

---

# **Linux X86 Documentation**

**The kernel development community**

**Jan 15, 2023**



# CONTENTS

<b>1</b>	<b>The Linux/x86 Boot Protocol</b>	<b>1</b>
1.1	Memory Layout . . . . .	3
1.2	The Real-Mode Kernel Header . . . . .	4
1.3	Details of Header Fields . . . . .	6
1.4	The kernel_info . . . . .	16
1.5	Details of the kernel_info Fields . . . . .	18
1.6	The Image Checksum . . . . .	18
1.7	The Kernel Command Line . . . . .	18
1.8	Memory Layout of The Real-Mode Code . . . . .	19
1.9	Sample Boot Configurartion . . . . .	19
1.10	Loading The Rest of The Kernel . . . . .	21
1.11	Special Command Line Options . . . . .	21
1.12	Running the Kernel . . . . .	22
1.13	Advanced Boot Loader Hooks . . . . .	22
1.14	32-bit Boot Protocol . . . . .	23
1.15	64-bit Boot Protocol . . . . .	23
1.16	EFI Handover Protocol (deprecated) . . . . .	24
<b>2</b>	<b>DeviceTree Booting</b>	<b>25</b>
<b>3</b>	<b>x86 Feature Flags</b>	<b>27</b>
3.1	Introduction . . . . .	27
3.2	How are feature flags created? . . . . .	27
3.3	Naming of Flags . . . . .	28
3.4	Flags are missing when one or more of these happen . . . . .	29
<b>4</b>	<b>x86 Topology</b>	<b>31</b>
4.1	Package . . . . .	31
4.2	Cores . . . . .	32
4.3	Threads . . . . .	33
4.4	System topology examples . . . . .	33
<b>5</b>	<b>Kernel level exception handling</b>	<b>37</b>
<b>6</b>	<b>Kernel Stacks</b>	<b>45</b>
6.1	Kernel stacks on x86-64 bit . . . . .	45
6.2	Printing backtraces on x86 . . . . .	46
<b>7</b>	<b>Kernel Entries</b>	<b>49</b>

<b>8</b>	<b>Early Printk</b>	<b>51</b>
8.1	Hardware requirements . . . . .	51
8.2	Software requirements . . . . .	52
8.3	Testing . . . . .	53
<b>9</b>	<b>ORC unwinder</b>	<b>55</b>
9.1	Overview . . . . .	55
9.2	ORC vs frame pointers . . . . .	55
9.3	ORC vs DWARF . . . . .	56
9.4	ORC unwind table generation . . . . .	56
9.5	Unwinder implementation details . . . . .	57
9.6	Etymology . . . . .	57
<b>10</b>	<b>Zero Page</b>	<b>59</b>
<b>11</b>	<b>The TLB</b>	<b>61</b>
<b>12</b>	<b>MTRR (Memory Type Range Register) control</b>	<b>63</b>
12.1	Phasing out MTRR use . . . . .	63
12.2	Reading MTRRs from the shell . . . . .	64
12.3	Creating overlapping MTRRs . . . . .	65
12.4	Removing MTRRs from the C-shel . . . . .	65
12.5	Reading MTRRs from a C program using ioctl()’s . . . . .	65
12.6	Creating MTRRs from a C programme using ioctl()’s . . . . .	67
<b>13</b>	<b>PAT (Page Attribute Table)</b>	<b>71</b>
13.1	PAT APIs . . . . .	71
13.2	Advanced APIs for drivers . . . . .	72
13.3	MTRR effects on PAT / non-PAT systems . . . . .	73
13.4	PAT debugging . . . . .	74
13.5	PAT Initialