section_header->sh_size;
305
306             const Elf32_Shdr *string_section_header = elf_get_section_header(
307                 elf_header,
308                 section_header->sh_link);
309
310             string_table = &elf_file[string_section_header->sh_offset];
311
312             break;
313         }
314     }
315
316     if(symbols_table == NULL) {
317         /* no symbol table */
318         return -1;
319     }
320
321     const char *symbol = symbols_table;
322
323     while(symbol < symbols_table + symbol_table_size) {
324         const Elf32_Sym *symbol_header = (const Elf32_Sym *)symbol;
325
326         if(ELF32_ST_TYPE(symbol_header->st_info) == type) {
327             Elf32_Addr lookup_addr = (Elf32_Addr)addr;
328             Elf32_Addr start       = symbol_header->st_value;
329             Elf32_Addr end         = start + symbol_header->st_size;
330
331             if(lookup_addr >= start && lookup_addr < end) {
332                 result->addr = symbol_header->st_value;
333                 result->name = &string_table[symbol_header->st_name];
334
335                 return 0;
336             }
337         }
338
339         symbol += symbol_entry_size;
340     }
341
342     /* not found */
343     return -1;
344 }

```

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
218     /* initial stack allocation */
219     for(vpage = (addr_t)STACK_START; vpage < (addr_t)STACK_BASE; vpage +=
220     PAGE_SIZE) {
221         page = pfalloc();
222         vm_map_user(info->addr_space, vpage, page, VM_FLAG_READ_WRITE);
223     }

```

```
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
228      * it temporarily in this address space so we can write to it. */
229     addr_t top_page = vm_alloc(global_page_allocator);
230     vm_map_kernel(top_page, page, VM_FLAG_READ_WRITE);
231
232     /* start at the top */
233     uint32_t *sp = (uint32_t *) (top_page + PAGE_SIZE);
234
235     /* Program name string: "proc", null-terminated */
236     * (--sp) = 0;
237     * (--sp) = 0x636f7270;
238
239     char *argv0 = (char *) STACK_BASE - 2 * sizeof(uint32_t);
240
241     /* auxiliary vectors */
242     Elf32_auxv_t *auxvp = (Elf32_auxv_t *) sp - 7;
243
244     auxvp[0].a_type = AT_PHDR;
245     auxvp[0].a_un.a_val = (int32_t) info->at_phdr;
246
247     auxvp[1].a_type = AT_PHENT;
248     auxvp[1].a_un.a_val = (int32_t) info->at_phent;
249
250     auxvp[2].a_type = AT_PHNUM;
251     auxvp[2].a_un.a_val = (int32_t) info->at_phnum;
252
253     auxvp[3].a_type = AT_PAGESZ;
254     auxvp[3].a_un.a_val = PAGE_SIZE;
255
256     auxvp[4].a_type = AT_ENTRY;
257     auxvp[4].a_un.a_val = (int32_t) info->entry;
258
259     auxvp[5].a_type = AT_STACKBASE;
260     auxvp[5].a_un.a_val = STACK_BASE;
261
262     auxvp[6].a_type = AT_NULL;
263     auxvp[6].a_un.a_val = 0;
264
265     sp = (uint32_t *) auxvp;
266
267     /* empty environment variables */
268     * (--sp) = 0;
269
270     /* argv with only program name */
271     * (--sp) = 0;
272     * (--sp) = (uint32_t) argv0;
273
274     /* argc */
275     * (--sp) = 1;
276
277     info->stack_addr = (addr_t) STACK_BASE - PAGE_SIZE + ((addr_t) sp - top_page);
278
279     /* unmap and free temporary page */
280     vm_unmap_kernel(top_page);
281     vm_free(global_page_allocator, top_page);
282 }
```


4.79.1 Function Documentation

4.79.1.1 void ipc_boot_init (void)

Definition at line 58 of file ipc.c.

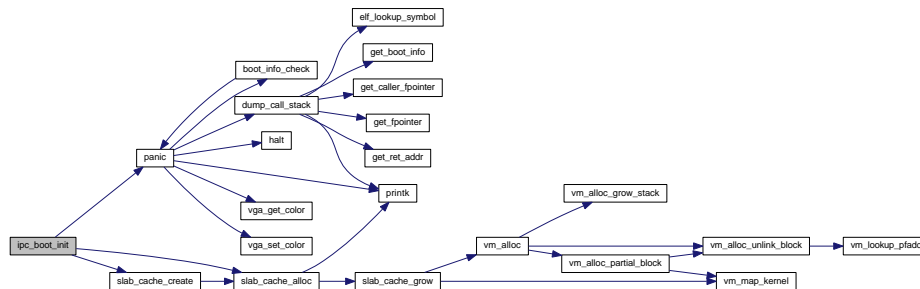
References `NULL`, `panic()`, `slab_cache_alloc()`, `slab_cache_create()`, and `SLAB_DEFAULTS`.

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()`.

```

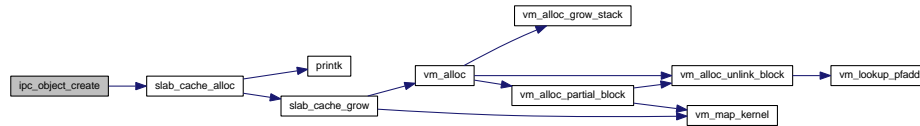
74      {
75      ipc_t *ipc = slab_cache_alloc(ipc_object_cache);
```

```

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

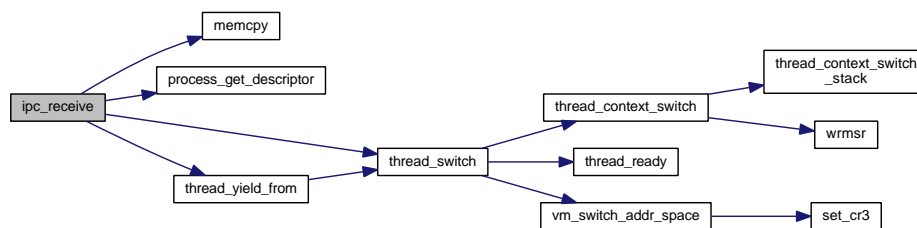
```

```

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->rcv_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_addrref(&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:



4.79.1.5 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) {
307         syscall_args_set_error(args, JINUE_EINVAL);
308         return;
309     }
310
311     size_t buffer_size              = jinue_args_get_buffer_size(args);
312     size_t data_size                = jinue_args_get_data_size(args);
313     size_t desc_n                  = jinue_args_get_n_desc(args);
314     size_t total_size               =
315         data_size +
316         desc_n * sizeof(jinue_ipc_descriptor_t);
317
318     if(buffer_size > JINUE_SEND_MAX_SIZE) {
319         syscall_args_set_error(args, JINUE_EINVAL);
320         return;
321     }
322
323     if(total_size > buffer_size) {
324         syscall_args_set_error(args, JINUE_EINVAL);
325         return;
326     }
327
328     if(desc_n > JINUE_SEND_MAX_N_DESC) {
329         syscall_args_set_error(args, JINUE_EINVAL);
330         return;
331     }
332
333     /* the reply must fit in the sender's buffer */
334     if(total_size > send_thread->message_info.buffer_size) {
335         syscall_args_set_error(args, JINUE_EINVAL);
336         return;
337     }
338
339     if(desc_n > 0) {
340         syscall_args_set_error(args, JINUE_ENOSYS);
341         return;
342     }
343
344     const char *user_ptr = (const char *)args->arg2;
345
346     if(! user_buffer_check(user_ptr, buffer_size)) {
347         syscall_args_set_error(args, JINUE_EINVAL);
348         return;
349     }
350
351     memcpy(&send_thread->message_buffer, user_ptr, data_size);
352
353     syscall_args_set_return(send_thread->message_args, 0);
354     send_thread->message_args->arg3 =
355         args->arg3 & ~(JINUE_SEND_SIZE_MASK << JINUE_SEND_BUFFER_SIZE_OFFSET);
356
357     send_thread->message_info.data_size = data_size;
358     send_thread->message_info.desc_n    = desc_n;
359
360

```

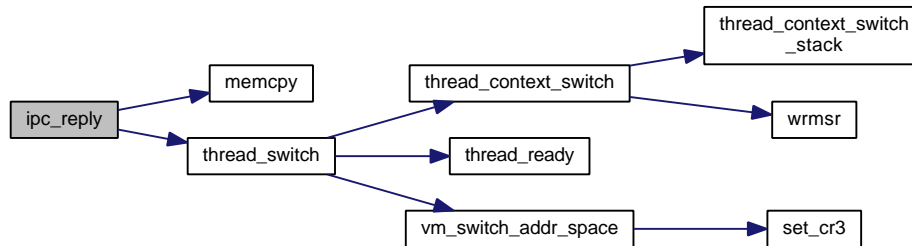


```

365     object_subref(&send_thread->header);
366     thread->sender = NULL;
367
368     syscall_args_set_return(args, 0);
369
370     /* switch back to sender thread to return from call immediately */
371     thread_switch(
372         thread,
373         send_thread,
374         false, /* don't block (put this thread back in ready queue) */
375         false); /* don't destroy */
376 }

```

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::rcv_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
116 if(message_info->desc_n > 0) {
117     syscall_args_set_error(args, JINUE_ENOSYS);
118     return;
119 }
120
121 int fd = (int)args->arg1;
122
123 object_ref_t *ref = process_get_descriptor(thread->process, fd);
124
125 if(! object_ref_is_valid(ref)) {
126     syscall_args_set_error(args, JINUE_EBADF);
127     return;
128 }
129
130 if(object_ref_is_closed(ref)) {
131     syscall_args_set_error(args, JINUE_EIO);
132     return;
133 }
134
135 message_info->cookie = ref->cookie;
136
137 object_header_t *header = ref->object;
138
139 if(object_is_destroyed(header)) {
140     ref->flags |= OBJECT_REF_FLAG_CLOSED;
141     object_subref(header);
142
143     syscall_args_set_error(args, JINUE_EIO);
144     return;
145 }
146
147 if(header->type != OBJECT_TYPE_IPC) {
148     syscall_args_set_error(args, JINUE_EBADF);
149     return;
150 }
151
152 ipc_t *ipc = (ipc_t *)header;
153
154 char *user_ptr = (char *)args->arg2;
155
156 if(! user_buffer_check(user_ptr, message_info->buffer_size)) {
157     syscall_args_set_error(args, JINUE_EINVAL);
158     return;
159 }
160
161 memcpy(&thread->message_buffer, user_ptr, message_info->data_size);
162
163 /* return values are set by ipc_reply() (or by ipc_receive() if the call
164  * fails because the message is too big for the receiver's buffer) */
165 thread->message_args = args;
166
167 thread_t *recv_thread = jinue_node_entry(
168     jinue_list_dequeue(&ipc->recv_list),
169     thread_t,
170     thread_list);
171
172 if(recv_thread == NULL) {
173     /* No thread is waiting to receive this message, so we must wait on the
174      * sender list. */
175     jinue_list_enqueue(&ipc->send_list, &thread->thread_list);
176
177     thread_yield_from(
178         thread,
179         true, /* make thread block */
180         false); /* don't destroy */
181 }
182 else {
183     object_addr(&thread->header);
184     recv_thread->sender = thread;
185
186     /* switch to receiver thread, which will resume inside syscall_receive() */
187     thread_switch(
188         thread,

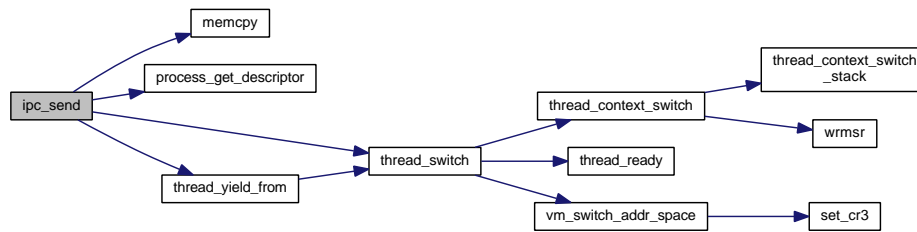
```

```

192         rcv_thread,
193         true,      /* block sender thread */
194         false);    /* don't destroy sender */
195     }
196
197     /* copy reply to user space buffer */
198     memcpy(user_ptr, &thread->message_buffer, message_info->data_size);
199
201 }

```

Here is the call graph for this function:



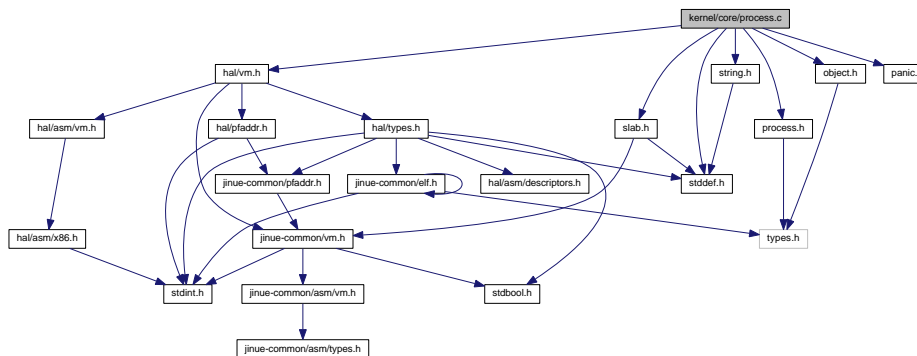
4.80 kernel/core/process.c File Reference

```

#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.

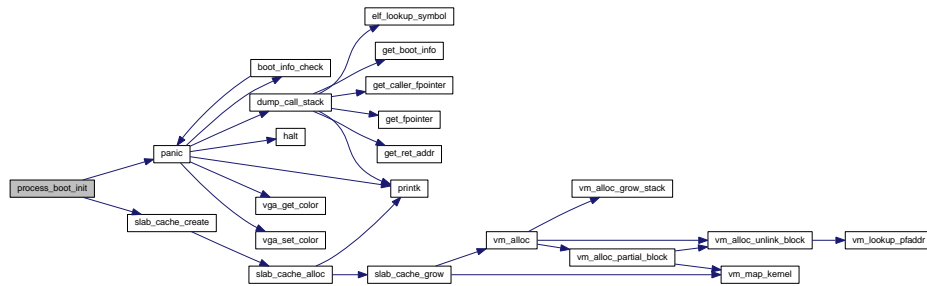
Referenced by kmain().

```

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().

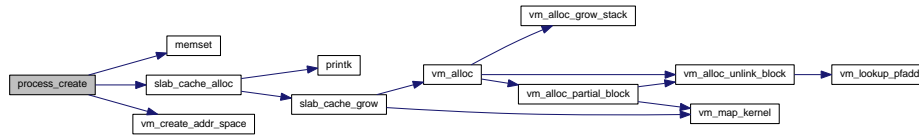
Referenced by kmain().

```

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

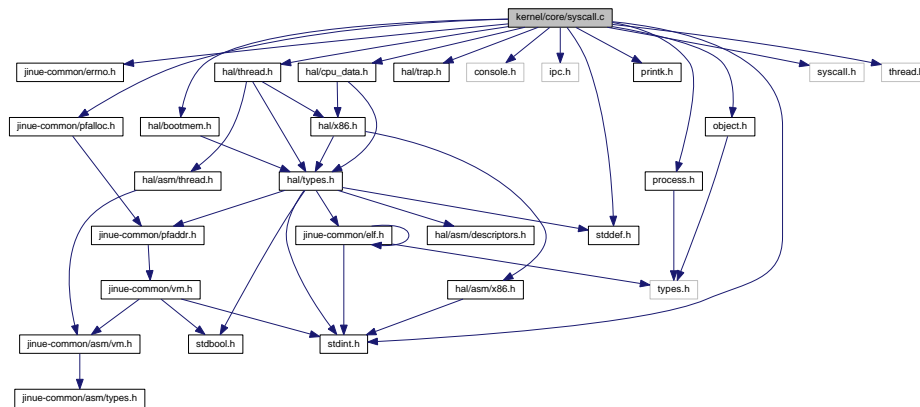
```
#include <jinue-common/errno.h>
```

```

#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_printrn(), 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
52     uintptr_t function_number = args->arg0;
53
54     if(function_number < SYSCALL_FUNCT_PROC_BASE) {
55         /* microkernel system calls */
56         switch(function_number) {
57
58             case SYSCALL_FUNCT_SYSCALL_METHOD:
59                 syscall_args_set_return(args, syscall_method);
60                 break;
61
62             case SYSCALL_FUNCT_CONSOLE_PUTC:
63                 console_putc((char)args->arg1);
64                 syscall_args_set_return(args, 0);
65                 break;
66
67             case SYSCALL_FUNCT_CONSOLE_PUTS:
68                 console_printn((char *)args->arg2, jinue_args_get_data_size(args));
69                 syscall_args_set_return(args, 0);
70                 break;
71
72             case SYSCALL_FUNCT_THREAD_CREATE:
73             {
74                 thread_t *thread = thread_create(
75                     /* TODO use arg1 as an address space reference if specified */
76                     get_current_thread()->process,
77                     (addr_t)args->arg2,
78                     (addr_t)args->arg3);
79
80                 if(thread == NULL) {
81                     syscall_args_set_error(args, JINUE_EAGAIN);
82                 }
83                 else {
84                     syscall_args_set_return(args, 0);
85                 }
86             }
87             break;
88
89             case SYSCALL_FUNCT_THREAD_YIELD:
90                 thread_yield_from(
91                     get_current_thread(),
92                     false, /* don't block */
93                     args->arg1); /* destroy (aka. exit) thread if true */
94                 syscall_args_set_return(args, 0);
95                 break;
96
97             case SYSCALL_FUNCT_SET_THREAD_LOCAL_ADDR:
98                 thread_context_set_local_storage(
99                     &get_current_thread()->thread_ctx,
100                     (addr_t)args->arg1,
101                     (size_t)args->arg2);
102                 syscall_args_set_return(args, 0);
103                 break;
104
105             case SYSCALL_FUNCT_GET_THREAD_LOCAL_ADDR:
106                 syscall_args_set_return_ptr(
107                     args,
108                     thread_context_get_local_storage(
109                         &get_current_thread()->thread_ctx));
110                 break;
111
112             case SYSCALL_FUNCT_GET_FREE_MEMORY:
113             {
114                 bootmem_t *block;
115                 memory_block_t *block_dest;
116                 unsigned int count, count_max;
117
118

```

```

122     size_t buffer_size = jinue_args_get_buffer_size(args);
123     block_dest = (memory_block_t *)jinue_args_get_buffer_ptr(args);
124
125     count_max = buffer_size / sizeof(memory_block_t);
126
127     for(count = 0; count < count_max; ++count) {
128         block = bootmem_get_block();
129
130         if(block == NULL) {
131             break;
132         }
133
134         block_dest->addr = block->addr;
135         block_dest->count = block->count;
136
137         ++block_dest;
138     }
139
140     args->arg0 = (uintptr_t)count;
141
142     if(count == count_max && bootmem_root != NULL) {
143         args->arg1 = JINUE_EMORE;
144     }
145     else {
146         args->arg1 = 0;
147     }
148
149     args->arg2 = 0;
150     args->arg3 = 0;
151 }
152 break;
153
154 case SYSCALL_FUNCT_CREATE_IPC:
155 {
156     ipc_t *ipc;
157
158     thread_t *thread = get_current_thread();
159
160     int fd = process_unused_descriptor(thread->process);
161
162     if(fd < 0) {
163         syscall_args_set_error(args, JINUE_EAGAIN);
164         break;
165     }
166
167     if(args->arg1 & JINUE_IPC_PROC) {
168         ipc = ipc_get_proc_object();
169     }
170     else {
171         int flags = IPC_FLAG_NONE;
172
173         if(args->arg1 & JINUE_IPC_SYSTEM) {
174             flags |= IPC_FLAG_SYSTEM;
175         }
176
177         ipc = ipc_object_create(flags);
178
179         if(ipc == NULL) {
180             syscall_args_set_error(args, JINUE_EAGAIN);
181             break;
182         }
183     }
184
185     object_ref_t *ref = process_get_descriptor(thread->process, fd);
186
187     object_addrref(&ipc->header);
188
189     ref->object = &ipc->header;
190     ref->flags = OBJECT_REF_FLAG_VALID | OBJECT_REF_FLAG_OWNER;
191     ref->cookie = 0;
192
193     syscall_args_set_return(args, fd);
194
195 }
196 break;
197 case SYSCALL_FUNCT_RECEIVE:
198     ipc_receive(args);
199     break;
200
201 case SYSCALL_FUNCT_REPLY:
202     ipc_reply(args);

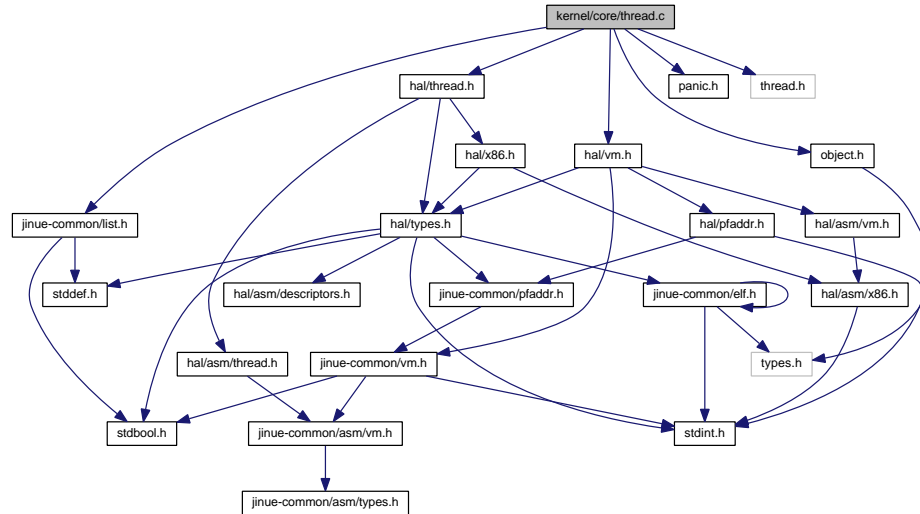
```


Here is the call graph for this function:



Generated on Fri Jan 4 2019 22:25:11 for Jinue by Doxygen

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().

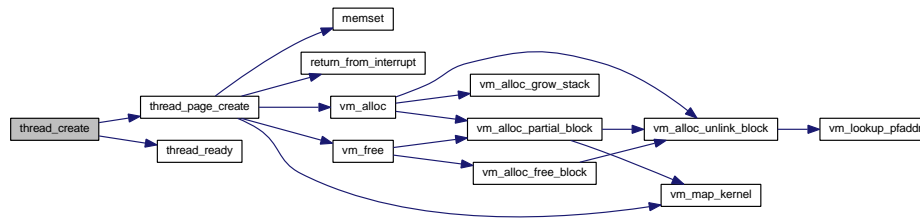
Referenced by dispatch_syscall(), and kmain().

```

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:



4.82.1.2 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().

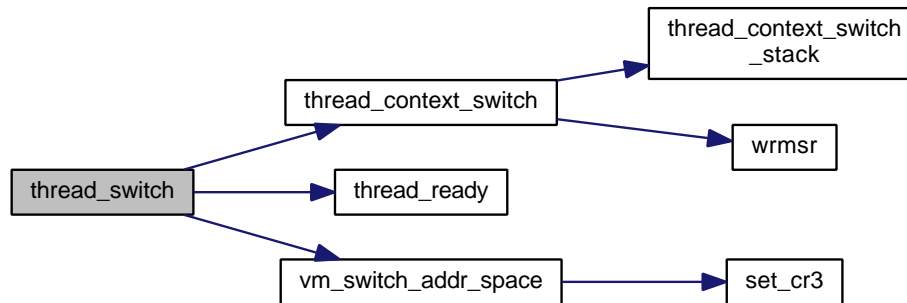
Referenced by ipc_receive(), ipc_reply(), ipc_send(), and thread_yield_from().

```

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.

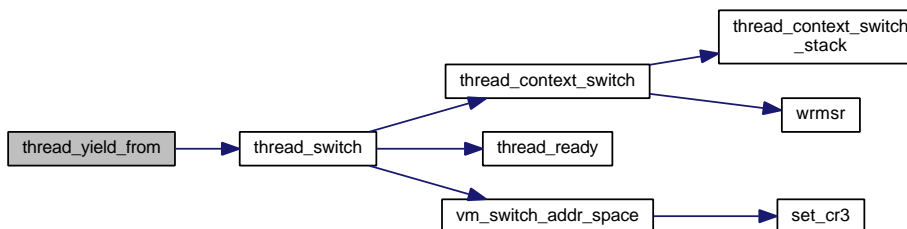
References thread_switch().

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 }
  
```

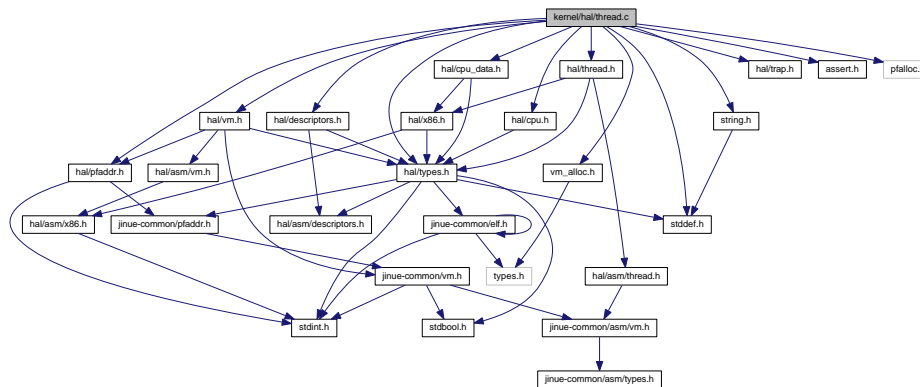
Here is the call graph for this function:



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 <pfallloc.h>
#include <stddef.h>
#include <string.h>
#include <vm_alloc.h>
```



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().

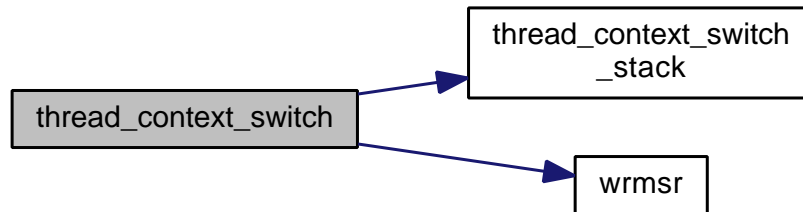
```
148
149
151     assert(to_ctx != NULL);
152
```

```

154     assert(from_ctx != NULL || ! destroy_from);
155
156     /* nothing to do if this is already the current thread */
157     if(from_ctx != to_ctx) {
158         /* setup TSS with kernel stack base for this thread context */
159         addr_t kernel_stack_base = get_kernel_stack_base(to_ctx);
160         tss_t *tss = get_tss();
161
162         tss->esp0 = kernel_stack_base;
163         tss->esp1 = kernel_stack_base;
164         tss->esp2 = kernel_stack_base;
165
166         /* update kernel stack address for SYSENTER instruction */
167         if(cpu_has_feature(CPU_FEATURE_SYSENTER)) {
168             wrmsr(MSR_IA32_SYSENTER_ESP, (uint64_t)(uintptr_t)kernel_stack_base);
169         }
170
171         /* switch thread context stack */
172         thread_context_switch_stack(from_ctx, to_ctx, destroy_from);
173     }
174 }

```

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().

```

87     {
88
89         /* allocate thread context */
90         thread_t *thread = (thread_t *)vm_alloc( global_page_allocator );
91
92         if(thread != NULL) {
93             pfaddr_t pf = pfalloc();
94
95             if(pf == PFNULL) {
96                 vm_free(global_page_allocator, (addr_t)thread);
97                 return NULL;
98             }
99
100             vm_map_kernel((addr_t)thread, pf, VM_FLAG_READ_WRITE);
101         }

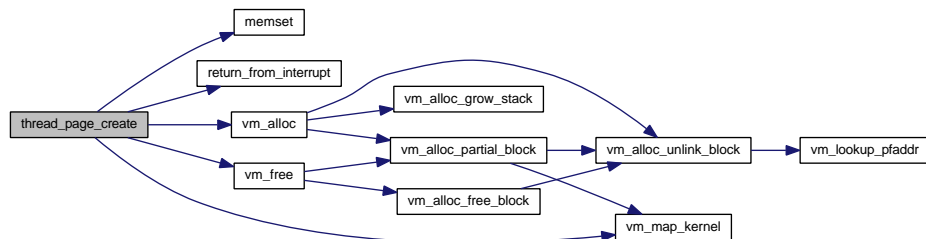
```

```

102     /* initialize fields */
103     thread_context_t *thread_ctx = &thread->thread_ctx;
104
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->savestack_pointer = (addr_t)kernel_context;
133 }
134
135 return thread;
136 }

```

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.

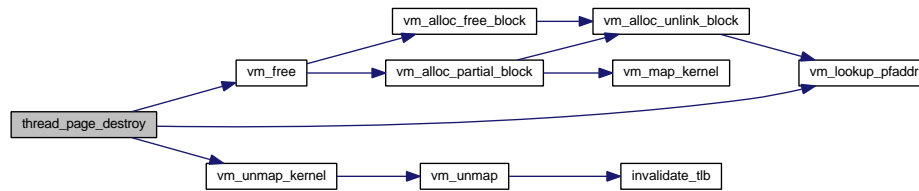
References global_page_allocator, NULL, pffree, vm_free(), vm_lookup_pfaddr(), and vm_unmap_kernel().

```

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 }

```

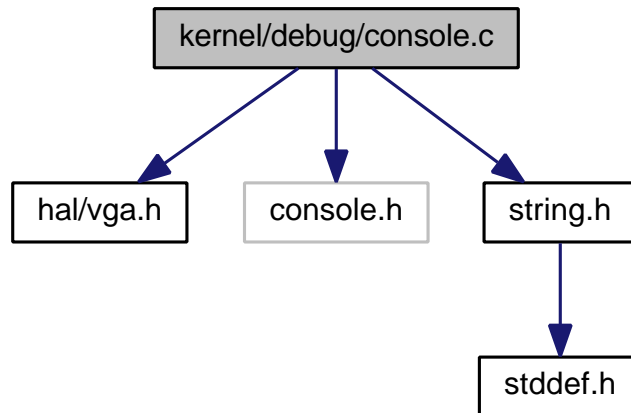
Here is the call graph for this function:



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.

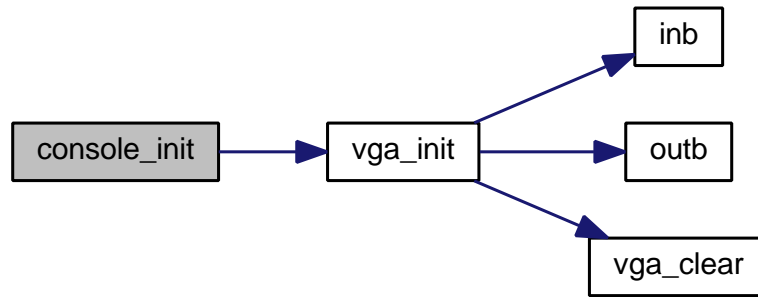
References `vga_init()`.

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)

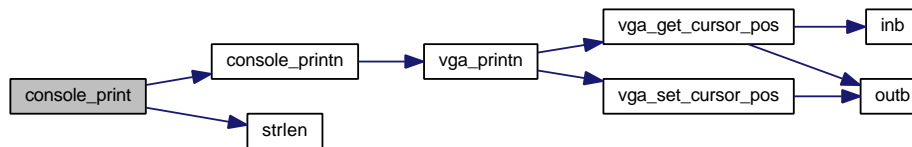
Definition at line 49 of file console.c.

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.

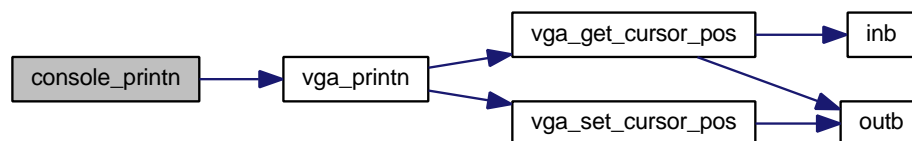
References `vga_printn()`.

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().

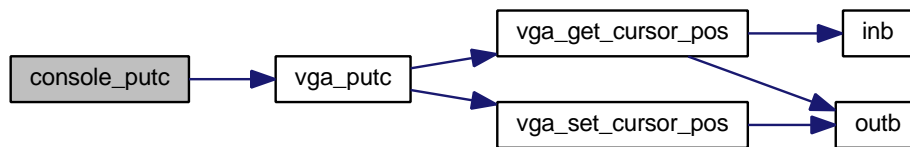
Referenced by dispatch_syscall().

```

45      {
46          vga_putc(c);
47      }

```

Here is the call graph for this function:



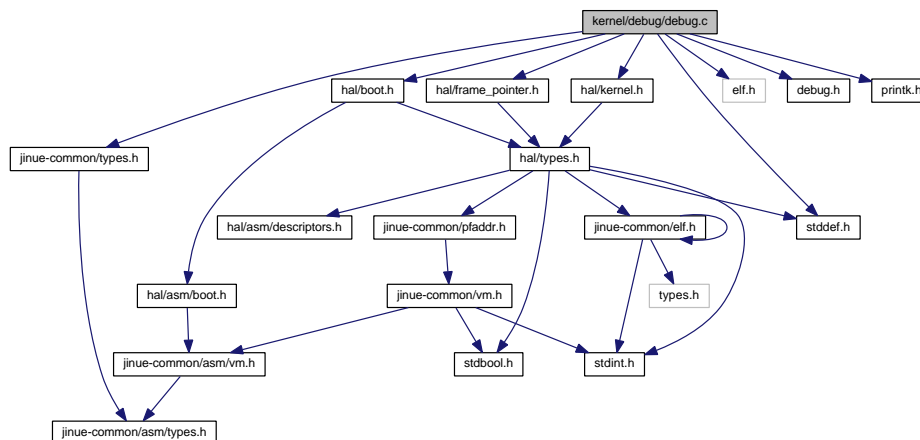
4.85 kernel/debug/debug.c File Reference

```

#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.

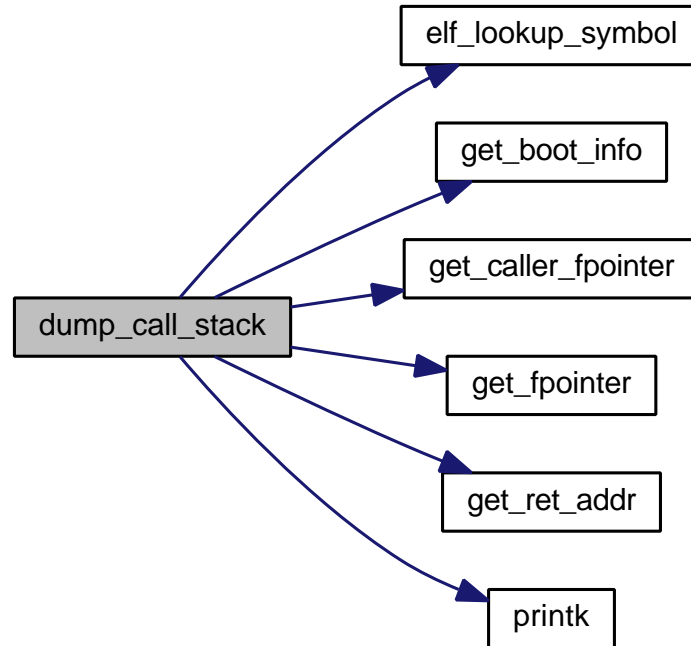
References `elf_symbol_t::addr`, `boot_info`, `elf_lookup_symbol()`, `get_boot_info()`, `get_caller_fpointer()`, `get_fpointer()`, `get_ret_addr()`, `boot_info_t::kernel_start`, `elf_symbol_t::name`, `NULL`, `printk()`, and `STT_FUNCTION`.

Referenced by `panic()`.

```

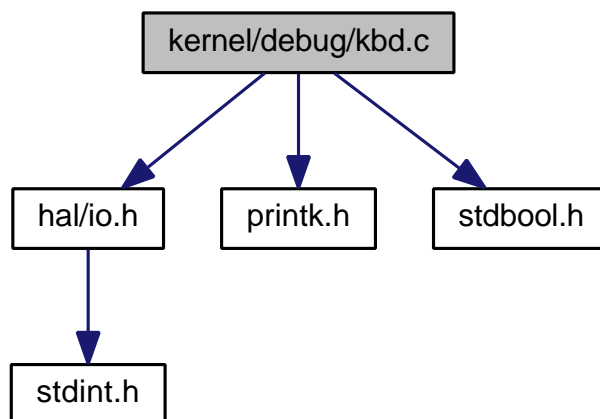
42     {
43     addr_t      fptr;
44
45     const boot_info_t *boot_info = get_boot_info();
46
47     printk("Call stack dump:\n");
48
49     fptr = get_fpointer();
50
51     while(fptr != NULL) {
52         addr_t return_addr = get_ret_addr(fptr);
53         if(return_addr == NULL) {
54             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

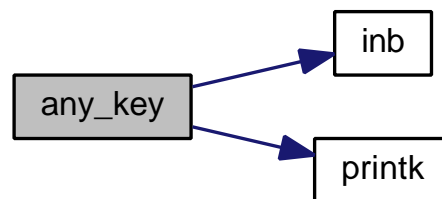
4.86.1.1 void any_key (void)

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 }
```

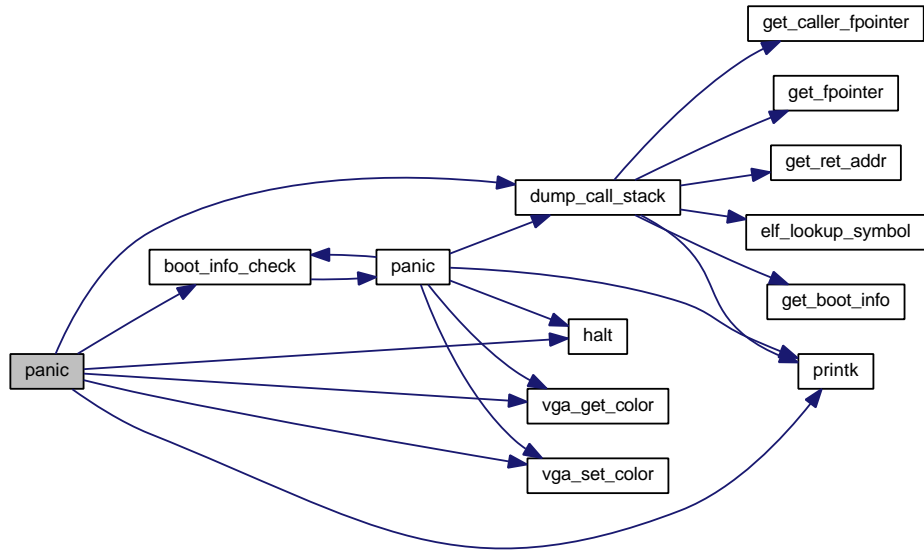
Here is the call graph for this function:



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>
```

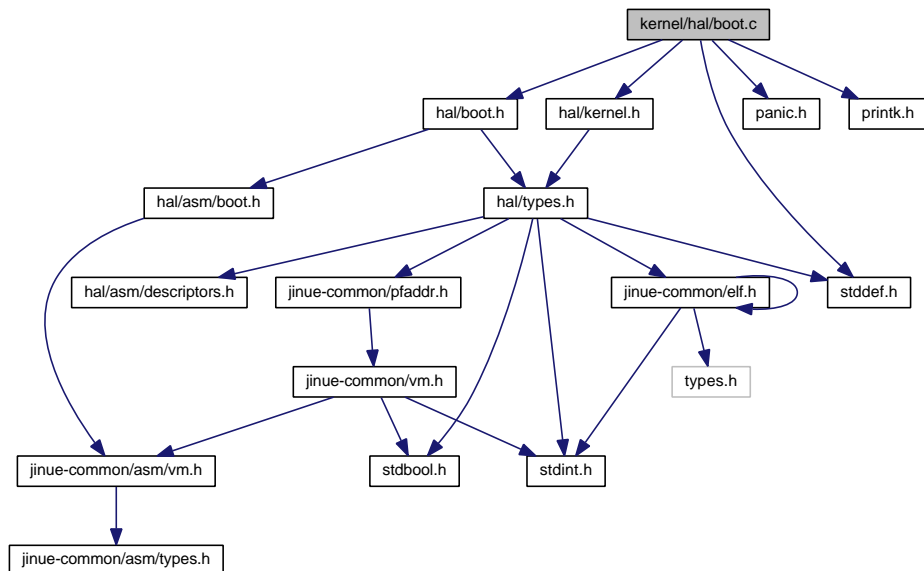

Here is the call graph for this function:



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.

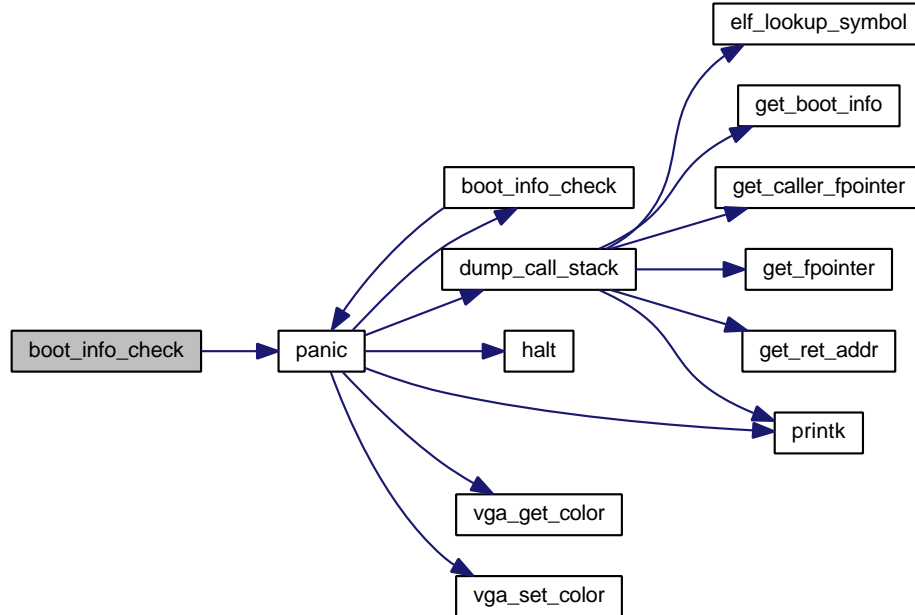
References BOOT_SETUP_MAGIC, NULL, panic(), and boot_info_t::setup_signature.

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.

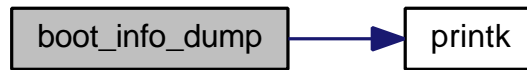
References boot_info_t::boot_end, boot_info_t::boot_heap, boot_info_t::e820_entries, boot_info_t::e820_map, boot_info_t::image_start, boot_info_t::image_top, boot_info_t::kernel_size, boot_info_t::kernel_start, boot_info_t::page_directory, boot_info_t::page_table, printk(), boot_info_t::proc_size, boot_info_t::proc_start, 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>
```

- 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)

- **bootmem_t * ram_map**
kernel memory map
- **bootmem_t * bootmem_root**
available memory map (allocator)
- **void * boot_heap**
current top of boot heap

```
4.89.1.1 void apply_mem_hole ( e820_addr_t hole_start, e820_addr_t hole_end, bootmem_t ** head )
```

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

```

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(dp_ptr = head, ptr = *head; ptr != NULL; dp_ptr = &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          *dp_ptr = 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
110     /* fix size or addr if block overlaps hole */
111     if(addr >= hole_addr && addr < hole_top) {
112         ptr->addr = hole_top;
113         ptr->count = (top - hole_top) / (PAGE_SIZE >> PFADDR_SHIFT);
114
115         return;
116     }
117
118     if(top > hole_addr && top <= hole_top) {
119         ptr->count = (hole_addr - addr) / (PAGE_SIZE >> PFADDR_SHIFT);
120     }
121 }
122 }

```

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;

```

```

251     }
252
253     return block;
254 }

```

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
133         /* copy the available ram entries from the e820 map and insert them
134          * in a linked list */
135         ram_map = NULL;
136
137         for(idx = 0; idx < boot_info->e820_entries; ++idx) {
138             const e820_t *e820_entry = &boot_info->e820_map[idx];
139
140             if(! e820_is_valid(e820_entry)) {
141                 continue;
142             }
143
144             if( e820_is_available(e820_entry) ) {
145                 /* get memory entry start and end addresses */
146                 e820_addr_t start = e820_entry->addr;
147                 e820_addr_t end = start + e820_entry->size;
148
149                 /* align on page boundaries */
150                 if( OFFSET_OF(start, PAGE_SIZE) != 0 ) {
151                     start = (start & (e820_addr_t)~PAGE_MASK) + PAGE_SIZE;
152                 }
153
154                 if( OFFSET_OF(end, PAGE_SIZE) != 0 ) {
155                     end = (end & (e820_addr_t)~PAGE_MASK);
156                 }
157
158                 /* If Physical Address Extension (PAE) is disabled, memory above the
159                  * 4GB mark is not usable. */
160                 if(! use_pae) {
161                     /* If this memory region is completely above the 4GB mark, exclude it. */
162                     if(start >= ADDR_4GB) {
163                         continue;
164                     }
165
166                     /* If this memory region starts below the 4GB mark but extends
167                      * beyond it, crop at 4GB. */
168                     if(end > ADDR_4GB) {
169                         end = ADDR_4GB;
170                     }
171                 }
172
173                 /* add entry to linked list */
174                 if(end > start) {
175                     new_ram_map_entry(ADDR_TO_PFADDR(start), (uint32_t)(end - start) /
176                                     PAGE_SIZE, &ram_map);
177                 }
178             }
179         }
180     }

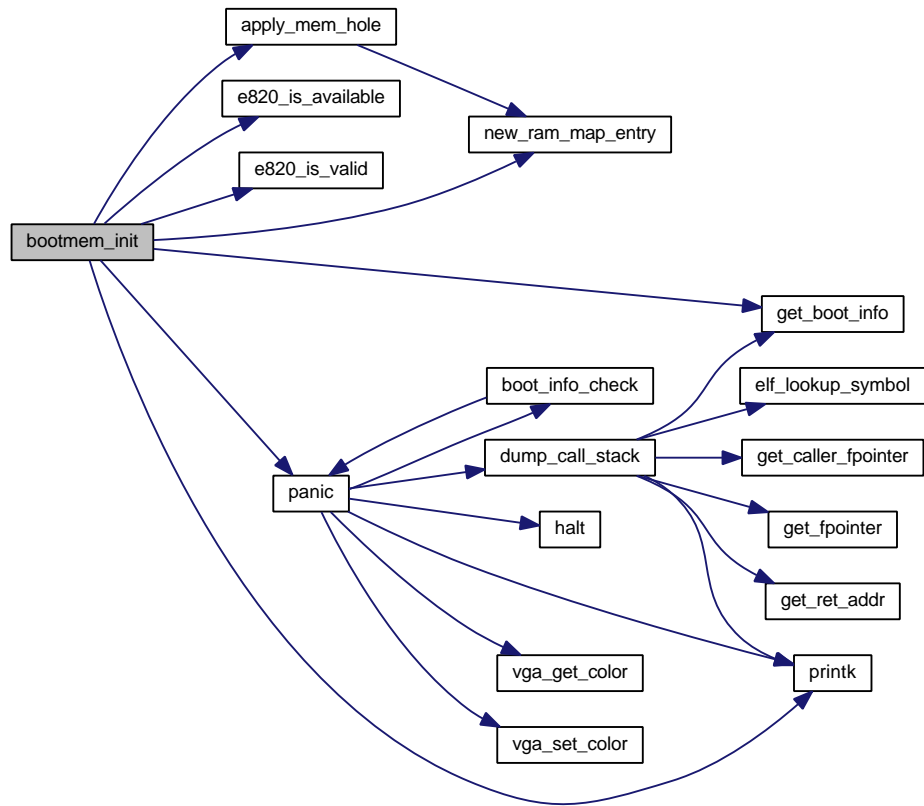
```

```

179     }
180 }
181
182 /* apply every unavailable entries from the e820 map as holes */
183 for(idx = 0; idx < boot_info->e820_entries; ++idx) {
184     const e820_t *e820_entry = &boot_info->e820_map[idx];
185
186     if(! e820_is_valid(e820_entry)) {
187         continue;
188     }
189
190     if( e820_is_available(e820_entry) ) {
191         continue;
192     }
193
194     e820_addr_t start = e820_entry->addr;
195     e820_addr_t end   = start + e820_entry->size;
196
197     apply_mem_hole(start, end, &ram_map);
198 }
199
200 /* Apparently, the first 64k of memory are corrupted by some BIOSes.
201  * It would be nice to try to detect this. In the meantime, let's
202  * assume the problem is present. */
203 apply_mem_hole(0, 0x10000, &ram_map);
204
205 /* the kernel image, its heap and stack, and early-allocated pages */
206 apply_mem_hole((uint32_t)boot_info->image_start, (uint32_t)kernel_region_top, &
ram_map);
207
208 /* Entry removal may have left garbage on the heap (bootmem_t
209  * structures which were allocated on the heap but are no longer
210  * linked). Let's clean up. */
211 temp_root = NULL;
212
213 for(ptr = ram_map; ptr != NULL; ptr = ptr->next) {
214     new_ram_map_entry(ptr->addr, ptr->count, &temp_root);
215 }
216
217 ram_map = NULL;
218 boot_heap = initial_boot_heap;
219
220 for(ptr = temp_root; ptr != NULL; ptr = ptr->next) {
221     new_ram_map_entry(ptr->addr, ptr->count, &ram_map);
222 }
223
224 /* at this point, we should have at least one block of available RAM */
225 if( ram_map == NULL ) {
226     panic("no available memory.");
227 }
228
229 /* Let's count and display the total amount of available memory */
230 uint32_t page_count = 0;
231 for(ptr = ram_map; ptr != NULL; ptr = ptr->next) {
232     page_count += ptr->count;
233 }
234
235 printk("%u kilobytes (%u pages) of memory available.\n",
236        (uint32_t)(page_count * PAGE_SIZE / KB),
237        (uint32_t)(page_count) );
238
239 /* head pointer for bootmem_get_block() */
240 bootmem_root = ram_map;
241
242 }

```

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

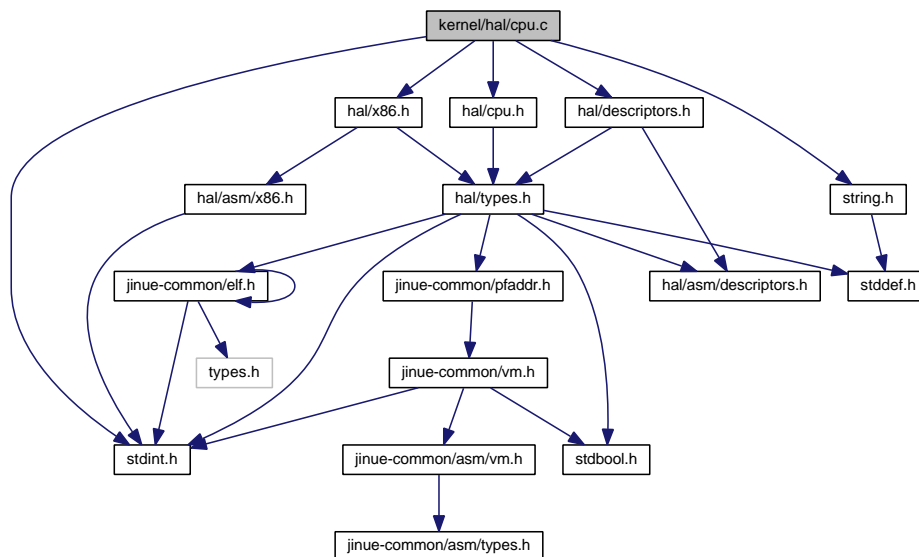
Definition at line 46 of file bootmem.c.

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`, `CPU_FEATURE_SYSCALL`, `CPU_FEATURE_SYSENTER`, `CPU_VENDOR_AMD`, `CPU_VENDOR_AMD_DW0`, `CPU_VENDOR_AMD_DW1`, `CPU_VENDOR_AMD_DW2`, `CPU_VENDOR_GENERIC`, `CPU_VENDOR_INTEL`, `CPU_VENDOR_INTEL_DW0`, `CPU_VENDOR_INTEL_DW1`, `CPU_VENDOR_INTEL_DW2`, `cpuid()`, `CPUID_EXT_FEATURE_SYSCALL`, `CPUID_FEATURE_APIC`, `CPUID_FEATURE_CLFLUSH`, `CPUID_FEATURE_PAE`, `CPUID_FEATURE_SEP`, `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

```

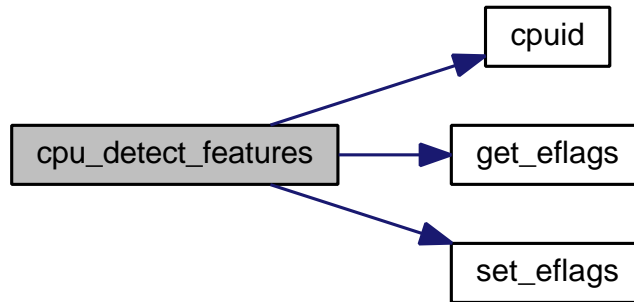
```

137         && vendor_dw1 == CPU_VENDOR_INTEL_DW1
138         && vendor_dw2 == CPU_VENDOR_INTEL_DW2) {
139
140     cpu_info.vendor = CPU_VENDOR_INTEL;
141 }
142
143 /* get processor signature (family/model/stepping) and feature flags */
144 if(cpuid_max >= 1) {
145     /* function 1: processor signature and feature flags */
146     regs.eax = 1;
147
148     /* call CPUID instruction */
149     signature = cpuid(&regs);
150
151     /* set processor signature */
152     cpu_info.stepping = signature & 0xf;
153     cpu_info.model = (signature>>4) & 0xf;
154     cpu_info.family = (signature>>8) & 0xf;
155
156     /* feature flags */
157     flags = regs.edx;
158
159     /* cache alignment */
160     if(flags & CPUID_FEATURE_CLFLUSH) {
161         cpu_info.dcache_alignment = ((regs.ebx >> 8) & 0xff) * 8;
162     }
163 }
164
165 /* extended function 0: max value of eax when calling CPUID (extended function) */
166 regs.eax = 0x80000000;
167 cpuid_ext_max = cpuid(&regs);
168
169 /* get extended feature flags */
170 if(cpuid_ext_max >= 0x80000001) {
171     /* extended function 1: extended feature flags */
172     regs.eax = 0x80000001;
173     (void)cpuid(&regs);
174
175     /* extended feature flags */
176     ext_flags = regs.edx;
177 }
178
179 /* support for SYSENTER/SYSEXIT instructions */
180 if(flags & CPUID_FEATURE_SEP) {
181     if(cpu_info.vendor == CPU_VENDOR_AMD) {
182         cpu_info.features |= CPU_FEATURE_SYSENTER;
183     }
184     else if(cpu_info.vendor == CPU_VENDOR_INTEL) {
185         if(cpu_info.family == 6 && cpu_info.model < 3 && cpu_info.
stepping < 3) {
186             /* not supported */
187         }
188         else {
189             cpu_info.features |= CPU_FEATURE_SYSENTER;
190         }
191     }
192 }
193
194 /* support for SYSCALL/SYSRET instructions */
195 if(cpu_info.vendor == CPU_VENDOR_AMD) {
196     if(ext_flags & CPUID_EXT_FEATURE_SYSCALL) {
197         cpu_info.features |= CPU_FEATURE_SYSCALL;
198     }
199 }
200
201 /* support for local APIC */
202 if(cpu_info.vendor == CPU_VENDOR_AMD || cpu_info.vendor ==
CPU_VENDOR_INTEL) {
203     if(flags & CPUID_FEATURE_APIC) {
204         cpu_info.features |= CPU_FEATURE_LOCAL_APIC;
205     }
206 }
207
208 /* support for physical address extension (PAE) */
209 if(cpu_info.vendor == CPU_VENDOR_AMD || cpu_info.vendor ==
CPU_VENDOR_INTEL) {
210     if(flags & CPUID_FEATURE_PAE) {
211         cpu_info.features |= CPU_FEATURE_PAE;
212     }
213 }
214 }

```

215 }

Here is the call graph for this function:



4.90.1.2 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, 0xffff, SEG_TYPE_CODE |
58         SEG_FLAG_KERNEL | SEG_FLAG_NORMAL);
59
60     data->gdt[GDT_KERNEL_DATA] =
61         SEG_DESCRIPTOR( 0, 0xffff, SEG_TYPE_DATA |
62         SEG_FLAG_KERNEL | SEG_FLAG_NORMAL);
63
64     data->gdt[GDT_USER_CODE] =
65         SEG_DESCRIPTOR( 0, 0xffff, SEG_TYPE_CODE |
66         SEG_FLAG_USER | SEG_FLAG_NORMAL);
67
68     data->gdt[GDT_USER_DATA] =
69         SEG_DESCRIPTOR( 0, 0xffff, SEG_TYPE_DATA |
70         SEG_FLAG_USER | SEG_FLAG_NORMAL);
71
72     data->gdt[GDT_TSS] =
73         SEG_DESCRIPTOR( tss, TSS_LIMIT-1, SEG_TYPE_TSS |
74         SEG_FLAG_KERNEL | SEG_FLAG_TSS);
75
76     data->gdt[GDT_PER_CPU_DATA] =
77         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);
73
74     data->gdt[GDT_USER_TLS_DATA] = SEG_DESCRIPTOR(0, 0, 0);
75
76     /* setup kernel stack in TSS */
77     tss->ss0 = SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL);
78     tss->ss1 = SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL);
79     tss->ss2 = SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL);
80
81     /* kernel stack address is updated by thread_context_switch() */
82     tss->esp0 = NULL;
83     tss->esp1 = NULL;
84     tss->esp2 = NULL;
85 }

```

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()`.

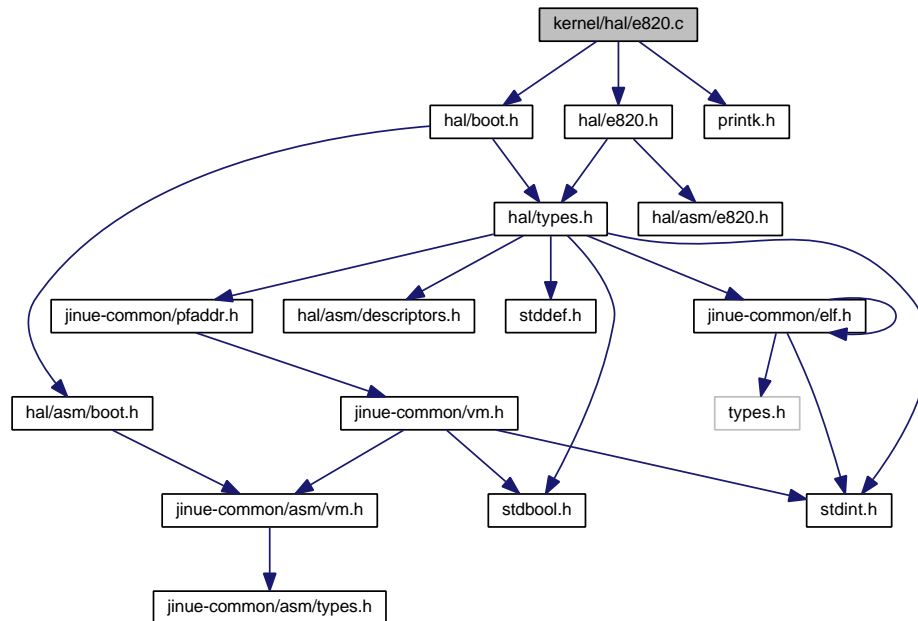
4.91 `kernel/hal/e820.c` File Reference

```

#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.

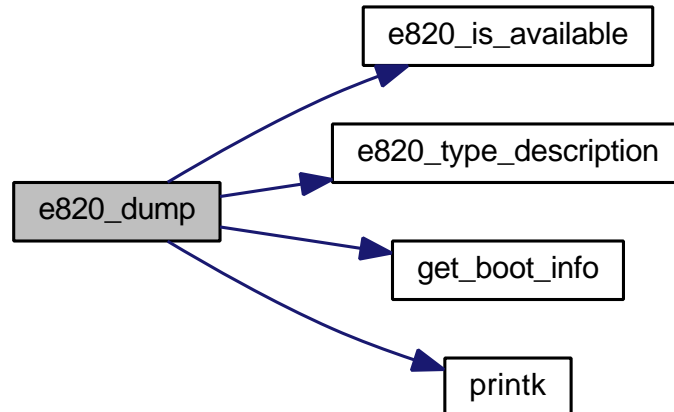
References **e820_t::addr**, **boot_info**, **boot_info_t::e820_entries**, **e820_is_available()**, **boot_info_t::e820_map**, **e820_type_description()**, **get_boot_info()**, **printk()**, **e820_t::size**, and **e820_t::type**.

```

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 }
```

```

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 }

```

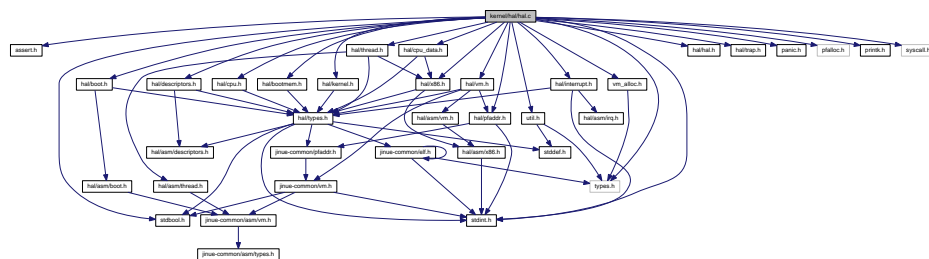
4.92 kernel/hal/hal.c File Reference

```

#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 <pfallloc.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
82         assert(page_offset_of(boot_info->image_start) == 0);
83
84         assert(page_offset_of(boot_info->kernel_start) == 0);
85
86         printk("Kernel size is %u bytes.\n", boot_info->kernel_size);
87
88         /* This must be done before any boot heap allocation. */
89         boot_heap = boot_info->boot_heap;
90
91         /* This must be done before any call to pfalloc_early(). */
92         kernel_region_top = boot_info->boot_end;
93
94         /* get cpu info */
95         cpu_detect_features();
96
97         /* allocate new kernel stack */

```



```

100     stack = pfalloc_early();
101     stack += PAGE_SIZE;
102
103     /* allocate per-CPU data
104     *
105     * We need to ensure that the Task State Segment (TSS) contained in this
106     * memory block does not cross a page boundary. */
107     assert(sizeof(cpu_data_t) < CPU_DATA_ALIGNMENT);
108
109     boot_heap = ALIGN_END(boot_heap, CPU_DATA_ALIGNMENT);
110
111     cpu_data = boot_heap;
112     boot_heap = cpu_data + 1;
113
114     /* initialize per-CPU data */
115     cpu_init_data(cpu_data, stack);
116
117     /* allocate pseudo-descriptor for GDT and IDT (temporary allocation) */
118     boot_heap_old = boot_heap;
119
120     boot_heap = ALIGN_END(boot_heap, sizeof(pseudo_descriptor_t));
121
122     pseudo = (pseudo_descriptor_t *)boot_heap;
123     boot_heap = (pseudo_descriptor_t *)boot_heap + 1;
124
125     /* load new GDT and TSS */
126     pseudo->addr = (addr_t)&cpu_data->gdt;
127     pseudo->limit = GDT_LENGTH * 8 - 1;
128
129     lgdt(pseudo);
130
131     set_cs( SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL) );
132     set_ss( SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL) );
133     set_data_segments( SEG_SELECTOR(GDT_KERNEL_DATA, RPL_KERNEL) );
134     set_gs( SEG_SELECTOR(GDT_PER_CPU_DATA, RPL_KERNEL) );
135
136     ltr( SEG_SELECTOR(GDT_TSS, RPL_KERNEL) );
137
138     /* initialize IDT */
139     for(idx = 0; idx < IDT_VECTOR_COUNT; ++idx) {
140         /* get address, which is already stored in the IDT entry */
141         addr = (addr_t)(uintptr_t)idt[idx];
142
143         /* set interrupt gate flags */
144         flags = SEG_TYPE_INTERRUPT_GATE | SEG_FLAG_NORMAL_GATE;
145
146         if(idx == SYSCALL_IRQ) {
147             flags |= SEG_FLAG_USER;
148         }
149         else {
150             flags |= SEG_FLAG_KERNEL;
151         }
152
153         /* create interrupt gate descriptor */
154         idt[idx] = GATE_DESCRIPTOR(
155             SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL),
156             addr,
157             flags,
158             NULL );
159     }
160
161     pseudo->addr = (addr_t)idt;
162     pseudo->limit = IDT_VECTOR_COUNT * sizeof(seg_descriptor_t) - 1;
163     lidt(pseudo);
164
165     /* de-allocate pseudo-descriptor */
166     boot_heap = boot_heap_old;
167
168     /* initialize the page frame allocator */
169     page_stack_buffer = (paddr_t *)pfalloc_early();
170     init_pfcache(&global_pfcache, page_stack_buffer);
171
172     for(idx = 0; idx < KERNEL_PAGE_STACK_INIT; ++idx) {
173         pffree( EARLY_PTR_TO_PFADDR( pfalloc_early() ) );
174     }
175
176     /* initialize virtual memory management, enable paging
177     *
178     * below this point, it is no longer safe to call pfalloc_early() */
179     vm_boot_init();
180

```

```

181  /* choose system call method */
182  syscall_method = SYSCALL_METHOD_INTR;
183
184  if(cpu_has_feature(CPU_FEATURE_SYSENTER)) {
185      syscall_method = SYSCALL_METHOD_FAST_INTEL;
186
187      wrmsr(MSR_IA32_SYSENTER_CS,  SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL));
188      wrmsr(MSR_IA32_SYSENTER_EIP, (uint64_t)(uintptr_t)fast_intel_entry);
189
190      /* kernel stack address is set when switching thread context */
191      wrmsr(MSR_IA32_SYSENTER_ESP, (uint64_t)(uintptr_t)NULL);
192  }
193
194  if(cpu_has_feature(CPU_FEATURE_SYSCALL)) {
195      syscall_method = SYSCALL_METHOD_FAST_AMD;
196
197      msrval = rdmsr(MSR_EFER);
198      msrval |= MSR_FLAG_STAR_SCE;
199      wrmsr(MSR_EFER, msrval);
200
201      msrval = (uint64_t)(uintptr_t)fast_amd_entry;
202      msrval |= (uint64_t)SEG_SELECTOR(GDT_KERNEL_CODE, RPL_KERNEL) << 32;
203      msrval |= (uint64_t)SEG_SELECTOR(GDT_USER_CODE,   RPL_USER)   << 48;
204
205      wrmsr(MSR_STAR, msrval);
206  }
207 }

```

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>
```

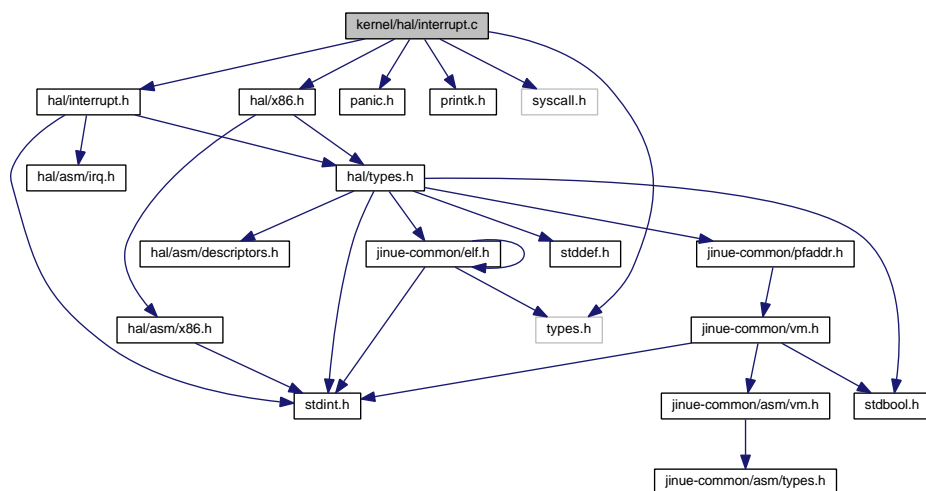
```
#include <panic.h>
```

```
#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.

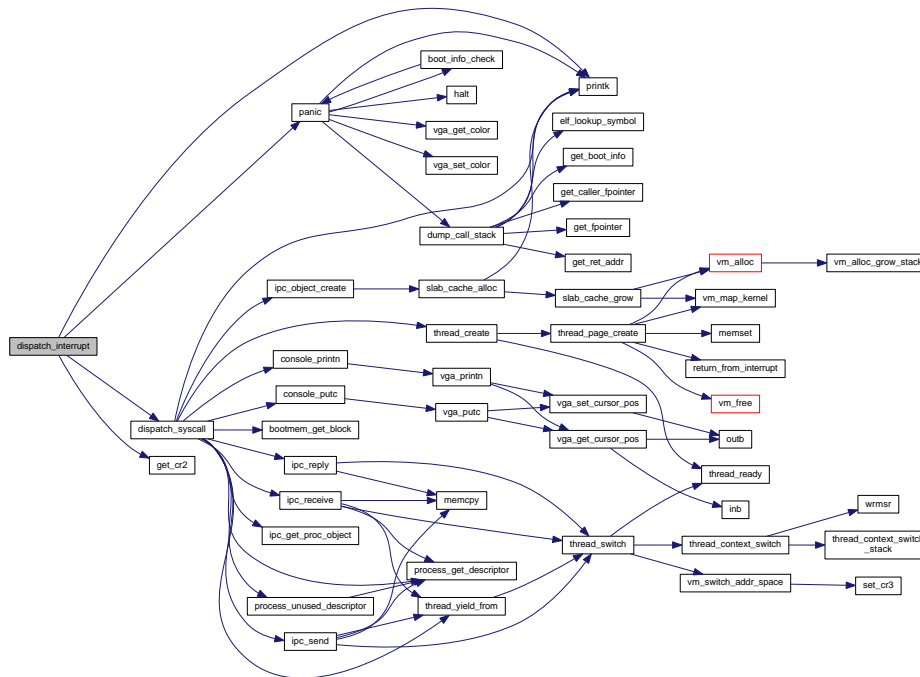
References `dispatch_syscall()`, `trapframe_t::eip`, `trapframe_t::errcode`, `get_cr2()`, `IDT_FIRST_IRQ`, `trapframe_t::ivt`, `panic()`, `printk()`, and `SYSCALL_IRQ`.

```

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:

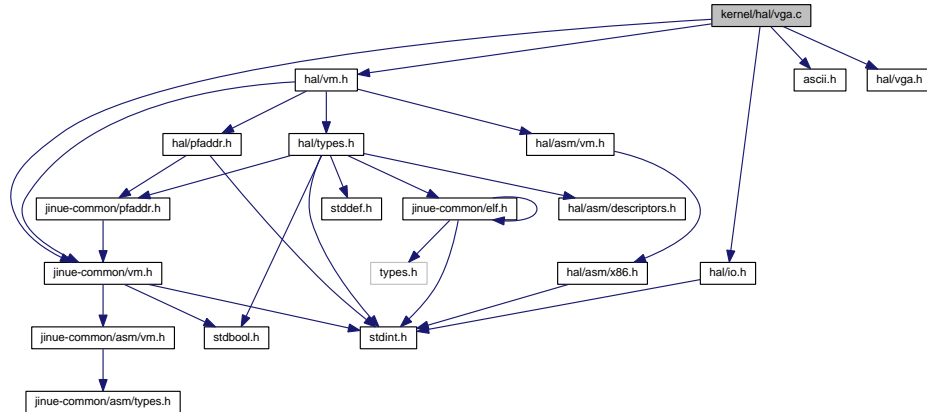


4.94 kernel/hal/vga.c File Reference

```
#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.

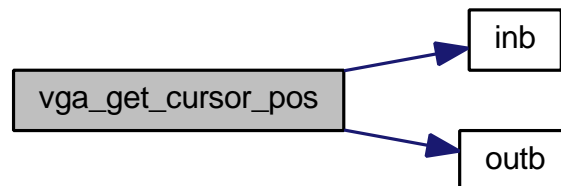
Referenced by vga_print(), vga_printn(), and vga_putc().

```

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 */

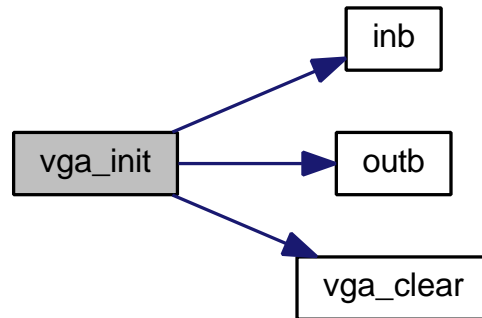
```

```

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.

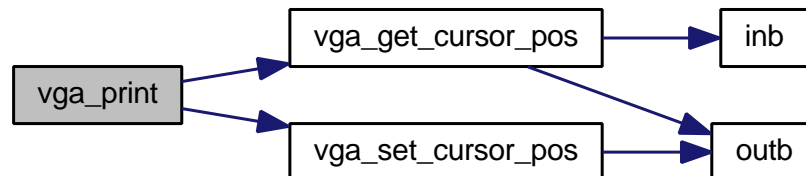
References `vga_get_cursor_pos()`, and `vga_set_cursor_pos()`.

```

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()`.

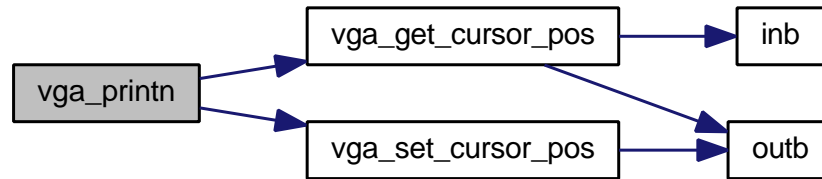
Referenced by `console_printn()`.

```

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()`.

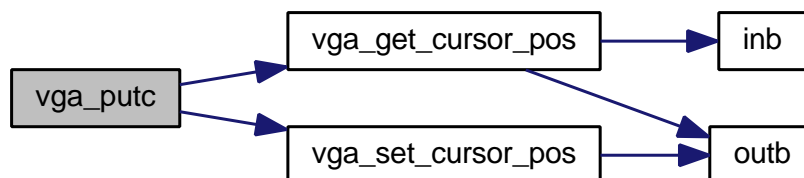
Referenced by `console_putc()`.

```

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`.

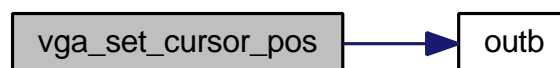
Referenced by `vga_print()`, `vga_printn()`, and `vga_putc()`.

```

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:



- **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) {

```

```

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
543     * same address once paging is enabled. */
544     page_table = (pte_t *)pfalloc_early();
545
546     if(first_pd && idx == start_index) {
547         /* remember the address of the first page table for use by
548         * vm_map() later */
549         global_page_tables = page_table;
550     }
551
552     set_pte(
553         get_pte_with_offset(page_directory, idx),
554         EARLY_PTR_TO_PFADDR(page_table),
555         VM_FLAG_PRESENT | VM_FLAG_READ_WRITE );
556
557     /* clear page table */
558     for(idy = 0; idy < page_table_entries; ++idy) {
559         clear_pte( get_pte_with_offset(page_table, idy) );
560     }
561 }
562
563 return page_directory;
564 }

```

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
105         use_pfalloc_early = false;
106
107         /* 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 +=
126      PAGE_SIZE) {
127          vm_map_early((addr_t)addr, EARLY_PTR_TO_PFADDR(addr), VM_FLAG_KERNEL |
128      VM_FLAG_READ_WRITE);
129      }
130
131      /* map VGA text buffer in the new address space
132      *
133      * This is a good place to do this because:
134      *
135      * 1) It is our last chance to allocate a continuous region of virtual memory.
136      *    Once the page allocator is initialized (see call to vm_alloc_init_allocator()
137      *    below) and we start using vm_alloc() to allocate memory, pages can only
138      *    be allocated one at a time.
139      *
140      * 2) Doing this last makes things simpler because this is the only place where
141      *    we have to allocate a continuous region of virtual memory but no physical
142      *    memory to back it. To allocate it, we just have to increase kernel_vm_top,
143      *    which represents the end of the virtual memory region that is used by the
144      *    kernel. */
145      addr_t kernel_vm_top = kernel_region_top;
146      addr = (addr_t)VGA_TEXT_VID_BASE;
147
148      addr_t vga_text_base = kernel_vm_top;
149
150      while(addr < (addr_t)VGA_TEXT_VID_TOP) {
151          vm_map_early(kernel_vm_top, ADDR_TO_PFADDR((uintptr_t)addr),
152      VM_FLAG_KERNEL | VM_FLAG_READ_WRITE);
153          kernel_vm_top += PAGE_SIZE;
154          addr += PAGE_SIZE;
155      }
156
157      /* remap VGA text buffer
158      *
159      * Note: after the call to vga_set_base_addr() below until we switch to the
160      * new address space, VGA output is not possible. Calling printk() will cause
161      * a kernel panic due to a page fault (and the panic handler calls printk()). */
162      printk("Remapping text video memory at 0x%x\n", kernel_vm_top);
163
164      vga_set_base_addr(vga_text_base);
165
166      if(use_pae) {
167          /* If we are enabling PAE, this is where the switch to the new page
168          * tables actually happens instead of at the call to vm_switch_addr_space()
169          * as would be expected.
170          *
171          * From Intel 64 and IA-32 Architectures Software Developer's Manual
172          * Volume 3: System Programming Guide, section 4.4.1 "PDPTE Registers":
173          *
174          * " The logical processor loads [the PDPTE] registers from the PDPTEs
175          *   in memory as part of certain operations:
176          *
177          *   * If PAE paging would be in use following an execution of MOV to
178          *     CR0 or MOV to CR4 (see Section 4.1.1) and the instruction is
179          *     modifying any of (...) CR4.PAE, (...); then the PDPTEs are
180          *     loaded from the address in CR3. "
181          *
182          * There are bootstrapping issues when enabling PAE while paging is enabled.
183          * See the comment at the top of the vm_pae_create_initial_addr_space()
184          * function in vm_pae.c for more detail. */
185          vm_pae_enable();
186      }
187
188      /* switch to new address space */
189      vm_switch_addr_space(addr_space);
190
191

```



```

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:



4.95.1.4 pfaddr_t vm_clone_page_directory (pfaddr_t template_pfaddr, unsigned int start_index)

Definition at line 472 of file vm.c.

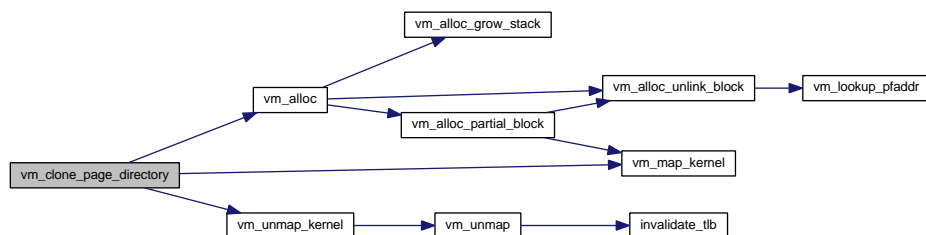
References `clear_pte`, `copy_pte`, `get_pte_with_offset`, `page_table_entries`, `pfalloc`, `vm_alloc()`, `VM_FLAG_READ_WRITE`, `vm_map_kernel()`, and `vm_unmap_kernel()`.

```

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 (pfnaddr_t pdpfaddr, unsigned int from_index, unsigned int to_index)`

Definition at line 581 of file `vm.c`.

References `get_pte_flags`, `get_pte_pfnaddr`, `get_pte_with_offset`, `pfnfree`, `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) {
```

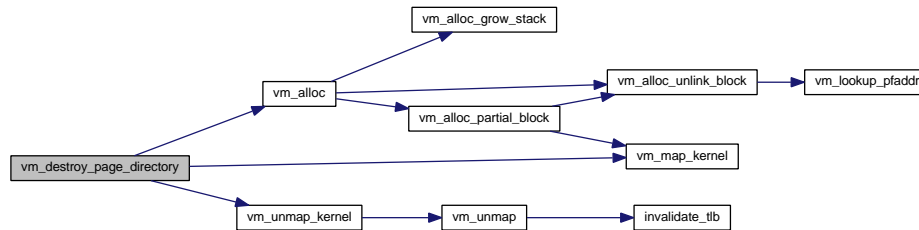


```

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:



4.95.1.9 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
434     assert(pte != NULL && (get_pte_flags(pte) & VM_FLAG_PRESENT));
435
436     pfaddr_t pfaddr = get_pte_pfaddr(pte);
437
438     vm_free_page_table_entry(addr, pte);
439
440     return pfaddr;
441 }
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
462     assert( is_fast_map_pointer(vaddr) );
463
464     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().

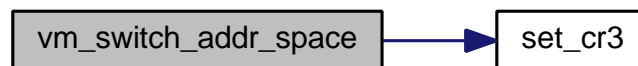
Referenced by thread_switch(), and vm_boot_init().

```

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

<i>addr_space</i>	address space from which to unmap, can be NULL for global mappings (<i>addr</i> >= KLIMIT)
<i>addr</i>	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         vm_free_page_table_entry(addr, pte);
408
409         /* invalidate TLB entry for newly mapped page */
410         invalidate_tlb(addr);
411     }
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.

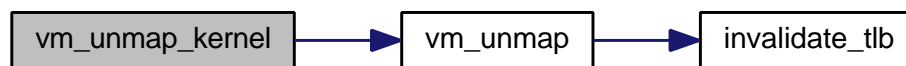
References `NULL`, and `vm_unmap()`.

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)

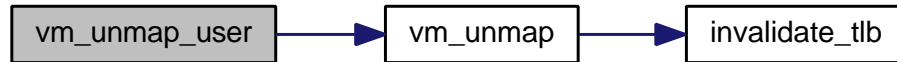
Definition at line 427 of file vm.c.

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)

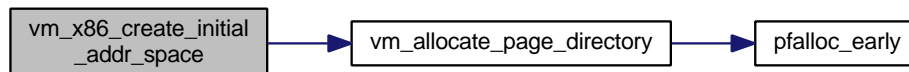
Definition at line 566 of file vm.c.

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`.

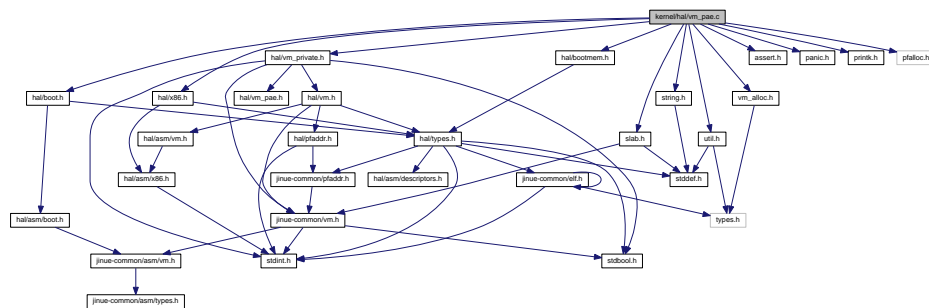
Referenced by `vm_change_flags()`, and `vm_pae_boot_init()`.

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`.

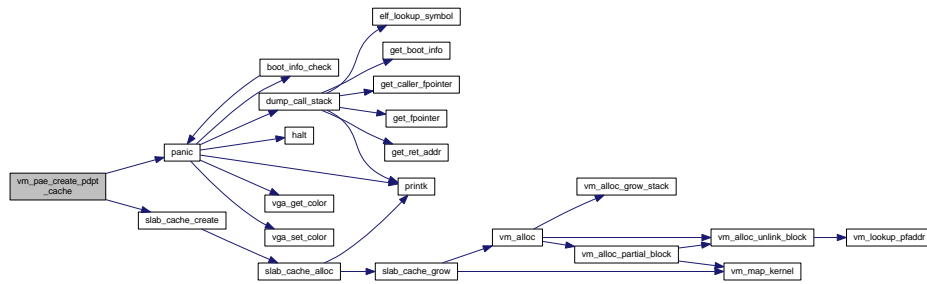
References `NULL`, `panic()`, `slab_cache_create()`, and `SLAB_DEFAULTS`.

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**.

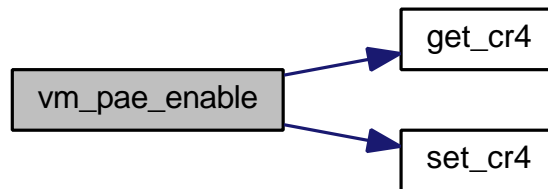
Referenced by **vm_boot_init()**.

```

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**.

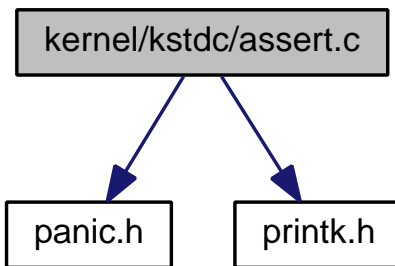
4.97 kernel/kstdd/assert.c File Reference

```

#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

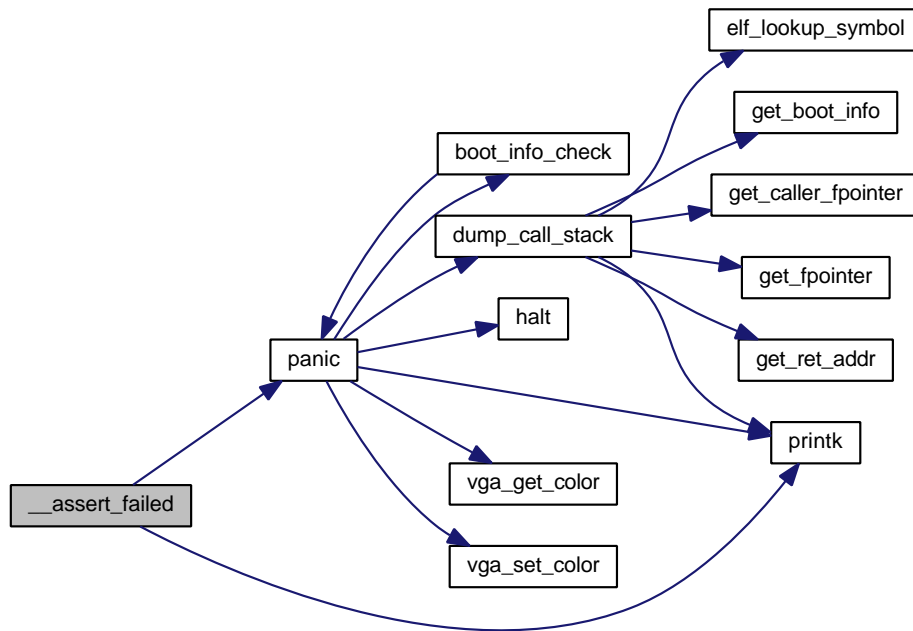
4.97.1.1 `void __assert_failed (const char * expr, const char * file, unsigned int line, const char * func)`

Definition at line 36 of file `assert.c`.

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

```
40         {
41
42     printk(
43         "ASSERTION FAILED [%s]: %s at line %u in function %s.\n",
44         expr, file, line, func );
45
46     panic("Assertion failed.");
47 }
```

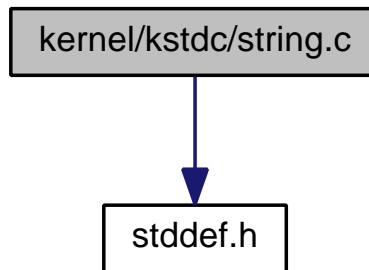
Here is the call graph for this function:



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()`.

```

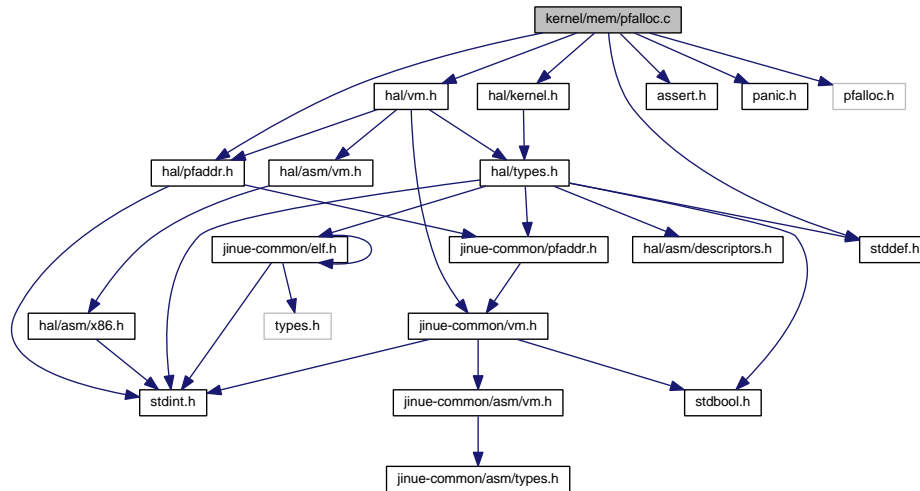
57                                     {
58     size_t count = 0;
59
60     while(*s != 0) {
61         ++s;
62         ++count;
63     }
64
65     return count;
66 }
```

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
49      assert (use_pfalloc_early);
50
51      page = kernel_region_top;
52      kernel_region_top += PAGE_SIZE;
53
54      return page;
55 }
56 }
```

4.99.1.3 `pfaddr_t pfalloc_from (pfcache_t* pfcache)`

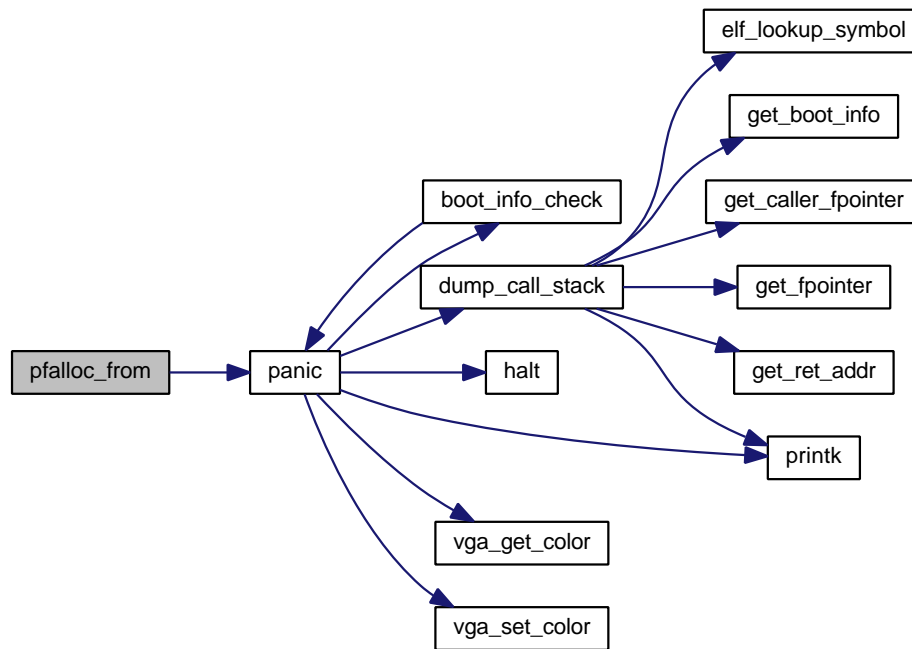
ASSERTION: `pfalloc_early` must be used early

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

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

```
72      {
73      assert ( ! use_pfalloc_early );
74
75      if (pfcache->count == 0) {
76          panic("pfalloc_from(): no more pages to allocate");
77      }
78
79      --pfcache->count;
80
81      return * (--pfcache->ptr);
82 }
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) {
87             return;
88         }
89     }
90     ++pfcache->count;
91     (pfcache->ptr++)[0] = pf;
92 }

```

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

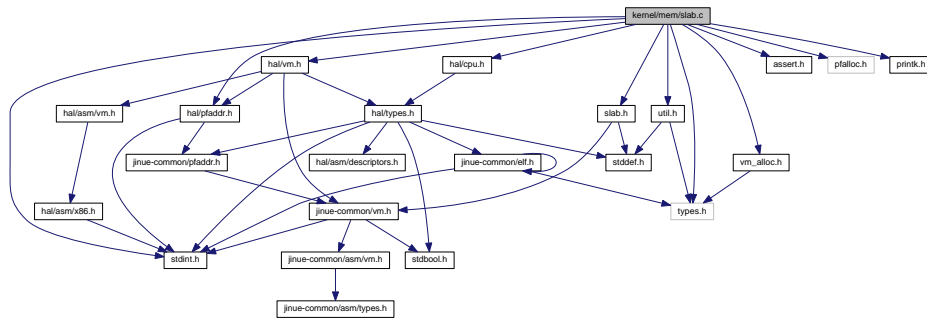
Definition at line 41 of file pfalloc.c.

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
245         assert(slab != NULL);
246
247         /* We are about to allocate one object from this slab, so it will
248          * not be empty anymore...*/
249         cache->slabs_empty = slab->next;
250
251         --(cache->empty_count);
252
253         slab->next = cache->slabs_partial;
254         if(slab->next != NULL) {
255             slab->next->prev = slab;
256         }
257         cache->slabs_partial = slab;
258     }
259
260     bufctl = slab->free_list;
261
262     assert(bufctl != NULL);
263
264     slab->free_list = bufctl->next;
265     slab->obj_count += 1;
266
267     /* If we just allocated the last buffer, move the slab to the full
268      * list */
269     if(slab->free_list == NULL) {
270         /* remove from the partial slabs list */
271
272         assert(cache->slabs_partial == slab);
273
274         cache->slabs_partial = slab->next;
275
276         if(slab->next != NULL) {
277             slab->next->prev = slab->prev;
278         }
279
280         /* add to the full slabs list */
281         slab->next = cache->slabs_full;
282         cache->slabs_full = slab;
283
284         if(slab->next != NULL) {
285             slab->next->prev = slab;
286         }
287     }
288 }

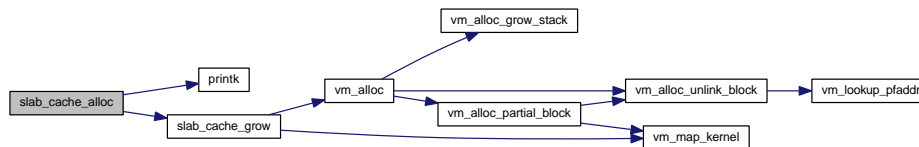
```

```

300
301     buffer = (uint32_t *) ( (char *)bufctl - cache->bufctl_offset );
302
303     if(cache->flags & SLAB_POISON) {
304         dump_lines = 0;
305
306         for(idx = 0; idx < cache->obj_size / sizeof(uint32_t); ++idx) {
307             if(buffer[idx] != SLAB_POISON_DEAD_VALUE) {
308                 if(dump_lines == 0) {
309                     printk("detected write to freed object, cache: %s buffer: 0x%x:\n",
310                         cache->name,
311                         (unsigned int)buffer
312                     );
313                 }
314
315                 if(dump_lines < 4) {
316                     printk(" value 0x%x at byte offset %u\n", buffer[idx], idx * sizeof(
317                         uint32_t));
318                 }
319
320                 ++dump_lines;
321             }
322             buffer[idx] = SLAB_POISON_ALIVE_VALUE;
323         }
324
325         /* If both SLAB_POISON and SLAB_RED_ZONE are enabled, we perform
326          * redzone checking even on freed objects. */
327         if(cache->flags & SLAB_RED_ZONE) {
328             if(buffer[idx] != SLAB_RED_ZONE_VALUE) {
329                 printk("detected write past the end of freed object, cache: %s buffer: 0x%x value: 0x%x\n",
330                     cache->name,
331                     (unsigned int)buffer,
332                     buffer[idx]
333                 );
334             }
335             buffer[idx] = SLAB_RED_ZONE_VALUE;
336         }
337     }
338
339     if(cache->ctor != NULL) {
340         cache->ctor((void *)buffer, cache->obj_size);
341     }
342 }
343 else if(cache->flags & SLAB_RED_ZONE) {
344     buffer[cache->obj_size / sizeof(uint32_t)] = SLAB_RED_ZONE_VALUE;
345 }
346
347 return (void *)buffer;
348 }

```

Here is the call graph for this function:



4.100.12 `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().

```

95         {
96
97     slab_cache_t    *cache;
98     size_t          avail_space;
99     size_t          wasted_space;
100     unsigned int    buffers_per_slab;
101
102
103     assert( size >= sizeof(void *) );
104
105     assert(name != NULL);
106
107     cache = slab_cache_alloc(&slab_cache_cache);
108
109     cache->name          = name;
110     cache->ctor          = ctor;
111     cache->dtor          = dtor;
112     cache->slabs_empty   = NULL;
113     cache->slabs_partial = NULL;
114     cache->slabs_full    = NULL;
115     cache->empty_count   = 0;
116     cache->flags         = flags;
117     cache->next_colour   = 0;
118     cache->working_set   = SLAB_DEFAULT_WORKING_SET;
119
120     /* add new cache to cache list */
121     cache->next          = slab_cache_list;
122     slab_cache_list      = cache;
123
124     if(cache->next != NULL) {
125         cache->next->prev = cache;
126     }
127
128     /* compute actual alignment */
129     if(alignment == 0) {
130         cache->alignment = sizeof(uint32_t);
131     }
132     else {
133         cache->alignment = alignment;
134     }
135
136     if((flags & SLAB_HWCACHE_ALIGN) && cache->alignment < cpu_info.
dcache_alignment) {
137         cache->alignment = cpu_info.dcache_alignment;
138     }
139
140     if(cache->alignment % sizeof(uint32_t) != 0) {
141         cache->alignment += sizeof(uint32_t) - cache->alignment % sizeof(
uint32_t);
142     }
143
144     /* reserve space for bufctl and/or redzone word */
145     cache->obj_size = size;
146
147     if(cache->obj_size % sizeof(uint32_t) != 0) {
148         cache->obj_size += sizeof(uint32_t) - cache->obj_size % sizeof(uint32_t);
149     }
150
151     if((flags & SLAB_POISON) && (flags & SLAB_RED_ZONE)) {
152         /* bufctl and redzone word appended to buffer */
153         cache->alloc_size = cache->obj_size + sizeof(uint32_t) + sizeof(
slab_bufctl_t);
154     }
155     else if((flags & SLAB_POISON) || (flags & SLAB_RED_ZONE)) {
156         /* bufctl and/or redzone word appended to buffer
157          * (can be shared) */
158         cache->alloc_size = cache->obj_size + sizeof(uint32_t);
159     }
160     else if(ctor != NULL && ! (flags & SLAB_COMPACT)) {
161         /* If a constructor is defined, we cannot put the bufctl inside
162          * the object because that could overwrite constructed state,
163          * unless client explicitly says it's ok (SLAB_COMPACT flag). */
164         cache->alloc_size = cache->obj_size + sizeof(slab_bufctl_t);
165     }

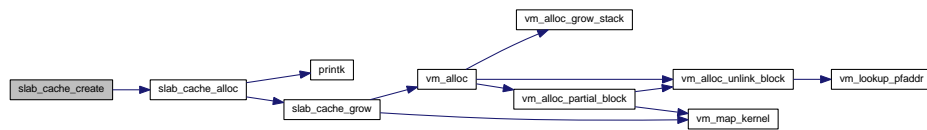
```

```

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
193
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         destroy_slab(cache, slab);
215     }
216 }

```

```

217     slab = next;
218     ++empty_count;
219 }
220
222     assert(cache->empty_count == empty_count);
223
224     /* free cache structure */
225     slab_cache_free(cache);
226 }

```

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         if(slab->next != NULL) {
378             slab->next->prev = slab->prev;
379         }
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

```

```

391     if(cache->flags & SLAB_RED_ZONE) {
392         rz_word = (uint32_t *) ( (char *)buffer + cache->obj_size );
393
394         if(*rz_word != SLAB_RED_ZONE_VALUE) {
395             printk("detected write past the end of object, cache: %s buffer: 0x%x value: 0x%x\n",
396                 cache->name,
397                 (unsigned int)buffer,
398                 *rz_word
399             );
400         }
401
402         *rz_word = SLAB_RED_ZONE_VALUE;
403     }
404
405     if(cache->flags & SLAB_POISON) {
406         if(cache->dtor != NULL) {
407             cache->dtor(buffer, cache->obj_size);
408         }
409
410         buffer32 = (uint32_t *)buffer;
411
412         for(idx = 0; idx < cache->obj_size / sizeof(uint32_t); ++idx) {
413             buffer32[idx] = SLAB_POISON_DEAD_VALUE;
414         }
415     }
416
417     /* link buffer into slab free list */
418     bufctl->next = slab->free_list;
419     slab->free_list = bufctl;
420     slab->obj_count -= 1;
421
422     /* If we just returned the last object to the slab, move the slab to
423      * the empty list. */
424     if(slab->obj_count == 0) {
425         /* remove from partial slabs list */
426         if(cache->slabs_partial == slab) {
427             cache->slabs_partial = slab->next;
428         }
429         else {
430             slab->prev->next = slab->next;
431         }
432
433         if(slab->next != NULL) {
434             slab->next->prev = slab->prev;
435         }
436
437         /* add to empty slabs list */
438         slab->next = cache->slabs_empty;
439         cache->slabs_empty = slab;
440
441         if(slab->next != NULL) {
442             slab->next->prev = slab;
443         }
444
445         ++(cache->empty_count);
446     }
447 }

```

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().

```

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
461     assert(slab_addr != NULL);
462
463     vm_map_kernel(slab_addr, pfalloc(), VM_FLAG_READ_WRITE);
464
465     slab = (slab_t *) ( (char *)slab_addr + SLAB_SIZE - sizeof(slab_t) );
466
467     slab->cache = cache;
468
469     /* slab is initially empty */
470     slab->obj_count = 0;
471
472     slab->next      = cache->slabs_empty;
473     cache->slabs_empty = slab;
474
475     if(slab->next != NULL) {
476         slab->next->prev = slab;
477     }
478
479     ++(cache->empty_count);
480
481     /* set slab colour and update cache next colour */
482     slab->colour = cache->next_colour;
483
484     if(cache->next_colour < cache->max_colour) {
485         cache->next_colour += cache->alignment;
486     }
487     else {
488         cache->next_colour = 0;
489     }
490
491     /* compute address of first bufctl */
492     bufctl = (slab_bufctl_t *) ( (char *)slab_addr + slab->colour + cache->
493     bufctl_offset );
494     slab->free_list = bufctl;
495
496     while(1) {
497         buffer = (addr_t)bufctl - cache->bufctl_offset;
498
499         if(cache->flags & SLAB_POISON) {
500             buffer_end = (uint32_t *) (buffer + cache->obj_size);
501
502             for(ptr = (uint32_t *)buffer; ptr < buffer_end; ++ptr) {
503                 *ptr = SLAB_POISON_DEAD_VALUE;
504             }
505
506             /* If both SLAB_POISON and SLAB_RED_ZONE are enabled, we
507              * perform redzone checking even on freed objects. */
508             if(cache->flags & SLAB_RED_ZONE) {
509                 *ptr = SLAB_RED_ZONE_VALUE;
510             }
511         }
512         else if (cache->ctor != NULL) {
513             cache->ctor((void *)buffer, cache->obj_size);
514         }
515
516         next = (slab_bufctl_t *) ( (char *)bufctl + cache->alloc_size );
517     }

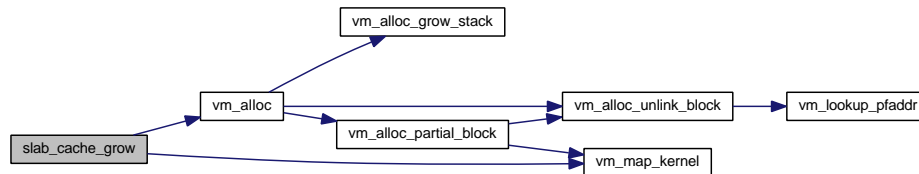
```

```

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.

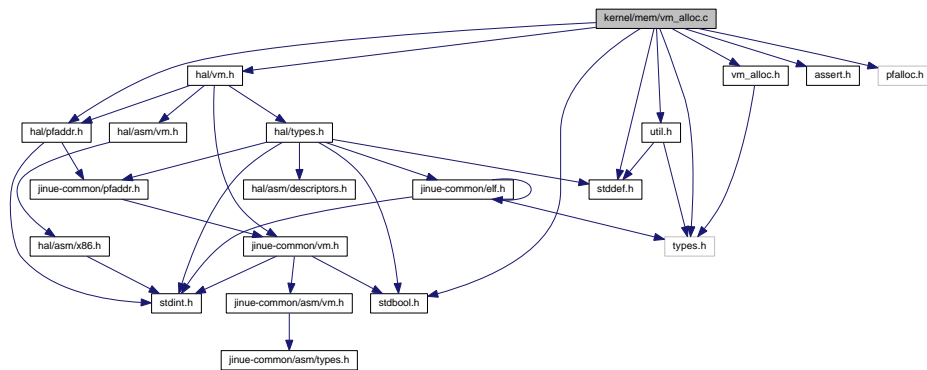
Referenced by slab_cache_create().

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

<i>allocator</i>	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;

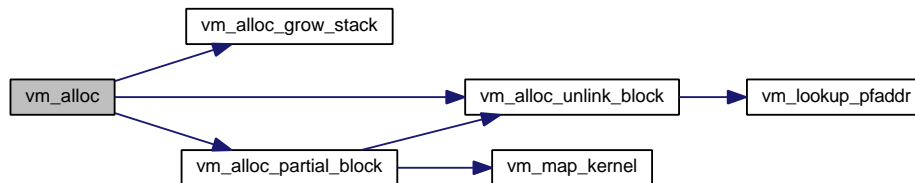
```

```

90
91  assert(allocator != NULL);
92
93  block = allocator->partial_list;
94
95  if(block == NULL) {
96      block = allocator->free_list;
97
98      if(block == NULL) {
99          return (addr_t) NULL;
100      }
101
102      vm_alloc_partial_block(block);
103  }
104
105  assert(block->stack_ptr != NULL);
106
107  /* if the page stack is empty, perform deferred page stack initialization */
108  if( VM_ALLOC_EMPTY_STACK(block) ) {
109      vm_alloc_grow_stack(block);
110  }
111
112  assert( ! VM_ALLOC_EMPTY_STACK(block) );
113
114  page = *(block->stack_ptr++);
115
116  /* if we just emptied the stack, mark the block as used */
117  if( VM_ALLOC_EMPTY_STACK(block) ) {
118      vm_alloc_unlink_block(block);
119  }
120
121  return page;
122 }

```

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

<i>allocator</i>	vm_alloc_t (p. 52) structure for a virtual memory allocator
<i>start_addr</i>	start address of the region
<i>end_addr</i>	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;

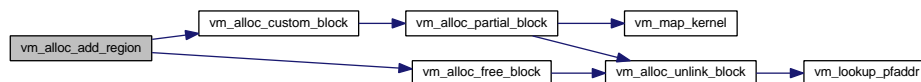
```

```

350     addr_t        limit;
351
352     /* skip the block array */
353     if(start_addr >= allocator->start_addr) {
354         start_addr_adjusted = start_addr;
355     }
356     else {
357         start_addr_adjusted = allocator->start_addr;
358     }
359
360     /* start and end block indices */
361     start = ((unsigned int)start_addr_adjusted - (unsigned int)allocator->
base_addr) / VM_ALLOC_BLOCK_SIZE;
362     end   = ((unsigned int)end_addr
base_addr) / VM_ALLOC_BLOCK_SIZE;
363
364     /* check and remember whether last block is partial (last_full < end) or
365      * completely free (last_full == end) */
366     if( OFFSET_OF(end_addr, VM_ALLOC_BLOCK_SIZE) == 0 ) {
367         end_full = end;
368     }
369     else {
370         end_full = end + 1;
371     }
372
373     /* array initialization -- first block (if partial) */
374     idx = start;
375
376     if( OFFSET_OF(start_addr_adjusted, VM_ALLOC_BLOCK_SIZE) != 0 ) {
377         limit = ALIGN_END(start_addr_adjusted, VM_ALLOC_BLOCK_SIZE);
378
379         if(end_addr < limit) {
380             limit = end_addr;
381         }
382
383         vm_alloc_custom_block(&allocator->block_array[idx], start_addr_adjusted, limit);
384
385         ++idx;
386     }
387
388     /* array initialization -- free blocks */
389     for(; idx < end; ++idx) {
390         vm_alloc_free_block(&allocator->block_array[idx]);
391     }
392
393     /* array initialization -- last block (if partial) */
394     if(idx < end_full) {
395         vm_alloc_custom_block(&allocator->block_array[idx], allocator->
block_array[idx].base_addr, end_addr);
396     }
397 }

```

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_PARTIAL`, `VM_ALLOC_IS_USED`, and `vm_alloc_partial_block()`.

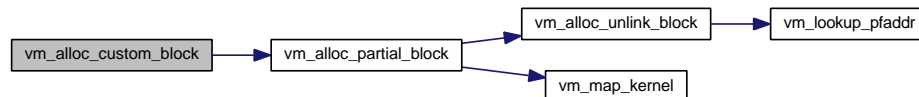
Referenced by `vm_alloc_add_region()`.

```

561
562 #ifndef NDEBUG
563     addr_t    limit;
564 #endif
565     addr_t    page;
566     addr_t    adjusted_start;
567
568     assert(block != NULL);
569
570     assert(page_offset_of(start_addr) == 0 && page_offset_of(end_addr) == 0);
571
572 #ifndef NDEBUG
573     limit = block->base_addr + VM_ALLOC_BLOCK_SIZE;
574 #endif
575
576     assert(start_addr >= block->base_addr && end_addr <= limit && start_addr < end_addr );
577
578     assert( ! VM_ALLOC_IS_FREE(block) );
579
580     adjusted_start = start_addr;
581
582     if( VM_ALLOC_IS_USED(block) ) {
583         /* if no stack address is specified at this point, use the first page
584          * of the address range for this purpose */
585         if( block->stack_addr == NULL ) {
586             block->stack_addr = (addr_t *)start_addr;
587             adjusted_start    = start_addr + PAGE_SIZE;
588
589             /* if the address range contained only a single page, there is
590              * nothing left to do here */
591             if(adjusted_start >= end_addr) {
592                 return;
593             }
594         }
595     }
596
597     vm_alloc_partial_block(block);
598
599     assert( VM_ALLOC_IS_PARTIAL(block) );
600
601     /* initialize stack */
602     page = adjusted_start;
603     while(page < end_addr) {
604         assert( ! VM_ALLOC_FULL_STACK(block) );
605
606         *(--block->stack_ptr) = page;
607         page += PAGE_SIZE;
608     }
609 }

```

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()`.

```

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:



4.101.2.5 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

<i>block</i>	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
411     assert(block != NULL);
412
413     /* unlink from partial list if necessary */
414     vm_alloc_unlink_block(block);

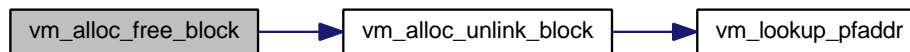
```

```

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

<i>block</i>	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
737     assert(block != NULL);
738
739     assert(block->next != NULL && block->prev != NULL);
740
741     assert(block->stack_ptr != NULL);
742
743 }

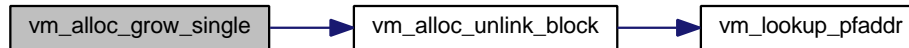
```

```

747     assert( ! VM_ALLOC_CANNOT_GROW(block) );
748
749     page          = block->stack_next;
750     block->stack_next = page + PAGE_SIZE;
751
752     if( VM_ALLOC_CANNOT_GROW(block) ) {
753         /* block is now used up, remove it from the partial list */
754         vm_alloc_unlink_block(block);
755     }
756
757     return page;
758 }

```

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

<i>block</i>	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
700     assert(block != NULL);
701
702     assert(block->next != NULL && block->prev != NULL);
703
704     assert(block->stack_ptr != NULL);
705
706     stack_ptr = block->stack_ptr;
707     page      = block->stack_next;
708     limit     = block->base_addr + VM_ALLOC_BLOCK_SIZE;
709
710     while(page < limit) {
711         assert( ! VM_ALLOC_FULL_STACK(block) );
712
713         *(--stack_ptr) = page;
714
715         page += PAGE_SIZE;
716     }
717
718     block->stack_ptr = stack_ptr;
719     block->stack_next = limit;
720 }
721
722
723
724 }

```


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

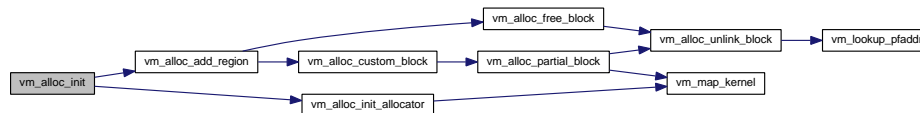
Definition at line 213 of file vm_alloc.c.

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

<i>allocator</i>	vm_alloc_t (p. 52) structure for a virtual memory allocator
<i>start_addr</i>	start address of the region managed by the allocator
<i>size</i>	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
269     assert(allocator != NULL);
270
271     assert( page_offset_of(start_addr) == 0 && page_offset_of(end_addr) == 0 );
272
273     /* align base and end addresses to block size */
```

```

277     base_addr  = (addr_t)ALIGN_START(start_addr, VM_ALLOC_BLOCK_SIZE);
278     aligned_end = (addr_t)ALIGN_END(end_addr, VM_ALLOC_BLOCK_SIZE);
279
280     /* calculate number of memory blocks managed by this allocator */
281     block_count = ( (char *)aligned_end - (char *)base_addr ) /
VM_ALLOC_BLOCK_SIZE;
282
283     /* calculate the number of pages required to store the memory block
284      * descriptor array */
285     array_size = block_count * sizeof(vm_block_t);
286     array_page_count = array_size / PAGE_SIZE;
287     if(array_size % PAGE_SIZE != 0) {
288         ++array_page_count;
289     }
290
291     /* address of the block array */
292     block_array = (vm_block_t *)start_addr;
293
294     /* adjust base address to skip block descriptor array */
295     adjusted_start = start_addr + array_page_count * PAGE_SIZE;
296
297     /* initialize allocator struct */
298     allocator->start_addr  = adjusted_start;
299     allocator->end_addr    = end_addr;
300     allocator->base_addr   = base_addr;
301     allocator->block_count = block_count;
302     allocator->block_array = block_array;
303     allocator->array_pages = array_page_count;
304     allocator->free_list   = NULL;
305     allocator->partial_list = NULL;
306
307     /* allocate block descriptor array pages */
308     addr = (addr_t)block_array;
309     for(idx = 0; idx < array_page_count; ++idx) {
310         /* allocate and map page */
311         paddr = pfalloc();
312         vm_map_kernel(addr, paddr, VM_FLAG_READ_WRITE);
313
314         /* calculate address of next page */
315         addr += PAGE_SIZE;
316     }
317
319     assert(addr == adjusted_start);
320
321     /* basic initialization of array (all blocks unlinked/used) */
322     addr = base_addr;
323     for(idx = 0; idx < block_count; ++idx) {
324         block_array[idx].base_addr = addr;
325         block_array[idx].allocator = allocator;
326
327         /* mark block as unlinked for now */
328         block_array[idx].next = NULL;
329
330         /* a null stack base indicates the block is uninitialized */
331         block_array[idx].stack_addr = NULL;
332
333         /* calculate address of next block */
334         addr += VM_ALLOC_BLOCK_SIZE;
335     }
336 }

```

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

<i>allocator</i>	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.

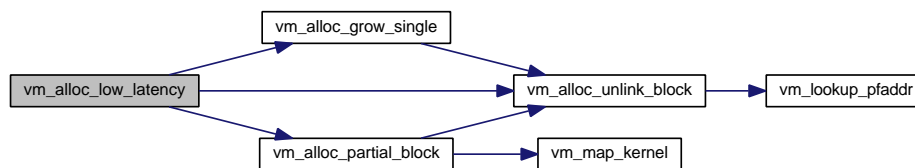
References `assert`, `vm_alloc_t::free_list`, `NULL`, `vm_alloc_t::partial_list`, `vm_block_t::stack_ptr`, `VM_ALLOC_EMPTY_STACK`, `vm_alloc_grow_single()`, `vm_alloc_partial_block()`, and `vm_alloc_unlink_block()`.

```

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

<i>block</i>	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     paddr_t         paddr;
459     bool            was_free;
460
461
462     assert(block != NULL);
463
464     /* To keep in mind...
465      *
466      * When the allocator is initialized, some blocks may be created partial
467      * (typical for the first and the last block of the region). If there is a
468      * hole at the start of the block, the page stack will be at the first
469      * available page, not at the start of the block. Since these blocks have
470      * holes, they will never be in the free state.
471      *
472      * So, when a block is free on function entry, we ensure the stack is placed
473      * at the start of the block so that all the remaining pages can be
474      * allocated sequentially (see deferred stack initialization below). However,
475      * if the block is in the used state on function entry, we leave the stack
476      * at its previous location since the first page of the block might not be
477      * available. */
478
479     if(block->next == NULL) {
480         assert(block->stack_addr != NULL);
481
482         /* block was used on function entry */
483         was_free = false;
484     }
485     else {
486         if(block->stack_ptr != NULL) {
487             /* block is already partial, leave it untouched */
488             return;
489         }
490
491         /* block was free on function entry */
492         was_free = true;
493
494         /* unlink from free list */
495         vm_alloc_unlink_block(block);
496
497         /* use first page of block for the stack */
498         block->stack_addr = (addr_t *)block->base_addr;
499     }
500
501     /* allocate the page stack */
502     stack_addr = block->stack_addr;
503     paddr      = pfalloc();
504     vm_map_kernel((addr_t)stack_addr, paddr, VM_FLAG_READ_WRITE);
505
506     assert(block->allocator != NULL);
507
508     /* link block to the partial list */
509     if(block->allocator->partial_list == NULL) {
510         /* special case: partial list is empty */
511         block->allocator->partial_list = block;
512
513         block->next = block;

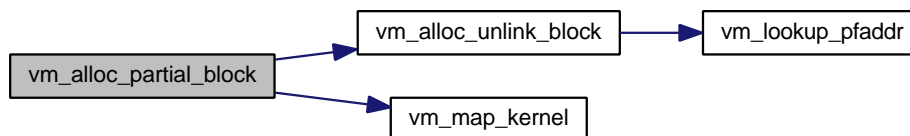
```

```

517     block->prev = block;
518 }
519 else {
520     /* block will be at to the end of the partial block list */
521     next = block->allocator->partial_list;
522     prev = next->prev;
523
524     /* re-link block */
525     block->prev = prev;
526     block->next = next;
527
528     prev->next = block;
529     next->prev = block;
530 }
531
532 /* Ok, here's the deal (deferred stack intialization)...
533  *
534  * We do not want to initialize the page stack right now because this is
535  * a time consuming operation, and we might be in time-critical code
536  * (interrupt handling code for example). Instead, the stack initialization
537  * is deferred until the next page allocations. The first non-time critical
538  * allocation which encounters an empty stack will initialize the whole
539  * stack. In the meantime, time critical ones will just allocate pages
540  * sequentially from the start of the block.
541  *
542  * The stack_next pointer in the vm_block_t structure points to the next
543  * page available for sequential allocation. The memory block is actually
544  * used up (no more pages available) when the page stack is empty AND the
545  * stack_next pointer has reached the end of the block. */
546
547 /* initialize the stack as empty */
548 block->stack_ptr = (addr_t *) ( (char *)stack_addr + PAGE_SIZE );
549
550 if(was_free) {
551     /* free block: we skip the first page as it was allocated for the
552      * stack itself */
553     block->stack_next = block->base_addr + PAGE_SIZE;
554 }
555 else {
556     /* used block: sequential allocation no longer possible */
557     block->stack_next = block->base_addr + VM_ALLOC_BLOCK_SIZE;
558 }
559 }

```

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

<i>block</i>	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

<i>allocator</i>	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()`.

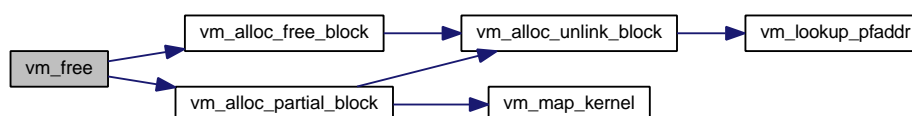
Referenced by `elf_load()`, `elf_setup_stack()`, `thread_page_create()`, and `thread_page_destroy()`.

```

172                                     {
173     vm_block_t    *block;
174     unsigned int  idx;
175
176     assert(allocator != NULL);
177
178     assert(page >= allocator->start_addr && page < allocator->end_addr);
179
180     assert(page_offset_of(page) == 0);
181
182     /* find the block to which the free page belong */
183     idx = ( (unsigned int)page - (unsigned int)allocator->base_addr ) /
184     VM_ALLOC_BLOCK_SIZE;
185     block = &allocator->block_array[idx];
186
187     /* if the block was a used block, make it a partial block */
188     if( VM_ALLOC_IS_USED(block) ) {
189         if(block->stack_addr == NULL) {
190             block->stack_addr = (addr_t *)page;
191             return;
192         }
193     }
194
195     vm_alloc_partial_block(block);
196
197     assert( VM_ALLOC_IS_PARTIAL(block) );
198
199     assert( ! VM_ALLOC_FULL_STACK(block) );
200
201     *(--block->stack_ptr) = page;
202
203     /* check if we just freed the whole block */
204     if( VM_ALLOC_FULL_STACK(block) ) {
205         vm_alloc_free_block(block);
206     }
207 }
208
209
210
211

```

Here is the call graph for this function:



Index

- __assert_failed
 - assert.c, 346
 - assert.h, 224
- __bool_true_false_are_defined
 - stdbool.h, 226
- a_type
 - Elf32_auxv_t, 15
- a_un
 - Elf32_auxv_t, 15
- a_val
 - Elf32_auxv_t, 15
- ADDR_4GB
 - hal/vm.h, 121
- ADDR_TO_PFADDR
 - hal/pfaddr.h, 164
- ALIGN_END
 - util.h, 249
- ALIGN_START
 - util.h, 249
- AT_BASE
 - jinue-common/elf.h, 73
- AT_DCACHEBSIZE
 - jinue-common/elf.h, 73
- AT_ENTRY
 - jinue-common/elf.h, 73
- AT_EXECFD
 - jinue-common/elf.h, 73
- AT_FLAGS
 - jinue-common/elf.h, 73
- AT_HWCAP
 - jinue-common/elf.h, 73
- AT_HWCAP2
 - jinue-common/elf.h, 73
- AT_ICACHEBSIZE
 - jinue-common/elf.h, 73
- AT_IGNORE
 - jinue-common/elf.h, 73
- AT_NULL
 - jinue-common/elf.h, 74
- AT_PAGESZ
 - jinue-common/elf.h, 74
- AT_PHDR
 - jinue-common/elf.h, 74
- AT_PHENT
 - jinue-common/elf.h, 74

- AT_PHNUM
 - jinue-common/elf.h, 74
- AT_STACKBASE
 - jinue-common/elf.h, 74
- AT_SYSINFO_EHDR
 - jinue-common/elf.h, 74
- AT_UCACHEBSIZE
 - jinue-common/elf.h, 75
- addr
 - bootmem_t, 11
 - e820_t, 14
 - elf_symbol_t, 24
 - memory_block_t, 31
 - pseudo_descriptor_t, 37
- addr_space
 - elf_info_t, 23
 - process_t, 36
- addr_space_t, 7
 - cr3, 8
 - pd, 8
 - pdpt, 8
 - top_level, 8
- addr_t
 - hal/types.h, 170
- alignment
 - slab_cache_t, 40
- alloc_backward
 - util.h, 249
- alloc_forward
 - util.h, 249
- alloc_size
 - slab_cache_t, 40
- allocator
 - vm_block_t, 55
- any_key
 - kbd.c, 301
 - kbd.h, 222
- apply_mem_hole
 - bootmem.c, 307
 - bootmem.h, 140
- arg0
 - jinue_syscall_args_t, 29
- arg1
 - jinue_syscall_args_t, 29
- arg2

- jinue_syscall_args_t, 29
- arg3
 - jinue_syscall_args_t, 29
- array_pages
 - vm_alloc_t, 53
- ascii.h
 - CHAR_BS, 59
 - CHAR_CR, 59
 - CHAR_HT, 59
 - CHAR_LF, 60
- asm/boot.h
 - BOOT_DATA_STRUCT, 91
 - BOOT_E820_ENTRIES, 91
 - BOOT_E820_MAP, 91
 - BOOT_E820_MAP_END, 91
 - BOOT_E820_MAP_SIZE, 91
 - BOOT_MAGIC, 91
 - BOOT_SETUP, 91
 - BOOT_SETUP32_ADDR, 91
 - BOOT_SETUP32_SIZE, 91
 - BOOT_SETUP_ADDR, 91
 - BOOT_SETUP_HEADER, 91
 - BOOT_SETUP_MAGIC, 91
 - BOOT_SETUP_SECTS, 92
 - BOOT_SIGNATURE, 92
 - BOOT_STACK_SIZE, 92
 - BOOT_SYSIZE, 92
- asm/descriptors.h
 - GDT_KERNEL_CODE, 97
 - GDT_KERNEL_DATA, 97
 - GDT_LENGTH, 97
 - GDT_NULL, 97
 - GDT_PER_CPU_DATA, 97
 - GDT_TSS, 97
 - GDT_USER_CODE, 98
 - GDT_USER_DATA, 98
 - GDT_USER_TLS_DATA, 98
 - RPL_KERNEL, 98
 - RPL_USER, 98
 - SEG_FLAG_16BIT, 98
 - SEG_FLAG_16BIT_GATE, 98
 - SEG_FLAG_32BIT, 98
 - SEG_FLAG_32BIT_GATE, 99
 - SEG_FLAG_BUSY, 99
 - SEG_FLAG_IN_BYTES, 99
 - SEG_FLAG_IN_PAGES, 99
 - SEG_FLAG_KERNEL, 99
 - SEG_FLAG_NORMAL, 99
 - SEG_FLAG_NORMAL_GATE, 99
 - SEG_FLAG_NOSYSTEM, 99
 - SEG_FLAG_PRESENT, 100
 - SEG_FLAG_SYSTEM, 100
 - SEG_FLAG_TSS, 100
 - SEG_FLAG_USER, 100
 - SEG_FLAGS_OFFSET, 100
 - SEG_SELECTOR, 100
 - SEG_TYPE_CALL_GATE, 100
 - SEG_TYPE_CODE, 101
 - SEG_TYPE_DATA, 101
 - SEG_TYPE_INTERRUPT_GATE, 101
 - SEG_TYPE_READ_ONLY, 101
 - SEG_TYPE_TASK_GATE, 101
 - SEG_TYPE_TRAP_GATE, 101
 - SEG_TYPE_TSS, 101
 - TSS_LIMIT, 101
- asm/e820.h
 - E820_ACPI, 104
 - E820_RAM, 104
 - E820_RESERVED, 104
 - E820_SMAP, 104
- asm/x86.h
 - X86_CR0_PG, 132
 - X86_CR0_WP, 132
 - X86_CR4_PAE, 133
 - X86_CR4_PGE, 133
 - X86_CR4_PSE, 133
 - X86_PDE_PAGE_SIZE, 133
 - X86_PTE_ACCESSED, 133
 - X86_PTE_CACHE_DISABLE, 133
 - X86_PTE_DIRTY, 133
 - X86_PTE_GLOBAL, 133
 - X86_PTE_NX, 134
 - X86_PTE_PRESENT, 134
 - X86_PTE_READ_WRITE, 134
 - X86_PTE_USER, 134
 - X86_PTE_WRITE_THROUGH, 134
- assert
 - assert.h, 223
- assert.c
 - __assert_failed, 346
- assert.h
 - __assert_failed, 224
 - assert, 223
- at_phdr
 - elf_info_t, 23
- at_phent
 - elf_info_t, 23
- at_phnum
 - elf_info_t, 23
- auxv_t
 - jinue-common/elf.h, 83
- BOOT_DATA_STRUCT
 - asm/boot.h, 91
- BOOT_E820_ENTRIES
 - asm/boot.h, 91
- BOOT_E820_MAP
 - asm/boot.h, 91

BOOT_E820_MAP_END
 asm/boot.h, 91
 BOOT_E820_MAP_SIZE
 asm/boot.h, 91
 BOOT_MAGIC
 asm/boot.h, 91
 BOOT_SETUP
 asm/boot.h, 91
 BOOT_SETUP32_ADDR
 asm/boot.h, 91
 BOOT_SETUP32_SIZE
 asm/boot.h, 91
 BOOT_SETUP_ADDR
 asm/boot.h, 91
 BOOT_SETUP_HEADER
 asm/boot.h, 91
 BOOT_SETUP_MAGIC
 asm/boot.h, 91
 BOOT_SETUP_SECTS
 asm/boot.h, 92
 BOOT_SIGNATURE
 asm/boot.h, 92
 BOOT_STACK_SIZE
 asm/boot.h, 92
 BOOT_SYSIZE
 asm/boot.h, 92
 base_addr
 vm_alloc_t, 53
 vm_block_t, 55
 block_array
 vm_alloc_t, 53
 block_count
 vm_alloc_t, 53
 bool
 stdbool.h, 226
 boot.c
 boot_info, 306
 boot_info_check, 304
 boot_info_dump, 305
 get_boot_info, 306
 boot.h
 boot_info_check, 93
 boot_info_dump, 94
 get_boot_info, 95
 boot_end
 boot_info_t, 9
 boot_heap
 boot_info_t, 9
 bootmem.c, 311
 bootmem.h, 144
 boot_info
 boot.c, 306
 boot_info_check
 boot.c, 304
 boot.h, 93
 boot_info_dump
 boot.c, 305
 boot.h, 94
 boot_info_t, 8
 boot_end, 9
 boot_heap, 9
 e820_entries, 9
 e820_map, 9
 image_start, 9
 image_top, 9
 kernel_size, 10
 kernel_start, 10
 page_directory, 10
 page_table, 10
 proc_size, 10
 proc_start, 10
 setup_signature, 10
 bootmem.c
 apply_mem_hole, 307
 boot_heap, 311
 bootmem_get_block, 308
 bootmem_init, 309
 bootmem_root, 312
 new_ram_map_entry, 311
 ram_map, 312
 bootmem.h
 apply_mem_hole, 140
 boot_heap, 144
 bootmem_get_block, 141
 bootmem_init, 141
 bootmem_root, 144
 bootmem_t, 140
 new_ram_map_entry, 143
 ram_map, 144
 bootmem_get_block
 bootmem.c, 308
 bootmem.h, 141
 bootmem_init
 bootmem.c, 309
 bootmem.h, 141
 bootmem_root
 bootmem.c, 312
 bootmem.h, 144
 bootmem_t, 11
 addr, 11
 bootmem.h, 140
 count, 11
 next, 11
 bufctl_offset
 slab_cache_t, 40
 buffer_size
 jinue_message_t, 27
 message_info_t, 32

- CHAR_BS
 - ascii.h, 59
- CHAR_CR
 - ascii.h, 59
- CHAR_HT
 - ascii.h, 59
- CHAR_LF
 - ascii.h, 60
- CONSOLE_SERIAL_BAUD_RATE
 - jinue-common/console.h, 61
- CONSOLE_SERIAL_IOPORT
 - jinue-common/console.h, 61
- CPU_DATA_ALIGNMENT
 - cpu_data.h, 153
- CPU_EFLAGS_ID
 - cpu.h, 146
- CPU_FEATURE_CPUID
 - cpu.h, 146
- CPU_FEATURE_LOCAL_APIC
 - cpu.h, 146
- CPU_FEATURE_PAE
 - cpu.h, 146
- CPU_FEATURE_SYSCALL
 - cpu.h, 146
- CPU_FEATURE_SYSENTER
 - cpu.h, 147
- CPU_VENDOR_AMD
 - cpu.h, 147
- CPU_VENDOR_AMD_DW0
 - cpu.h, 147
- CPU_VENDOR_AMD_DW1
 - cpu.h, 147
- CPU_VENDOR_AMD_DW2
 - cpu.h, 147
- CPU_VENDOR_GENERIC
 - cpu.h, 147
- CPU_VENDOR_INTEL
 - cpu.h, 147
- CPU_VENDOR_INTEL_DW0
 - cpu.h, 147
- CPU_VENDOR_INTEL_DW1
 - cpu.h, 147
- CPU_VENDOR_INTEL_DW2
 - cpu.h, 148
- CPUID_EXT_FEATURE_SYSCALL
 - cpu.h, 148
- CPUID_FEATURE_APIC
 - cpu.h, 148
- CPUID_FEATURE_CLFLUSH
 - cpu.h, 148
- CPUID_FEATURE_FPU
 - cpu.h, 148
- CPUID_FEATURE_HTTP
 - cpu.h, 148
- CPUID_FEATURE_PAE
 - cpu.h, 148
- CPUID_FEATURE_SEP
 - cpu.h, 148
- cache
 - slab_t, 42
- clear_pte
 - vm.c, 340
 - vm_private.h, 189
- cli
 - x86.h, 136
- colour
 - slab_t, 43
- common/errno.h
 - JINUE_E2BIG, 204
 - JINUE_EAGAIN, 204
 - JINUE_EBADF, 204
 - JINUE_EINVAL, 204
 - JINUE_EIO, 204
 - JINUE_EMORE, 204
 - JINUE_ENOMEM, 204
 - JINUE_ENOSYS, 204
 - JINUE_EPERM, 204
- common/list.h
 - JINUE_LIST_STATIC, 206
 - JINUE_OFFSETOF, 206
 - jinue_cursor_entry, 206
 - jinue_cursor_t, 207
 - jinue_list_pop, 206
 - jinue_node_entry, 206
 - jinue_node_t, 207
- console.c
 - console_init, 296
 - console_print, 297
 - console_printn, 297
 - console_putc, 297
- console_init
 - console.c, 296
 - jinue-common/console.h, 61
- console_print
 - console.c, 297
 - jinue-common/console.h, 61
- console_printn
 - console.c, 297
 - jinue-common/console.h, 62
- console_putc
 - console.c, 297
 - jinue-common/console.h, 62
- cookie
 - jinue_message_t, 27
 - message_info_t, 32
 - object_ref_t, 34
- copy_pte
 - vm.c, 340

- vm_private.h, 189
- core.c
 - kmain, 266
- core.h
 - kmain, 63
- core/thread.c
 - thread_create, 290
 - thread_ready, 291
 - thread_switch, 291
 - thread_yield_from, 292
- count
 - bootmem_t, 11
 - memory_block_t, 31
 - pfcache_t, 35
- cpu.c
 - cpu_detect_features, 313
 - cpu_info, 316
 - cpu_init_data, 315
- cpu.h
 - CPU_EFLAGS_ID, 146
 - CPU_FEATURE_CPUID, 146
 - CPU_FEATURE_LOCAL_APIC, 146
 - CPU_FEATURE_PAE, 146
 - CPU_FEATURE_SYSCALL, 146
 - CPU_FEATURE_SYSENTER, 147
 - CPU_VENDOR_AMD, 147
 - CPU_VENDOR_AMD_DW0, 147
 - CPU_VENDOR_AMD_DW1, 147
 - CPU_VENDOR_AMD_DW2, 147
 - CPU_VENDOR_GENERIC, 147
 - CPU_VENDOR_INTEL, 147
 - CPU_VENDOR_INTEL_DW0, 147
 - CPU_VENDOR_INTEL_DW1, 147
 - CPU_VENDOR_INTEL_DW2, 148
 - CPUID_EXT_FEATURE_SYSCALL, 148
 - CPUID_FEATURE_APIC, 148
 - CPUID_FEATURE_CLFLUSH, 148
 - CPUID_FEATURE_FPU, 148
 - CPUID_FEATURE_HTT, 148
 - CPUID_FEATURE_PAE, 148
 - CPUID_FEATURE_SEP, 148
 - cpu_detect_features, 149
 - cpu_info, 152
 - cpu_init_data, 151
 - MSR_EFER, 148
 - MSR_FLAG_STAR_SCE, 149
 - MSR_IA32_SYSENTER_CS, 149
 - MSR_IA32_SYSENTER_EIP, 149
 - MSR_IA32_SYSENTER_ESP, 149
 - MSR_STAR, 149
- cpu_data.h
 - CPU_DATA_ALIGNMENT, 153
- cpu_data_t, 11
 - current_addr_space, 12
- gdt, 12
- hal/types.h, 170
- self, 12
- tss, 13
- cpu_detect_features
 - cpu.c, 313
 - cpu.h, 149
- cpu_info
 - cpu.c, 316
 - cpu.h, 152
- cpu_info_t, 13
 - dcache_alignment, 13
 - family, 13
 - features, 13
 - model, 14
 - stepping, 14
 - vendor, 14
- cpu_init_data
 - cpu.c, 315
 - cpu.h, 151
- cpuid
 - x86.h, 136
- cr3
 - addr_space_t, 8
 - tss_t, 49
- create_addr_space
 - vm.c, 340
 - vm_private.h, 190
- create_initial_addr_space
 - vm.c, 341
 - vm_private.h, 190
- cs
 - trapframe_t, 47
 - tss_t, 49
- ctor
 - slab_cache_t, 40
- current_addr_space
 - cpu_data_t, 12
- data_size
 - jinue_message_t, 27
 - jinue_reply_t, 28
 - message_info_t, 32
- dcache_alignment
 - cpu_info_t, 13
- debug
 - tss_t, 49
- debug.c
 - dump_call_stack, 299
- debug.h
 - dump_call_stack, 66
- desc_n
 - jinue_message_t, 27
 - jinue_reply_t, 28

- message_info_t, 32
- descriptors
 - process_t, 36
- descriptors.h
 - GATE_DESCRIPTOR, 103
 - PACK_DESCRIPTOR, 103
 - SEG_DESCRIPTOR, 103
- destroy_addr_space
 - vm.c, 341
 - vm_private.h, 190
- dispatch_interrupt
 - interrupt.c, 323
 - interrupt.h, 159
- dispatch_syscall
 - syscall.c, 286
 - syscall.h, 219
- ds
 - trapframe_t, 47
 - tss_t, 50
- dtor
 - slab_cache_t, 40
- dump_call_stack
 - debug.c, 299
 - debug.h, 66
- e820.c
 - e820_dump, 317
 - e820_is_available, 318
 - e820_is_valid, 318
 - e820_type_description, 318
- e820.h
 - e820_dump, 105
 - e820_is_available, 106
 - e820_is_valid, 106
 - e820_type_description, 106
- E820 ACPI
 - asm/e820.h, 104
- E820_RAM
 - asm/e820.h, 104
- E820_RESERVED
 - asm/e820.h, 104
- E820_SMAP
 - asm/e820.h, 104
- e820_addr_t
 - hal/types.h, 170
- e820_dump
 - e820.c, 317
 - e820.h, 105
- e820_entries
 - boot_info_t, 9
- e820_is_available
 - e820.c, 318
 - e820.h, 106
- e820_is_valid
 - e820.c, 318
 - e820.h, 106
- e820_map
 - boot_info_t, 9
- e820_size_t
 - hal/types.h, 171
- e820_t, 14
 - addr, 14
 - size, 14
 - type, 15
- e820_type_description
 - e820.c, 318
 - e820.h, 106
- e820_type_t
 - hal/types.h, 171
- e_ehsize
 - Elf32_Ehdr, 16
- e_entry
 - Elf32_Ehdr, 16
- e_flags
 - Elf32_Ehdr, 16
- e_ident
 - Elf32_Ehdr, 16
- e_machine
 - Elf32_Ehdr, 17
- e_phentsize
 - Elf32_Ehdr, 17
- e_phnum
 - Elf32_Ehdr, 17
- e_phoff
 - Elf32_Ehdr, 17
- e_shentsize
 - Elf32_Ehdr, 17
- e_shnum
 - Elf32_Ehdr, 17
- e_shoff
 - Elf32_Ehdr, 17
- e_shstrndx
 - Elf32_Ehdr, 17
- e_type
 - Elf32_Ehdr, 17
- e_version
 - Elf32_Ehdr, 18
- EARLY_PHYS_TO_VIRT
 - hal/vm.h, 121
- EARLY_PTR_TO_PFADDR
 - hal/vm.h, 121
- EARLY_VIRT_TO_PHYS
 - hal/vm.h, 121
- EI_CLASS
 - jinue-common/elf.h, 75
- EI_DATA
 - jinue-common/elf.h, 75
- EI_MAG0

- jinue-common/elf.h, 75
- EI_MAG1
 - jinue-common/elf.h, 75
- EI_MAG2
 - jinue-common/elf.h, 75
- EI_MAG3
 - jinue-common/elf.h, 75
- EI_NIDENT
 - jinue-common/elf.h, 76
- EI_PAD
 - jinue-common/elf.h, 76
- EI_VERSION
 - jinue-common/elf.h, 76
- ELF32_ST_BIND
 - jinue-common/elf.h, 76
- ELF32_ST_TYPE
 - jinue-common/elf.h, 76
- ELF_MAGIC0
 - jinue-common/elf.h, 76
- ELF_MAGIC1
 - jinue-common/elf.h, 76
- ELF_MAGIC2
 - jinue-common/elf.h, 76
- ELF_MAGIC3
 - jinue-common/elf.h, 77
- ELFCLASS32
 - jinue-common/elf.h, 77
- ELFCLASS64
 - jinue-common/elf.h, 77
- ELFCLASSNONE
 - jinue-common/elf.h, 77
- ELFDATA2LSB
 - jinue-common/elf.h, 77
- ELFDATA2MSB
 - jinue-common/elf.h, 77
- ELFDATANONE
 - jinue-common/elf.h, 77
- EM_386
 - jinue-common/elf.h, 77
- EM_AARCH64
 - jinue-common/elf.h, 78
- EM_ALTERA_NIOS2
 - jinue-common/elf.h, 78
- EM_ARM
 - jinue-common/elf.h, 78
- EM_MICROBLAZE
 - jinue-common/elf.h, 78
- EM_MIPS
 - jinue-common/elf.h, 78
- EM_NONE
 - jinue-common/elf.h, 78
- EM_OPENRISC
 - jinue-common/elf.h, 78
- EM_SPARC
 - jinue-common/elf.h, 78
- EM_SPARC32PLUS
 - jinue-common/elf.h, 79
- EM_X86_64
 - jinue-common/elf.h, 79
- ET_CORE
 - jinue-common/elf.h, 79
- ET_DYN
 - jinue-common/elf.h, 79
- ET_EXEC
 - jinue-common/elf.h, 79
- ET_NONE
 - jinue-common/elf.h, 79
- ET_REL
 - jinue-common/elf.h, 79
- EXCEPTION_ALIGNMENT
 - irq.h, 108
- EXCEPTION_BOUND
 - irq.h, 108
- EXCEPTION_BREAK
 - irq.h, 108
- EXCEPTION_DIV_ZERO
 - irq.h, 108
- EXCEPTION_DOUBLE_FAULT
 - irq.h, 109
- EXCEPTION_GENERAL_PROTECTION
 - irq.h, 109
- EXCEPTION_INVALID_OP
 - irq.h, 109
- EXCEPTION_INVALID_TSS
 - irq.h, 109
- EXCEPTION_MACHINE_CHECK
 - irq.h, 109
- EXCEPTION_MATH
 - irq.h, 109
- EXCEPTION_NMI
 - irq.h, 109
- EXCEPTION_NO_COPROC
 - irq.h, 109
- EXCEPTION_OVERFLOW
 - irq.h, 109
- EXCEPTION_PAGE_FAULT
 - irq.h, 110
- EXCEPTION_SEGMENT_NOT_PRESENT
 - irq.h, 110
- EXCEPTION_SIMD
 - irq.h, 110
- EXCEPTION_STACK_SEGMENT
 - irq.h, 110
- eax
 - trapframe_t, 47
 - tss_t, 50
 - x86_cpuid_regs_t, 57
- ebp

- kernel_context_t, 30
- trapframe_t, 47
- tss_t, 50
- ebx
 - kernel_context_t, 30
 - trapframe_t, 47
 - tss_t, 50
 - x86_cpuid_regs_t, 57
- ecx
 - trapframe_t, 47
 - tss_t, 50
 - x86_cpuid_regs_t, 57
- edi
 - kernel_context_t, 30
 - trapframe_t, 47
 - tss_t, 50
- edx
 - trapframe_t, 47
 - tss_t, 50
 - x86_cpuid_regs_t, 57
- eflags
 - trapframe_t, 47
 - tss_t, 50
- eip
 - kernel_context_t, 30
 - trapframe_t, 47
 - tss_t, 50
- elf.c
 - elf_check, 269
 - elf_load, 271
 - elf_lookup_symbol, 273
 - elf_setup_stack, 274
- Elf32_Addr
 - jinue-common/elf.h, 83
- Elf32_Ehdr, 16
 - e_ehsize, 16
 - e_entry, 16
 - e_flags, 16
 - e_ident, 16
 - e_machine, 17
 - e_phentsize, 17
 - e_phnum, 17
 - e_phoff, 17
 - e_shentsize, 17
 - e_shnum, 17
 - e_shoff, 17
 - e_shstrndx, 17
 - e_type, 17
 - e_version, 18
- Elf32_Half
 - jinue-common/elf.h, 83
- Elf32_Off
 - jinue-common/elf.h, 83
- Elf32_Phdr, 18
 - p_align, 18
 - p_filesz, 18
 - p_flags, 18
 - p_memsz, 18
 - p_offset, 19
 - p_paddr, 19
 - p_type, 19
 - p_vaddr, 19
- Elf32_Shdr, 19
 - sh_addr, 20
 - sh_addralign, 20
 - sh_entsize, 20
 - sh_flags, 20
 - sh_info, 20
 - sh_link, 20
 - sh_name, 20
 - sh_offset, 20
 - sh_size, 20
 - sh_type, 20
- Elf32_Sword
 - jinue-common/elf.h, 83
- Elf32_Sym, 21
 - st_info, 21
 - st_name, 21
 - st_other, 21
 - st_shndx, 21
 - st_size, 21
 - st_value, 22
- Elf32_Word
 - jinue-common/elf.h, 83
- Elf32_auxv_t, 15
 - a_type, 15
 - a_un, 15
 - a_val, 15
- elf_check
 - elf.c, 269
 - jinue-common/elf.h, 84
- elf_info_t, 22
 - addr_space, 23
 - at_phdr, 23
 - at_phent, 23
 - at_phnum, 23
 - entry, 23
 - stack_addr, 23
- elf_load
 - elf.c, 271
 - jinue-common/elf.h, 85
- elf_lookup_symbol
 - elf.c, 273
 - jinue-common/elf.h, 87
- elf_setup_stack
 - elf.c, 274
 - jinue-common/elf.h, 88
- elf_symbol_t, 24

- addr, 24
- name, 24
- empty_count
 - slab_cache_t, 40
- end_addr
 - vm_alloc_t, 53
- entry
 - elf_info_t, 23
 - pte_t, 38
- errcode
 - trapframe_t, 47
- es
 - trapframe_t, 48
 - tss_t, 50
- esi
 - kernel_context_t, 30
 - trapframe_t, 48
 - tss_t, 50
- esp
 - trapframe_t, 48
 - tss_t, 51
- esp0
 - tss_t, 51
- esp1
 - tss_t, 51
- esp2
 - tss_t, 51
- false
 - stdbool.h, 226
- family
 - cpu_info_t, 13
- fast_amd_entry
 - trap.h, 168
- fast_intel_entry
 - trap.h, 168
- features
 - cpu_info_t, 13
- flags
 - object_header_t, 33
 - object_ref_t, 34
 - slab_cache_t, 40
- frame_pointer.h
 - get_caller_fpointer, 155
 - get_fpointer, 155
 - get_program_counter, 155
 - get_ret_addr, 155
- free_list
 - slab_t, 43
 - vm_alloc_t, 54
- fs
 - trapframe_t, 48
 - tss_t, 51
- function
 - jinue_message_t, 27
 - message_info_t, 32
- GATE_DESCRIPTOR
 - descriptors.h, 103
- GDT_KERNEL_CODE
 - asm/descriptors.h, 97
- GDT_KERNEL_DATA
 - asm/descriptors.h, 97
- GDT_LENGTH
 - asm/descriptors.h, 97
- GDT_NULL
 - asm/descriptors.h, 97
- GDT_PER_CPU_DATA
 - asm/descriptors.h, 97
- GDT_TSS
 - asm/descriptors.h, 97
- GDT_USER_CODE
 - asm/descriptors.h, 98
- GDT_USER_DATA
 - asm/descriptors.h, 98
- GDT_USER_TLS_DATA
 - asm/descriptors.h, 98
- gdt
 - cpu_data_t, 12
- get_boot_info
 - boot.c, 306
 - boot.h, 95
- get_caller_fpointer
 - frame_pointer.h, 155
- get_cr0
 - x86.h, 136
- get_cr1
 - x86.h, 136
- get_cr2
 - x86.h, 136
- get_cr3
 - x86.h, 136
- get_cr4
 - x86.h, 136
- get_eflags
 - x86.h, 136
- get_esp
 - x86.h, 137
- get_fpointer
 - frame_pointer.h, 155
- get_gs_ptr
 - x86.h, 137
- get_program_counter
 - frame_pointer.h, 155
- get_pte_flags
 - vm.c, 341
 - vm_private.h, 190
- get_pte_pfaddr

- vm.c, 341
- vm_private.h, 190
- get_pte_with_offset
 - vm.c, 341
 - vm_private.h, 190
- get_ret_addr
 - frame_pointer.h, 155
- global_page_allocator
 - vm.c, 341
 - vm_alloc.h, 266
- global_page_tables
 - vm.c, 341
 - vm_private.h, 190
- global_pfcache
 - pfalloc.c, 351
 - pfalloc.h, 212
- gs
 - trapframe_t, 48
 - tss_t, 51
- HAS_ERRCODE
 - irq.h, 110
- hal.c
 - hal_init, 320
 - kernel_region_top, 323
 - syscall_method, 323
- hal.h
 - hal_init, 155
- hal/asm/thread.h
 - THREAD_CONTEXT_MASK, 111
 - THREAD_CONTEXT_SIZE, 111
- hal/asm/vm.h
 - VM_FLAG_ACCESSED, 119
 - VM_FLAG_DIRTY, 119
 - VM_FLAG_KERNEL, 119
 - VM_FLAG_PRESENT, 119
 - VM_FLAG_READ_ONLY, 119
 - VM_FLAG_READ_WRITE, 119
 - VM_FLAG_USER, 119
- hal/pfaddr.h
 - ADDR_TO_PFADDR, 164
 - PFADDR_CHECK, 164
 - PFADDR_CHECK_4GB, 164
 - PFADDR_TO_ADDR, 164
- hal/thread.c
 - thread_context_switch, 293
 - thread_context_switch_stack, 294
 - thread_page_create, 294
 - thread_page_destroy, 295
- hal/thread.h
 - thread_context_switch, 112
 - thread_page_create, 113
 - thread_page_destroy, 114
- hal/types.h
 - addr_t, 170
 - cpu_data_t, 170
 - e820_addr_t, 170
 - e820_size_t, 171
 - e820_type_t, 171
 - msg_arg0, 170
 - msg_arg1, 170
 - msg_arg2, 170
 - msg_arg3, 170
 - pdpt_t, 171
 - pte_t, 171
 - seg_descriptor_t, 171
 - seg_selector_t, 171
- hal/vm.h
 - ADDR_4GB, 121
 - EARLY_PHYS_TO_VIRT, 121
 - EARLY_PTR_TO_PFADDR, 121
 - EARLY_VIRT_TO_PHYS, 121
 - vm_boot_init, 122
 - vm_change_flags, 124
 - vm_create_addr_space, 124
 - vm_create_initial_addr_space, 125
 - vm_destroy_addr_space, 125
 - vm_lookup_pfaddr, 125
 - vm_map_early, 125
 - vm_map_kernel, 126
 - vm_map_user, 126
 - vm_switch_addr_space, 126
 - vm_unmap_kernel, 127
 - vm_unmap_user, 127
- hal_init
 - hal.c, 320
 - hal.h, 155
- halt
 - startup.h, 167
- head
 - jinue_list_t, 26
- header
 - ipc_t, 25
 - process_t, 36
 - thread_t, 45
- IDT_FIRST_IRQ
 - irq.h, 110
- IDT_IRQ_COUNT
 - irq.h, 110
- IDT_VECTOR_COUNT
 - irq.h, 110
- INT64_C
 - stdint.h, 228
- IPC_FLAG_NONE
 - jinue-common/ipc.h, 196
- IPC_FLAG_SYSTEM
 - jinue-common/ipc.h, 196

- idt
 - interrupt.h, 160
- image_start
 - boot_info_t, 9
- image_top
 - boot_info_t, 9
- inb
 - io.h, 161
- include/ascii.h, 59
- include/console.h, 60
- include/core.h, 63
- include/debug.h, 66
- include/elf.h, 67
- include/hal/asm/boot.h, 90
- include/hal/asm/descriptors.h, 95
- include/hal/asm/e820.h, 103
- include/hal/asm/irq.h, 107
- include/hal/asm/thread.h, 111
- include/hal/asm/vm.h, 118
- include/hal/asm/x86.h, 131
- include/hal/boot.h, 92
- include/hal/bootmem.h, 139
- include/hal/cpu.h, 144
- include/hal/cpu_data.h, 153
- include/hal/descriptors.h, 102
- include/hal/e820.h, 104
- include/hal/frame_pointer.h, 154
- include/hal/hal.h, 155
- include/hal/interrupt.h, 158
- include/hal/io.h, 161
- include/hal/kernel.h, 162
- include/hal/pfaddr.h, 163
- include/hal/startup.h, 167
- include/hal/thread.h, 112
- include/hal/trap.h, 168
- include/hal/types.h, 169
- include/hal/vga.h, 174
- include/hal/vm.h, 120
- include/hal/vm_pae.h, 183
- include/hal/vm_private.h, 185
- include/hal/x86.h, 134
- include/ipc.h, 191
- include/jinue-common/asm/ipc.h, 192
- include/jinue-common/asm/syscall.h, 214
- include/jinue-common/asm/types.h, 172
- include/jinue-common/asm/vm.h, 128
- include/jinue-common/console.h, 60
- include/jinue-common/elf.h, 68
- include/jinue-common/errno.h, 203
- include/jinue-common/ipc.h, 195
- include/jinue-common/list.h, 205
- include/jinue-common/pfaddr.h, 165
- include/jinue-common/pfalloc.h, 208
- include/jinue-common/syscall.h, 218
- include/jinue-common/types.h, 173
- include/jinue-common/vm.h, 130
- include/jinue/console.h, 60
- include/jinue/elf.h, 67
- include/jinue/errno.h, 203
- include/jinue/ipc.h, 191
- include/jinue/list.h, 205
- include/jinue/pfaddr.h, 164
- include/jinue/pfalloc.h, 207
- include/jinue/syscall.h, 213
- include/jinue/types.h, 172
- include/jinue/vm.h, 127
- include/kbd.h, 222
- include/kstdc/assert.h, 223
- include/kstdc/stdarg.h, 224
- include/kstdc/stdbool.h, 225
- include/kstdc/stddef.h, 226
- include/kstdc/stdint.h, 227
- include/kstdc/string.h, 229
- include/object.h, 230
- include/panic.h, 232
- include/pfalloc.h, 209
- include/printk.h, 233
- include/process.h, 234
- include/slab.h, 237
- include/syscall.h, 218
- include/thread.h, 115
- include/types.h, 173
- include/util.h, 248
- include/vm_alloc.h, 249
- init_pfcache
 - pfalloc.c, 349
 - pfalloc.h, 210
- initial_addr_space
 - vm.c, 341
 - vm_private.h, 190
- initial_pdpt
 - vm_pae.c, 345
- inl
 - io.h, 161
- int16_t
 - stdint.h, 228
- int32_t
 - stdint.h, 228
- int64_t
 - stdint.h, 228
- int8_t
 - stdint.h, 228
- interrupt.c
 - dispatch_interrupt, 323
- interrupt.h
 - dispatch_interrupt, 159
 - idt, 160
- intptr_t

- stdint.h, 228
- invalidate_tlb
 - x86.h, 137
- inw
 - io.h, 161
- io.h
 - inb, 161
 - inl, 161
 - inw, 161
 - outb, 161
 - outl, 161
 - outw, 162
- iomap
 - tss_t, 51
- ipc.c
 - ipc_boot_init, 277
 - ipc_get_proc_object, 277
 - ipc_object_create, 277
 - ipc_receive, 278
 - ipc_reply, 279
 - ipc_send, 281
- ipc_boot_init
 - ipc.c, 277
 - jinue-common/ipc.h, 196
- ipc_get_proc_object
 - ipc.c, 277
 - jinue-common/ipc.h, 197
- ipc_object_create
 - ipc.c, 277
 - jinue-common/ipc.h, 197
- ipc_receive
 - ipc.c, 278
 - jinue-common/ipc.h, 197
- ipc_reply
 - ipc.c, 279
 - jinue-common/ipc.h, 199
- ipc_send
 - ipc.c, 281
 - jinue-common/ipc.h, 201
- ipc_t, 24
 - header, 25
 - recv_list, 25
 - send_list, 25
- irq.h
 - EXCEPTION_ALIGNMENT, 108
 - EXCEPTION_BOUND, 108
 - EXCEPTION_BREAK, 108
 - EXCEPTION_DIV_ZERO, 108
 - EXCEPTION_DOUBLE_FAULT, 109
 - EXCEPTION_GENERAL_PROTECTION, 109
 - EXCEPTION_INVALID_OP, 109
 - EXCEPTION_INVALID_TSS, 109
 - EXCEPTION_MACHINE_CHECK, 109
 - EXCEPTION_MATH, 109
 - EXCEPTION_NMI, 109
 - EXCEPTION_NO_COPROC, 109
 - EXCEPTION_OVERFLOW, 109
 - EXCEPTION_PAGE_FAULT, 110
 - EXCEPTION_SEGMENT_NOT_PRESENT, 110
 - EXCEPTION_SIMD, 110
 - EXCEPTION_STACK_SEGMENT, 110
 - HAS_ERRCODE, 110
 - IDT_FIRST_IRQ, 110
 - IDT_IRQ_COUNT, 110
 - IDT_VECTOR_COUNT, 110
- ivt
 - trapframe_t, 48
- JINUE_ARGS_PACK_BUFFER_SIZE
 - jinue-common/asm/ipc.h, 193
- JINUE_ARGS_PACK_DATA_SIZE
 - jinue-common/asm/ipc.h, 193
- JINUE_ARGS_PACK_N_DESC
 - jinue-common/asm/ipc.h, 193
- JINUE_E2BIG
 - common/errno.h, 204
- JINUE_EAGAIN
 - common/errno.h, 204
- JINUE_EBADF
 - common/errno.h, 204
- JINUE_EINVAL
 - common/errno.h, 204
- JINUE_EIO
 - common/errno.h, 204
- JINUE_EMORE
 - common/errno.h, 204
- JINUE_ENOMEM
 - common/errno.h, 204
- JINUE_ENOSYS
 - common/errno.h, 204
- JINUE_EPERM
 - common/errno.h, 204
- JINUE_IPC_NONE
 - jinue-common/ipc.h, 196
- JINUE_IPC_PROC
 - jinue-common/ipc.h, 196
- JINUE_IPC_SYSTEM
 - jinue-common/ipc.h, 196
- JINUE_LIST_STATIC
 - common/list.h, 206
- JINUE_OFFSETOF
 - common/list.h, 206
- JINUE_SEND_BUFFER_SIZE_OFFSET
 - jinue-common/asm/ipc.h, 193
- JINUE_SEND_DATA_SIZE_OFFSET
 - jinue-common/asm/ipc.h, 193
- JINUE_SEND_MAX_N_DESC
 - jinue-common/asm/ipc.h, 194

- JINUE_SEND_MAX_SIZE
 - jinue-common/asm/ipc.h, 194
- JINUE_SEND_N_DESC_BITS
 - jinue-common/asm/ipc.h, 194
- JINUE_SEND_N_DESC_MASK
 - jinue-common/asm/ipc.h, 194
- JINUE_SEND_N_DESC_OFFSET
 - jinue-common/asm/ipc.h, 194
- JINUE_SEND_SIZE_BITS
 - jinue-common/asm/ipc.h, 194
- JINUE_SEND_SIZE_MASK
 - jinue-common/asm/ipc.h, 194
- jinue-common/asm/ipc.h
 - JINUE_ARGS_PACK_BUFFER_SIZE, 193
 - JINUE_ARGS_PACK_DATA_SIZE, 193
 - JINUE_ARGS_PACK_N_DESC, 193
 - JINUE_SEND_BUFFER_SIZE_OFFSET, 193
 - JINUE_SEND_DATA_SIZE_OFFSET, 193
 - JINUE_SEND_MAX_N_DESC, 194
 - JINUE_SEND_MAX_SIZE, 194
 - JINUE_SEND_N_DESC_BITS, 194
 - JINUE_SEND_N_DESC_MASK, 194
 - JINUE_SEND_N_DESC_OFFSET, 194
 - JINUE_SEND_SIZE_BITS, 194
 - JINUE_SEND_SIZE_MASK, 194
- jinue-common/asm/syscall.h
 - SYSCALL_FUNC_CONSOLE_PUTC, 215
 - SYSCALL_FUNC_CONSOLE_PUTS, 215
 - SYSCALL_FUNC_CREATE_IPC, 215
 - SYSCALL_FUNC_GET_FREE_MEMORY, 215
 - SYSCALL_FUNC_GET_THREAD_LOCAL_ADDR, 215
 - SYSCALL_FUNC_PROC_BASE, 216
 - SYSCALL_FUNC_RECEIVE, 216
 - SYSCALL_FUNC_REPLY, 216
 - SYSCALL_FUNC_SET_THREAD_LOCAL_ADDR, 216
 - SYSCALL_FUNC_SYSCALL_METHOD, 216
 - SYSCALL_FUNC_SYSTEM_BASE, 216
 - SYSCALL_FUNC_THREAD_CREATE, 216
 - SYSCALL_FUNC_THREAD_YIELD, 217
 - SYSCALL_FUNC_USER_BASE, 217
 - SYSCALL_IRQ, 217
 - SYSCALL_METHOD_FAST_AMD, 217
 - SYSCALL_METHOD_FAST_INTEL, 217
 - SYSCALL_METHOD_INTR, 217
- jinue-common/asm/types.h
 - KB, 172
 - MB, 172
- jinue-common/asm/vm.h
 - KLIMIT, 129
 - PAGE_BITS, 129
 - PAGE_MASK, 129
 - PAGE_SIZE, 129
- STACK_BASE, 129
- STACK_SIZE, 130
- STACK_START, 130
- jinue-common/console.h
 - CONSOLE_SERIAL_BAUD_RATE, 61
 - CONSOLE_SERIAL_IOPORT, 61
 - console_init, 61
 - console_print, 61
 - console_printn, 62
 - console_putc, 62
- jinue-common/elf.h
 - AT_BASE, 73
 - AT_DCACHEBSIZE, 73
 - AT_ENTRY, 73
 - AT_EXECPD, 73
 - AT_FLAGS, 73
 - AT_HWCAP, 73
 - AT_HWCAP2, 73
 - AT_ICACHEBSIZE, 73
 - AT_IGNORE, 73
 - AT_NULL, 74
 - AT_PAGESZ, 74
 - AT_PHDR, 74
 - AT_PHEMT, 74
 - AT_PHNUM, 74
 - AT_STACKBASE, 74
 - AT_SYSINFO_EHDR, 74
 - AT_UCACHEBSIZE, 75
 - auxv_t, 83
 - EI_CLASS, 75
 - EI_DATA, 75
 - EI_MAG0, 75
 - EI_MAG1, 75
 - EI_MAG2, 75
 - EI_MAG3, 75
 - EI_NIDENT, 76
 - EI_PAD, 76
 - EI_VERSION, 76
 - ELF32_ST_BIND, 76
 - ELF32_ST_TYPE, 76
 - ELF_MAGIC0, 76
 - ELF_MAGIC1, 76
 - ELF_MAGIC2, 76
 - ELF_MAGIC3, 77
 - ELFCLASS32, 77
 - ELFCLASS64, 77
 - ELFCLASSNONE, 77
 - ELFDATA2LSB, 77
 - ELFDATA2MSB, 77
 - ELFDATANONE, 77
 - EM_386, 77
 - EM_AARCH64, 78
 - EM_ALTERA_NIOS2, 78
 - EM_ARM, 78

- EM_MICROBLAZE, 78
- EM_MIPS, 78
- EM_NONE, 78
- EM_OPENRISC, 78
- EM_SPARC, 78
- EM_SPARC32PLUS, 79
- EM_X86_64, 79
- ET_CORE, 79
- ET_DYN, 79
- ET_EXEC, 79
- ET_NONE, 79
- ET_REL, 79
- Elf32_Addr, 83
- Elf32_Half, 83
- Elf32_Off, 83
- Elf32_Sword, 83
- Elf32_Word, 83
- elf_check, 84
- elf_load, 85
- elf_lookup_symbol, 87
- elf_setup_stack, 88
- PF_R, 79
- PF_W, 79
- PF_X, 80
- PT_DYNAMIC, 80
- PT_INTERP, 80
- PT_LOAD, 80
- PT_NOTE, 80
- PT_NULL, 80
- PT_PHDR, 80
- PT_SHLIB, 80
- SHT_DYNAMIC, 80
- SHT_DYNSYM, 81
- SHT_HASH, 81
- SHT_NOBITS, 81
- SHT_NOTE, 81
- SHT_NULL, 81
- SHT_PROGBITS, 81
- SHT_REL, 81
- SHT_RELA, 81
- SHT_SHLIB, 81
- SHT_STRTAB, 82
- SHT_SYMTAB, 82
- STB_GLOBAL, 82
- STB_LOCAL, 82
- STB_WEAK, 82
- STN_UNDEF, 82
- STT_FILE, 82
- STT_FUNCTION, 82
- STT_NOTYPE, 83
- STT_OBJECT, 83
- STT_SECTION, 83
- jinue-common/ipc.h
 - IPC_FLAG_SYSTEM, 196
 - ipc_boot_init, 196
 - ipc_get_proc_object, 197
 - ipc_object_create, 197
 - ipc_receive, 197
 - ipc_reply, 199
 - ipc_send, 201
 - JINUE_IPC_NONE, 196
 - JINUE_IPC_PROC, 196
 - JINUE_IPC_SYSTEM, 196
 - jinue_ipc_descriptor_t, 196
- jinue-common/pfaddr.h
 - PFADDR_SHIFT, 166
 - PFNULL, 166
 - pfaddr_t, 167
- jinue-common/pfalloc.h
 - KERNEL_PAGE_STACK_INIT, 208
 - KERNEL_PAGE_STACK_SIZE, 209
- jinue-common/vm.h
 - page_address_of, 131
 - page_number_of, 131
 - page_offset_of, 131
- jinue/ipc.h
 - jinue_create_ipc, 192
 - jinue_receive, 192
 - jinue_reply, 192
 - jinue_send, 192
- jinue/syscall.h
 - jinue_call, 213
 - jinue_call_raw, 213
 - jinue_get_free_memory, 214
 - jinue_get_syscall_implementation, 214
 - jinue_get_syscall_implementation_name, 214
 - jinue_get_thread_local_storage, 214
 - jinue_set_thread_local_storage, 214
 - jinue_thread_create, 214
 - jinue_thread_exit, 214
 - jinue_yield, 214
- jinue_call
 - jinue/syscall.h, 213
- jinue_call_raw
 - jinue/syscall.h, 213
- jinue_create_ipc
 - jinue/ipc.h, 192
- jinue_cursor_entry
 - common/list.h, 206
- jinue_cursor_t
 - common/list.h, 207
- jinue_get_free_memory
 - jinue/syscall.h, 214
- jinue_get_syscall_implementation
 - jinue/syscall.h, 214
- jinue_get_syscall_implementation_name
 - jinue/syscall.h, 214

- jinue_get_thread_local_storage
 - jinue/syscall.h, 214
- jinue_ipc_descriptor_t
 - jinue-common/ipc.h, 196
- jinue_list_pop
 - common/list.h, 206
- jinue_list_t, 26
 - head, 26
 - tail, 26
- jinue_message_t, 27
 - buffer_size, 27
 - cookie, 27
 - data_size, 27
 - desc_n, 27
 - function, 27
- jinue_node_entry
 - common/list.h, 206
- jinue_node_t, 27
 - common/list.h, 207
 - next, 28
- jinue_receive
 - jinue/ipc.h, 192
- jinue_reply
 - jinue/ipc.h, 192
- jinue_reply_t, 28
 - data_size, 28
 - desc_n, 28
- jinue_send
 - jinue/ipc.h, 192
- jinue_set_thread_local_storage
 - jinue/syscall.h, 214
- jinue_syscall_args_t, 29
 - arg0, 29
 - arg1, 29
 - arg2, 29
 - arg3, 29
- jinue_thread_create
 - jinue/syscall.h, 214
- jinue_thread_exit
 - jinue/syscall.h, 214
- jinue_yield
 - jinue/syscall.h, 214
- KB
 - jinue-common/asm/types.h, 172
- KERNEL_PAGE_STACK_INIT
 - jinue-common/pfalloc.h, 208
- KERNEL_PAGE_STACK_SIZE
 - jinue-common/pfalloc.h, 209
- KLIMIT
 - jinue-common/asm/vm.h, 129
- kbd.c
 - any_key, 301
- kbd.h
 - any_key, 222
- kernel.h
 - kernel_region_top, 162
- kernel/core/core.c, 266
- kernel/core/elf.c, 268
- kernel/core/ipc.c, 276
- kernel/core/process.c, 283
- kernel/core/syscall.c, 285
- kernel/core/thread.c, 289
- kernel/debug/console.c, 296
- kernel/debug/debug.c, 298
- kernel/debug/kbd.c, 300
- kernel/debug/panic.c, 301
- kernel/hal/boot.c, 303
- kernel/hal/bootmem.c, 306
- kernel/hal/cpu.c, 312
- kernel/hal/e820.c, 316
- kernel/hal/hal.c, 319
- kernel/hal/interrupt.c, 323
- kernel/hal/thread.c, 292
- kernel/hal/vga.c, 324
- kernel/hal/vm.c, 330
- kernel/hal/vm_pae.c, 342
- kernel/kstdd/assert.c, 345
- kernel/kstdd/string.c, 347
- kernel/mem/pfalloc.c, 348
- kernel/mem/slab.c, 352
- kernel/mem/vm_alloc.c, 361
- kernel_context_t, 30
 - ebp, 30
 - ebx, 30
 - edi, 30
 - eip, 30
 - esi, 30
- kernel_region_top
 - hal.c, 323
 - kernel.h, 162
- kernel_size
 - boot_info_t, 10
- kernel_start
 - boot_info_t, 10
- kmmain
 - core.c, 266
 - core.h, 63
- ldt
 - tss_t, 51
- lgdt
 - x86.h, 137
- lidt
 - x86.h, 137
- limit
 - pseudo_descriptor_t, 37
- local_storage_addr

- thread_context_t, 44
- local_storage_size
 - thread_context_t, 44
- lookup_page_directory
 - vm.c, 342
 - vm_private.h, 190
- ltr
 - x86.h, 137
- MB
 - jinue-common/asm/types.h, 172
- MSR_EFER
 - cpu.h, 148
- MSR_FLAG_STAR_SCE
 - cpu.h, 149
- MSR_IA32_SYSENTER_CS
 - cpu.h, 149
- MSR_IA32_SYSENTER_EIP
 - cpu.h, 149
- MSR_IA32_SYSENTER_ESP
 - cpu.h, 149
- MSR_STAR
 - cpu.h, 149
- max_colour
 - slab_cache_t, 40
- memcpy
 - string.c, 347
 - string.h, 229
- memory_block_t, 31
 - addr, 31
 - count, 31
- memset
 - string.c, 348
 - string.h, 230
- message_args
 - thread_t, 45
- message_buffer
 - thread_t, 45
- message_info
 - thread_t, 45
- message_info_t, 31
 - buffer_size, 32
 - cookie, 32
 - data_size, 32
 - desc_n, 32
 - function, 32
 - total_size, 32
- model
 - cpu_info_t, 14
- msg_arg0
 - hal/types.h, 170
- msg_arg1
 - hal/types.h, 170
- msg_arg2
 - hal/types.h, 170
- msg_arg3
 - hal/types.h, 170
- msr_addr_t
 - x86.h, 136
- NULL
 - stddef.h, 226
- name
 - elf_symbol_t, 24
 - slab_cache_t, 40
- new_ram_map_entry
 - bootmem.c, 311
 - bootmem.h, 143
- next
 - bootmem_t, 11
 - jinue_node_t, 28
 - slab_bufctl_t, 38
 - slab_cache_t, 41
 - slab_t, 43
 - vm_block_t, 56
- next_colour
 - slab_cache_t, 41
- OBJECT_FLAG_DESTROYED
 - object.h, 231
- OBJECT_FLAG_NONE
 - object.h, 231
- OBJECT_REF_FLAG_CLOSED
 - object.h, 231
- OBJECT_REF_FLAG_NONE
 - object.h, 231
- OBJECT_REF_FLAG_OWNER
 - object.h, 231
- OBJECT_REF_FLAG_VALID
 - object.h, 231
- OBJECT_TYPE_IPC
 - object.h, 232
- OBJECT_TYPE_PROCESS
 - object.h, 232
- OBJECT_TYPE_THREAD
 - object.h, 232
- OFFSET_OF
 - util.h, 249
- obj_count
 - slab_t, 43
- obj_size
 - slab_cache_t, 41
- object
 - object_ref_t, 34
- object.h
 - OBJECT_FLAG_DESTROYED, 231
 - OBJECT_FLAG_NONE, 231
 - OBJECT_REF_FLAG_CLOSED, 231
 - OBJECT_REF_FLAG_NONE, 231

- OBJECT_REF_FLAG_OWNER, 231
- OBJECT_REF_FLAG_VALID, 231
- OBJECT_TYPE_IPC, 232
- OBJECT_TYPE_PROCESS, 232
- OBJECT_TYPE_THREAD, 232
- object_header_t, 32
 - flags, 33
 - ref_count, 33
 - type, 33
- object_ref_t, 33
 - cookie, 34
 - flags, 34
 - object, 34
- offsetof
 - stddef.h, 227
- outb
 - io.h, 161
- outl
 - io.h, 161
- outw
 - io.h, 162
- p_align
 - Elf32_Phdr, 18
- p_filesz
 - Elf32_Phdr, 18
- p_flags
 - Elf32_Phdr, 18
- p_memsz
 - Elf32_Phdr, 18
- p_offset
 - Elf32_Phdr, 19
- p_paddr
 - Elf32_Phdr, 19
- p_type
 - Elf32_Phdr, 19
- p_vaddr
 - Elf32_Phdr, 19
- PACK_DESCRIPTOR
 - descriptors.h, 103
- PAGE_BITS
 - jinue-common/asm/vm.h, 129
- PAGE_DIRECTORY_OFFSET_OF
 - vm_private.h, 187
- PAGE_MASK
 - jinue-common/asm/vm.h, 129
- PAGE_SIZE
 - jinue-common/asm/vm.h, 129
- PAGE_TABLE_ENTRIES
 - vm_private.h, 187
- PAGE_TABLE_MASK
 - vm_private.h, 187
- PAGE_TABLE_OFFSET_OF
 - vm_private.h, 187
- PDPT_BITS
 - vm_pae.c, 344
- PDPT_ENTRIES
 - vm_pae.c, 344
- PF_R
 - jinue-common/elf.h, 79
- PF_W
 - jinue-common/elf.h, 79
- PF_X
 - jinue-common/elf.h, 80
- PFADDR_CHECK
 - hal/pfaddr.h, 164
- PFADDR_CHECK_4GB
 - hal/pfaddr.h, 164
- PFADDR_SHIFT
 - jinue-common/pfaddr.h, 166
- PFADDR_TO_ADDR
 - hal/pfaddr.h, 164
- PFNULL
 - jinue-common/pfaddr.h, 166
- PROCESS_MAX_DESCRIPTOR_TYPES
 - types.h, 174
- PT_DYNAMIC
 - jinue-common/elf.h, 80
- PT_INTERP
 - jinue-common/elf.h, 80
- PT_LOAD
 - jinue-common/elf.h, 80
- PT_NOTE
 - jinue-common/elf.h, 80
- PT_NULL
 - jinue-common/elf.h, 80
- PT_PHDR
 - jinue-common/elf.h, 80
- PT_SHLIB
 - jinue-common/elf.h, 80
- padding
 - pseudo_descriptor_t, 37
- page_address_of
 - jinue-common/vm.h, 131
- page_directory
 - boot_info_t, 10
- page_directory_offset_of
 - vm.c, 342
 - vm_private.h, 191
- page_number_of
 - jinue-common/vm.h, 131
- page_offset_of
 - jinue-common/vm.h, 131
- page_table
 - boot_info_t, 10
- page_table_entries
 - vm.c, 342
 - vm_private.h, 191

- page_table_offset_of
 - vm.c, 342
 - vm_private.h, 191
- panic
 - panic.c, 302
 - panic.h, 232
- panic.c
 - panic, 302
- panic.h
 - panic, 232
- partial_list
 - vm_alloc_t, 54
- pd
 - addr_space_t, 8
 - pdpt_t, 35
- pdpt
 - addr_space_t, 8
- pdpt_t, 34
 - hal/types.h, 171
 - pd, 35
- pfaddr_t
 - jinue-common/pfaddr.h, 167
- pfalloc
 - pfalloc.h, 210
- pfalloc.c
 - global_pfcache, 351
 - init_pfcache, 349
 - pfalloc_early, 349
 - pfalloc_from, 350
 - pf_free_to, 351
 - use_pfalloc_early, 351
- pfalloc.h
 - global_pfcache, 212
 - init_pfcache, 210
 - pfalloc, 210
 - pfalloc_early, 211
 - pfalloc_from, 211
 - pf_free, 210
 - pf_free_to, 212
 - use_pfalloc_early, 212
- pfalloc_early
 - pfalloc.c, 349
 - pfalloc.h, 211
- pfalloc_from
 - pfalloc.c, 350
 - pfalloc.h, 211
- pfcache_t, 35
 - count, 35
 - ptr, 35
- pf_free
 - pfalloc.h, 210
- pf_free_to
 - pfalloc.c, 351
 - pfalloc.h, 212
- prev
 - slab_cache_t, 41
 - slab_t, 43
 - tss_t, 51
 - vm_block_t, 56
- print_hex_b
 - printk.h, 234
- print_hex_l
 - printk.h, 234
- print_hex_nibble
 - printk.h, 234
- print_hex_q
 - printk.h, 234
- print_hex_w
 - printk.h, 234
- print_unsigned_int
 - printk.h, 234
- printk
 - printk.h, 234
- printk.h
 - print_hex_b, 234
 - print_hex_l, 234
 - print_hex_nibble, 234
 - print_hex_q, 234
 - print_hex_w, 234
 - print_unsigned_int, 234
 - printk, 234
- proc_size
 - boot_info_t, 10
- proc_start
 - boot_info_t, 10
- process
 - thread_t, 45
- process.c
 - process_boot_init, 284
 - process_create, 284
 - process_get_descriptor, 285
 - process_unused_descriptor, 285
- process.h
 - process_boot_init, 235
 - process_create, 235
 - process_get_descriptor, 236
 - process_unused_descriptor, 236
- process_boot_init
 - process.c, 284
 - process.h, 235
- process_create
 - process.c, 284
 - process.h, 235
- process_get_descriptor
 - process.c, 285
 - process.h, 236
- process_t, 35
 - addr_space, 36

- descriptors, 36
 - header, 36
- process_unused_descriptor
 - process.c, 285
 - process.h, 236
- pseudo_descriptor_t, 37
 - addr, 37
 - limit, 37
 - padding, 37
- pte_t, 37
 - entry, 38
 - hal/types.h, 171
- ptr
 - pfcache_t, 35
- ptrdiff_t
 - stddef.h, 227
- RPL_KERNEL
 - asm/descriptors.h, 98
- RPL_USER
 - asm/descriptors.h, 98
- ram_map
 - bootmem.c, 312
 - bootmem.h, 144
- rdmsr
 - x86.h, 137
- recv_list
 - ipc_t, 25
- ref_count
 - object_header_t, 33
- return_from_interrupt
 - trap.h, 168
- SEG_DESCRIPTOR
 - descriptors.h, 103
- SEG_FLAG_16BIT
 - asm/descriptors.h, 98
- SEG_FLAG_16BIT_GATE
 - asm/descriptors.h, 98
- SEG_FLAG_32BIT
 - asm/descriptors.h, 98
- SEG_FLAG_32BIT_GATE
 - asm/descriptors.h, 99
- SEG_FLAG_BUSY
 - asm/descriptors.h, 99
- SEG_FLAG_IN_BYTES
 - asm/descriptors.h, 99
- SEG_FLAG_IN_PAGES
 - asm/descriptors.h, 99
- SEG_FLAG_KERNEL
 - asm/descriptors.h, 99
- SEG_FLAG_NORMAL
 - asm/descriptors.h, 99
- SEG_FLAG_NORMAL_GATE
 - asm/descriptors.h, 99
- SEG_FLAG_NOSYSTEM
 - asm/descriptors.h, 99
- SEG_FLAG_PRESENT
 - asm/descriptors.h, 100
- SEG_FLAG_SYSTEM
 - asm/descriptors.h, 100
- SEG_FLAG_TSS
 - asm/descriptors.h, 100
- SEG_FLAG_USER
 - asm/descriptors.h, 100
- SEG_FLAGS_OFFSET
 - asm/descriptors.h, 100
- SEG_SELECTOR
 - asm/descriptors.h, 100
- SEG_TYPE_CALL_GATE
 - asm/descriptors.h, 100
- SEG_TYPE_CODE
 - asm/descriptors.h, 101
- SEG_TYPE_DATA
 - asm/descriptors.h, 101
- SEG_TYPE_INTERRUPT_GATE
 - asm/descriptors.h, 101
- SEG_TYPE_READ_ONLY
 - asm/descriptors.h, 101
- SEG_TYPE_TASK_GATE
 - asm/descriptors.h, 101
- SEG_TYPE_TRAP_GATE
 - asm/descriptors.h, 101
- SEG_TYPE_TSS
 - asm/descriptors.h, 101
- SHT_DYNAMIC
 - jinue-common/elf.h, 80
- SHT_DYNSYM
 - jinue-common/elf.h, 81
- SHT_HASH
 - jinue-common/elf.h, 81
- SHT_NOBITS
 - jinue-common/elf.h, 81
- SHT_NOTE
 - jinue-common/elf.h, 81
- SHT_NULL
 - jinue-common/elf.h, 81
- SHT_PROGBITS
 - jinue-common/elf.h, 81
- SHT_REL
 - jinue-common/elf.h, 81
- SHT_RELA
 - jinue-common/elf.h, 81
- SHT_SHLIB
 - jinue-common/elf.h, 81
- SHT_STRTAB
 - jinue-common/elf.h, 82
- SHT_SYMTAB
 - jinue-common/elf.h, 82

- SLAB_COMPACT
 - slab.h, 238
- SLAB_DEFAULT_WORKING_SET
 - slab.h, 238
- SLAB_DEFAULTS
 - slab.h, 238
- SLAB_HWCACHE_ALIGN
 - slab.h, 238
- SLAB_POISON
 - slab.h, 239
- SLAB_POISON_ALIVE_VALUE
 - slab.h, 239
- SLAB_POISON_DEAD_VALUE
 - slab.h, 239
- SLAB_RED_ZONE
 - slab.h, 239
- SLAB_RED_ZONE_VALUE
 - slab.h, 239
- SLAB_SIZE
 - slab.h, 239
- STACK_BASE
 - jinue-common/asm/vm.h, 129
- STACK_SIZE
 - jinue-common/asm/vm.h, 130
- STACK_START
 - jinue-common/asm/vm.h, 130
- STB_GLOBAL
 - jinue-common/elf.h, 82
- STB_LOCAL
 - jinue-common/elf.h, 82
- STB_WEAK
 - jinue-common/elf.h, 82
- STN_UNDEF
 - jinue-common/elf.h, 82
- STT_FILE
 - jinue-common/elf.h, 82
- STT_FUNCTION
 - jinue-common/elf.h, 82
- STT_NOTYPE
 - jinue-common/elf.h, 83
- STT_OBJECT
 - jinue-common/elf.h, 83
- STT_SECTION
 - jinue-common/elf.h, 83
- SYSCALL_FUNCT_CONSOLE_PUTC
 - jinue-common/asm/syscall.h, 215
- SYSCALL_FUNCT_CONSOLE_PUTS
 - jinue-common/asm/syscall.h, 215
- SYSCALL_FUNCT_CREATE_IPC
 - jinue-common/asm/syscall.h, 215
- SYSCALL_FUNCT_GET_FREE_MEMORY
 - jinue-common/asm/syscall.h, 215
- SYSCALL_FUNCT_GET_THREAD_LOCAL_ADDR
 - jinue-common/asm/syscall.h, 215
- SYSCALL_FUNCT_PROC_BASE
 - jinue-common/asm/syscall.h, 216
- SYSCALL_FUNCT_RECEIVE
 - jinue-common/asm/syscall.h, 216
- SYSCALL_FUNCT_REPLY
 - jinue-common/asm/syscall.h, 216
- SYSCALL_FUNCT_SET_THREAD_LOCAL_ADDR
 - jinue-common/asm/syscall.h, 216
- SYSCALL_FUNCT_SYSCALL_METHOD
 - jinue-common/asm/syscall.h, 216
- SYSCALL_FUNCT_SYSTEM_BASE
 - jinue-common/asm/syscall.h, 216
- SYSCALL_FUNCT_THREAD_CREATE
 - jinue-common/asm/syscall.h, 216
- SYSCALL_FUNCT_THREAD_YIELD
 - jinue-common/asm/syscall.h, 217
- SYSCALL_FUNCT_USER_BASE
 - jinue-common/asm/syscall.h, 217
- SYSCALL_IRQ
 - jinue-common/asm/syscall.h, 217
- SYSCALL_METHOD_FAST_AMD
 - jinue-common/asm/syscall.h, 217
- SYSCALL_METHOD_FAST_INTEL
 - jinue-common/asm/syscall.h, 217
- SYSCALL_METHOD_INTR
 - jinue-common/asm/syscall.h, 217
- saved_stack_pointer
 - thread_context_t, 44
- seg_descriptor_t
 - hal/types.h, 171
- seg_selector_t
 - hal/types.h, 171
- self
 - cpu_data_t, 12
- send_list
 - ipc_t, 25
- sender
 - thread_t, 45
- set_cr0
 - x86.h, 137
- set_cr1
 - x86.h, 137
- set_cr2
 - x86.h, 137
- set_cr3
 - x86.h, 137
- set_cr4
 - x86.h, 137
- set_cs
 - x86.h, 137
- set_data_segments
 - x86.h, 137
- set_ds
 - x86.h, 138

- set_eflags
 - x86.h, 138
- set_es
 - x86.h, 138
- set_fs
 - x86.h, 138
- set_gs
 - x86.h, 138
- set_pte
 - vm.c, 342
 - vm_private.h, 191
- set_pte_flags
 - vm.c, 342
 - vm_private.h, 191
- set_ss
 - x86.h, 138
- setup_signature
 - boot_info_t, 10
- sh_addr
 - Elf32_Shdr, 20
- sh_addralign
 - Elf32_Shdr, 20
- sh_entsize
 - Elf32_Shdr, 20
- sh_flags
 - Elf32_Shdr, 20
- sh_info
 - Elf32_Shdr, 20
- sh_link
 - Elf32_Shdr, 20
- sh_name
 - Elf32_Shdr, 20
- sh_offset
 - Elf32_Shdr, 20
- sh_size
 - Elf32_Shdr, 20
- sh_type
 - Elf32_Shdr, 20
- size
 - e820_t, 14
- size_t
 - stddef.h, 227
- slab.c
 - slab_cache_alloc, 352
 - slab_cache_create, 354
 - slab_cache_destroy, 356
 - slab_cache_free, 357
 - slab_cache_grow, 358
 - slab_cache_list, 360
 - slab_cache_reap, 360
 - slab_cache_set_working_set, 360
- slab.h
 - SLAB_COMPACT, 238
 - SLAB_DEFAULT_WORKING_SET, 238
 - SLAB_DEFAULTS, 238
 - SLAB_HWCACHE_ALIGN, 238
 - SLAB_POISON, 239
 - SLAB_POISON_ALIVE_VALUE, 239
 - SLAB_POISON_DEAD_VALUE, 239
 - SLAB_RED_ZONE, 239
 - SLAB_RED_ZONE_VALUE, 239
 - SLAB_SIZE, 239
 - slab_bufctl_t, 239
 - slab_cache_alloc, 240
 - slab_cache_create, 242
 - slab_cache_destroy, 243
 - slab_cache_free, 244
 - slab_cache_grow, 246
 - slab_cache_list, 248
 - slab_cache_reap, 247
 - slab_cache_set_working_set, 247
 - slab_cache_t, 239
 - slab_ctor_t, 239
 - slab_t, 240
- slab_bufctl_t, 38
 - next, 38
 - slab.h, 239
- slab_cache_alloc
 - slab.c, 352
 - slab.h, 240
- slab_cache_create
 - slab.c, 354
 - slab.h, 242
- slab_cache_destroy
 - slab.c, 356
 - slab.h, 243
- slab_cache_free
 - slab.c, 357
 - slab.h, 244
- slab_cache_grow
 - slab.c, 358
 - slab.h, 246
- slab_cache_list
 - slab.c, 360
 - slab.h, 248
- slab_cache_reap
 - slab.c, 360
 - slab.h, 247
- slab_cache_set_working_set
 - slab.c, 360
 - slab.h, 247
- slab_cache_t, 39
 - alignment, 40
 - alloc_size, 40
 - bufctl_offset, 40
 - ctor, 40
 - dtor, 40
 - empty_count, 40

- flags, 40
- max_colour, 40
- name, 40
- next, 41
- next_colour, 41
- obj_size, 41
- prev, 41
- slab.h, 239
- slabs_empty, 41
- slabs_full, 41
- slabs_partial, 41
- working_set, 41
- slab_ctor_t
 - slab.h, 239
- slab_t, 42
 - cache, 42
 - colour, 43
 - free_list, 43
 - next, 43
 - obj_count, 43
 - prev, 43
 - slab.h, 240
- slabs_empty
 - slab_cache_t, 41
- slabs_full
 - slab_cache_t, 41
- slabs_partial
 - slab_cache_t, 41
- ss
 - trapframe_t, 48
 - tss_t, 51
- ss0
 - tss_t, 52
- ss1
 - tss_t, 52
- ss2
 - tss_t, 52
- st_info
 - Elf32_Sym, 21
- st_name
 - Elf32_Sym, 21
- st_other
 - Elf32_Sym, 21
- st_shndx
 - Elf32_Sym, 21
- st_size
 - Elf32_Sym, 21
- st_value
 - Elf32_Sym, 22
- stack_addr
 - elf_info_t, 23
 - vm_block_t, 56
- stack_next
 - vm_block_t, 56
- stack_ptr
 - vm_block_t, 56
- start_addr
 - vm_alloc_t, 54
- startup.h
 - halt, 167
- stdarg.h
 - va_arg, 225
 - va_copy, 225
 - va_end, 225
 - va_list, 225
 - va_start, 225
- stdbool.h
 - __bool_true_false_are_defined, 226
 - bool, 226
 - false, 226
 - true, 226
- stddef.h
 - NULL, 226
 - offsetof, 227
 - ptrdiff_t, 227
 - size_t, 227
 - wchar_t, 227
- stdint.h
 - INT64_C, 228
 - int16_t, 228
 - int32_t, 228
 - int64_t, 228
 - int8_t, 228
 - intptr_t, 228
 - UINT64_C, 228
 - uint16_t, 228
 - uint32_t, 228
 - uint64_t, 228
 - uint8_t, 229
 - uintptr_t, 229
- stepping
 - cpu_info_t, 14
- sti
 - x86.h, 138
- string.c
 - memcpy, 347
 - memset, 348
 - strlen, 348
- string.h
 - memcpy, 229
 - memset, 230
 - strlen, 230
- strlen
 - string.c, 348
 - string.h, 230
- syscall.c
 - dispatch_syscall, 286
- syscall.h

- dispatch_syscall, 219
- syscall_method
 - hal.c, 323
 - trap.h, 168
- THREAD_CONTEXT_MASK
 - hal/asm/thread.h, 111
- THREAD_CONTEXT_SIZE
 - hal/asm/thread.h, 111
- TSS_LIMIT
 - asm/descriptors.h, 101
- tail
 - jinue_list_t, 26
- thread.h
 - thread_create, 115
 - thread_ready, 116
 - thread_switch, 116
 - thread_yield_from, 117
- thread_context_switch
 - hal/thread.c, 293
 - hal/thread.h, 112
- thread_context_switch_stack
 - hal/thread.c, 294
- thread_context_t, 43
 - local_storage_addr, 44
 - local_storage_size, 44
 - saved_stack_pointer, 44
- thread_create
 - core/thread.c, 290
 - thread.h, 115
- thread_ctx
 - thread_t, 45
- thread_list
 - thread_t, 46
- thread_page_create
 - hal/thread.c, 294
 - hal/thread.h, 113
- thread_page_destroy
 - hal/thread.c, 295
 - hal/thread.h, 114
- thread_ready
 - core/thread.c, 291
 - thread.h, 116
- thread_switch
 - core/thread.c, 291
 - thread.h, 116
- thread_t, 44
 - header, 45
 - message_args, 45
 - message_buffer, 45
 - message_info, 45
 - process, 45
 - sender, 45
 - thread_ctx, 45
 - thread_list, 46
 - types.h, 174
- thread_yield_from
 - core/thread.c, 292
 - thread.h, 117
- top_level
 - addr_space_t, 8
- total_size
 - message_info_t, 32
- trap.h
 - fast_amd_entry, 168
 - fast_intel_entry, 168
 - return_from_interrupt, 168
 - syscall_method, 168
- trapframe_t, 46
 - cs, 47
 - ds, 47
 - eax, 47
 - ebp, 47
 - ebx, 47
 - ecx, 47
 - edi, 47
 - edx, 47
 - eflags, 47
 - eip, 47
 - errcode, 47
 - es, 48
 - esi, 48
 - esp, 48
 - fs, 48
 - gs, 48
 - ivt, 48
 - ss, 48
- true
 - stdbool.h, 226
- tss
 - cpu_data_t, 13
- tss_t, 49
 - cr3, 49
 - cs, 49
 - debug, 49
 - ds, 50
 - eax, 50
 - ebp, 50
 - ebx, 50
 - ecx, 50
 - edi, 50
 - edx, 50
 - eflags, 50
 - eip, 50
 - es, 50
 - esi, 50
 - esp, 51
 - esp0, 51

- esp1, 51
- esp2, 51
- fs, 51
- gs, 51
- iomap, 51
- ldt, 51
- prev, 51
- ss, 51
- ss0, 52
- ss1, 52
- ss2, 52
- type
 - e820_t, 15
 - object_header_t, 33
- types.h
 - PROCESS_MAX_DESCRIPTOR, 174
 - thread_t, 174
- UINT64_C
 - stdint.h, 228
- uint16_t
 - stdint.h, 228
- uint32_t
 - stdint.h, 228
- uint64_t
 - stdint.h, 228
- uint8_t
 - stdint.h, 229
- uintptr_t
 - stdint.h, 229
- use_pfallloc_early
 - pfalloc.c, 351
 - pfalloc.h, 212
- util.h
 - ALIGN_END, 249
 - ALIGN_START, 249
 - alloc_backward, 249
 - alloc_forward, 249
 - OFFSET_OF, 249
- VGA_COL
 - vga.h, 176
- VGA_COLOR_BLACK
 - vga.h, 176
- VGA_COLOR_BLUE
 - vga.h, 176
- VGA_COLOR_BRIGHTBLUE
 - vga.h, 176
- VGA_COLOR_BRIGHTCYAN
 - vga.h, 176
- VGA_COLOR_BRIGHTGREEN
 - vga.h, 176
- VGA_COLOR_BRIGHTMAGENTA
 - vga.h, 176
- VGA_COLOR_BRIGHTRED
 - vga.h, 176
- VGA_COLOR_BRIGHTWHITE
 - vga.h, 176
- VGA_COLOR_BROWN
 - vga.h, 176
- VGA_COLOR_CYAN
 - vga.h, 176
- VGA_COLOR_DEFAULT
 - vga.h, 176
- VGA_COLOR_ERASE
 - vga.h, 177
- VGA_COLOR_GRAY
 - vga.h, 177
- VGA_COLOR_GREEN
 - vga.h, 177
- VGA_COLOR_MAGENTA
 - vga.h, 177
- VGA_COLOR_RED
 - vga.h, 177
- VGA_COLOR_WHITE
 - vga.h, 177
- VGA_COLOR_YELLOW
 - vga.h, 177
- VGA_CRTC_ADDR
 - vga.h, 177
- VGA_CRTC_DATA
 - vga.h, 177
- VGA_FB_FLAG_ACTIVE
 - vga.h, 177
- VGA_LINE
 - vga.h, 178
- VGA_LINES
 - vga.h, 178
- VGA_MISC_OUT_RD
 - vga.h, 178
- VGA_MISC_OUT_WR
 - vga.h, 178
- VGA_TAB_WIDTH
 - vga.h, 178
- VGA_TEXT_VID_BASE
 - vga.h, 178
- VGA_TEXT_VID_SIZE
 - vga.h, 178
- VGA_TEXT_VID_TOP
 - vga.h, 178
- VGA_WIDTH
 - vga.h, 178
- VM_ALLOC_BLOCK_MASK
 - vm_alloc.h, 251
- VM_ALLOC_BLOCK_SIZE
 - vm_alloc.h, 251
- VM_ALLOC_CANNOT_GROW
 - vm_alloc.h, 251
- VM_ALLOC_EMPTY_STACK

- vm_alloc.h, 251
- VM_ALLOC_FULL_STACK
 - vm_alloc.h, 251
- VM_ALLOC_IS_FREE
 - vm_alloc.h, 251
- VM_ALLOC_IS_PARTIAL
 - vm_alloc.h, 251
- VM_ALLOC_IS_USED
 - vm_alloc.h, 251
- VM_ALLOC_STACK_ENTRIES
 - vm_alloc.h, 252
- VM_ALLOC_WAS_FREE
 - vm_alloc.h, 252
- VM_ALLOC_WAS_USED
 - vm_alloc.h, 252
- VM_FLAG_ACCESSED
 - hal/asm/vm.h, 119
- VM_FLAG_DIRTY
 - hal/asm/vm.h, 119
- VM_FLAG_KERNEL
 - hal/asm/vm.h, 119
- VM_FLAG_PRESENT
 - hal/asm/vm.h, 119
- VM_FLAG_READ_ONLY
 - hal/asm/vm.h, 119
- VM_FLAG_READ_WRITE
 - hal/asm/vm.h, 119
- VM_FLAG_USER
 - hal/asm/vm.h, 119
- va_arg
 - stdarg.h, 225
- va_copy
 - stdarg.h, 225
- va_end
 - stdarg.h, 225
- va_list
 - stdarg.h, 225
- va_start
 - stdarg.h, 225
- vendor
 - cpu_info_t, 14
- vga.c
 - vga_clear, 325
 - vga_get_color, 325
 - vga_get_cursor_pos, 326
 - vga_init, 326
 - vga_print, 327
 - vga_printn, 327
 - vga_putc, 328
 - vga_scroll, 328
 - vga_set_base_addr, 329
 - vga_set_color, 329
 - vga_set_cursor_pos, 329
- vga.h
 - VGA_COL, 176
 - VGA_COLOR_BLACK, 176
 - VGA_COLOR_BLUE, 176
 - VGA_COLOR_BRIGHTBLUE, 176
 - VGA_COLOR_BRIGHTCYAN, 176
 - VGA_COLOR_BRIGHTGREEN, 176
 - VGA_COLOR_BRIGHTMAGENTA, 176
 - VGA_COLOR_BRIGHTRED, 176
 - VGA_COLOR_BRIGHTWHITE, 176
 - VGA_COLOR_BROWN, 176
 - VGA_COLOR_CYAN, 176
 - VGA_COLOR_DEFAULT, 176
 - VGA_COLOR_ERASE, 177
 - VGA_COLOR_GRAY, 177
 - VGA_COLOR_GREEN, 177
 - VGA_COLOR_MAGENTA, 177
 - VGA_COLOR_RED, 177
 - VGA_COLOR_WHITE, 177
 - VGA_COLOR_YELLOW, 177
 - VGA_CRTC_ADDR, 177
 - VGA_CRTC_DATA, 177
 - VGA_FB_FLAG_ACTIVE, 177
 - VGA_LINE, 178
 - VGA_LINES, 178
 - VGA_MISC_OUT_RD, 178
 - VGA_MISC_OUT_WR, 178
 - VGA_TAB_WIDTH, 178
 - VGA_TEXT_VID_BASE, 178
 - VGA_TEXT_VID_SIZE, 178
 - VGA_TEXT_VID_TOP, 178
 - VGA_WIDTH, 178
 - vga_clear, 179
 - vga_get_color, 179
 - vga_get_cursor_pos, 179
 - vga_init, 180
 - vga_pos_t, 179
 - vga_print, 180
 - vga_printn, 181
 - vga_putc, 181
 - vga_scroll, 182
 - vga_set_base_addr, 182
 - vga_set_color, 182
 - vga_set_cursor_pos, 183
- vga_clear
 - vga.c, 325
 - vga.h, 179
- vga_get_color
 - vga.c, 325
 - vga.h, 179
- vga_get_cursor_pos
 - vga.c, 326
 - vga.h, 179
- vga_init
 - vga.c, 326

- vga.h, 180
- vga_pos_t
 - vga.h, 179
- vga_print
 - vga.c, 327
 - vga.h, 180
- vga_printn
 - vga.c, 327
 - vga.h, 181
- vga_putc
 - vga.c, 328
 - vga.h, 181
- vga_scroll
 - vga.c, 328
 - vga.h, 182
- vga_set_base_addr
 - vga.c, 329
 - vga.h, 182
- vga_set_color
 - vga.c, 329
 - vga.h, 182
- vga_set_cursor_pos
 - vga.c, 329
 - vga.h, 183
- vm.c
 - clear_pte, 340
 - copy_pte, 340
 - create_addr_space, 340
 - create_initial_addr_space, 341
 - destroy_addr_space, 341
 - get_pte_flags, 341
 - get_pte_pfaddr, 341
 - get_pte_with_offset, 341
 - global_page_allocator, 341
 - global_page_tables, 341
 - initial_addr_space, 341
 - lookup_page_directory, 342
 - page_directory_offset_of, 342
 - page_table_entries, 342
 - page_table_offset_of, 342
 - set_pte, 342
 - set_pte_flags, 342
 - vm_allocate_page_directory, 331
 - vm_boot_init, 332
 - vm_change_flags, 334
 - vm_clone_page_directory, 335
 - vm_create_addr_space, 336
 - vm_create_initial_addr_space, 336
 - vm_destroy_addr_space, 336
 - vm_destroy_page_directory, 336
 - vm_lookup_pfaddr, 337
 - vm_map_early, 337
 - vm_map_kernel, 338
 - vm_map_user, 338
 - vm_switch_addr_space, 338
 - vm_unmap, 338
 - vm_unmap_kernel, 339
 - vm_unmap_user, 339
 - vm_x86_create_initial_addr_space, 340
- vm_alloc
 - vm_alloc.c, 362
 - vm_alloc.h, 252
- vm_alloc.c
 - vm_alloc, 362
 - vm_alloc_add_region, 363
 - vm_alloc_custom_block, 364
 - vm_alloc_destroy, 365
 - vm_alloc_free_block, 366
 - vm_alloc_grow_single, 367
 - vm_alloc_grow_stack, 368
 - vm_alloc_init, 368
 - vm_alloc_init_allocator, 369
 - vm_alloc_low_latency, 370
 - vm_alloc_partial_block, 371
 - vm_alloc_unlink_block, 373
 - vm_free, 375
- vm_alloc.h
 - global_page_allocator, 266
 - VM_ALLOC_BLOCK_MASK, 251
 - VM_ALLOC_BLOCK_SIZE, 251
 - VM_ALLOC_CANNOT_GROW, 251
 - VM_ALLOC_EMPTY_STACK, 251
 - VM_ALLOC_FULL_STACK, 251
 - VM_ALLOC_IS_FREE, 251
 - VM_ALLOC_IS_PARTIAL, 251
 - VM_ALLOC_IS_USED, 251
 - VM_ALLOC_STACK_ENTRIES, 252
 - VM_ALLOC_WAS_FREE, 252
 - VM_ALLOC_WAS_USED, 252
 - vm_alloc, 252
 - vm_alloc_add_region, 253
 - vm_alloc_custom_block, 254
 - vm_alloc_destroy, 255
 - vm_alloc_free_block, 256
 - vm_alloc_grow_single, 257
 - vm_alloc_grow_stack, 258
 - vm_alloc_init, 258
 - vm_alloc_init_allocator, 259
 - vm_alloc_low_latency, 260
 - vm_alloc_partial_block, 261
 - vm_alloc_t, 252
 - vm_alloc_unlink_block, 263
 - vm_block_t, 252
 - vm_free, 265
- vm_alloc_add_region
 - vm_alloc.c, 363
 - vm_alloc.h, 253
- vm_alloc_custom_block

- vm_alloc.c, 364
- vm_alloc.h, 254
- vm_alloc_destroy
 - vm_alloc.c, 365
 - vm_alloc.h, 255
- vm_alloc_free_block
 - vm_alloc.c, 366
 - vm_alloc.h, 256
- vm_alloc_grow_single
 - vm_alloc.c, 367
 - vm_alloc.h, 257
- vm_alloc_grow_stack
 - vm_alloc.c, 368
 - vm_alloc.h, 258
- vm_alloc_init
 - vm_alloc.c, 368
 - vm_alloc.h, 258
- vm_alloc_init_allocator
 - vm_alloc.c, 369
 - vm_alloc.h, 259
- vm_alloc_low_latency
 - vm_alloc.c, 370
 - vm_alloc.h, 260
- vm_alloc_partial_block
 - vm_alloc.c, 371
 - vm_alloc.h, 261
- vm_alloc_t, 52
 - array_pages, 53
 - base_addr, 53
 - block_array, 53
 - block_count, 53
 - end_addr, 53
 - free_list, 54
 - partial_list, 54
 - start_addr, 54
 - vm_alloc.h, 252
- vm_alloc_unlink_block
 - vm_alloc.c, 373
 - vm_alloc.h, 263
- vm_allocate_page_directory
 - vm.c, 331
 - vm_private.h, 187
- vm_block_t, 54
 - allocator, 55
 - base_addr, 55
 - next, 56
 - prev, 56
 - stack_addr, 56
 - stack_next, 56
 - stack_ptr, 56
 - vm_alloc.h, 252
- vm_boot_init
 - hal/vm.h, 122
 - vm.c, 332
- vm_change_flags
 - hal/vm.h, 124
 - vm.c, 334
- vm_clone_page_directory
 - vm.c, 335
 - vm_private.h, 188
- vm_create_addr_space
 - hal/vm.h, 124
 - vm.c, 336
- vm_create_initial_addr_space
 - hal/vm.h, 125
 - vm.c, 336
- vm_destroy_addr_space
 - hal/vm.h, 125
 - vm.c, 336
- vm_destroy_page_directory
 - vm.c, 336
 - vm_private.h, 189
- vm_free
 - vm_alloc.c, 375
 - vm_alloc.h, 265
- vm_lookup_pfaddr
 - hal/vm.h, 125
 - vm.c, 337
- vm_map_early
 - hal/vm.h, 125
 - vm.c, 337
- vm_map_kernel
 - hal/vm.h, 126
 - vm.c, 338
- vm_map_user
 - hal/vm.h, 126
 - vm.c, 338
- vm_pae.c
 - initial_pdpt, 345
 - PDPT_BITS, 344
 - PDPT_ENTRIES, 344
 - vm_pae_boot_init, 344
 - vm_pae_create_pdpt_cache, 344
 - vm_pae_enable, 345
- vm_pae.h
 - vm_pae_boot_init, 184
 - vm_pae_create_pdpt_cache, 184
 - vm_pae_enable, 184
- vm_pae_boot_init
 - vm_pae.c, 344
 - vm_pae.h, 184
- vm_pae_create_pdpt_cache
 - vm_pae.c, 344
 - vm_pae.h, 184
- vm_pae_enable
 - vm_pae.c, 345
 - vm_pae.h, 184
- vm_private.h

- clear_pte, 189
- copy_pte, 189
- create_addr_space, 190
- create_initial_addr_space, 190
- destroy_addr_space, 190
- get_pte_flags, 190
- get_pte_pfaddr, 190
- get_pte_with_offset, 190
- global_page_tables, 190
- initial_addr_space, 190
- lookup_page_directory, 190
- PAGE_DIRECTORY_OFFSET_OF, 187
- PAGE_TABLE_ENTRIES, 187
- PAGE_TABLE_MASK, 187
- PAGE_TABLE_OFFSET_OF, 187
- page_directory_offset_of, 191
- page_table_entries, 191
- page_table_offset_of, 191
- set_pte, 191
- set_pte_flags, 191
- vm_allocate_page_directory, 187
- vm_clone_page_directory, 188
- vm_destroy_page_directory, 189
- vm_switch_addr_space
 - hal/vm.h, 126
 - vm.c, 338
- vm_unmap
 - vm.c, 338
- vm_unmap_kernel
 - hal/vm.h, 127
 - vm.c, 339
- vm_unmap_user
 - hal/vm.h, 127
 - vm.c, 339
- vm_x86_create_initial_addr_space
 - vm.c, 340
- wchar_t
 - stddef.h, 227
- working_set
 - slab_cache_t, 41
- wrmsr
 - x86.h, 138
- x86.h
 - cli, 136
 - cpuid, 136
 - get_cr0, 136
 - get_cr1, 136
 - get_cr2, 136
 - get_cr3, 136
 - get_cr4, 136
 - get_eflags, 136
 - get_esp, 137
 - get_gs_ptr, 137
 - invalidate_tlb, 137
 - lgdt, 137
 - lidt, 137
 - ltr, 137
 - msr_addr_t, 136
 - rdmsr, 137
 - set_cr0, 137
 - set_cr1, 137
 - set_cr2, 137
 - set_cr3, 137
 - set_cr4, 137
 - set_cs, 137
 - set_data_segments, 137
 - set_ds, 138
 - set_eflags, 138
 - set_es, 138
 - set_fs, 138
 - set_gs, 138
 - set_ss, 138
 - sti, 138
 - wrmsr, 138
- X86_CR0_PG
 - asm/x86.h, 132
- X86_CR0_WP
 - asm/x86.h, 132
- X86_CR4_PAE
 - asm/x86.h, 133
- X86_CR4_PGE
 - asm/x86.h, 133
- X86_CR4_PSE
 - asm/x86.h, 133
- X86_PDE_PAGE_SIZE
 - asm/x86.h, 133
- X86_PTE_ACCESSED
 - asm/x86.h, 133
- X86_PTE_CACHE_DISABLE
 - asm/x86.h, 133
- X86_PTE_DIRTY
 - asm/x86.h, 133
- X86_PTE_GLOBAL
 - asm/x86.h, 133
- X86_PTE_NX
 - asm/x86.h, 134
- X86_PTE_PRESENT
 - asm/x86.h, 134
- X86_PTE_READ_WRITE
 - asm/x86.h, 134
- X86_PTE_USER
 - asm/x86.h, 134
- X86_PTE_WRITE_THROUGH
 - asm/x86.h, 134
- x86_cpuid_regs_t, 57
- eax, 57
- ebx, 57

ecx, 57

edx, 57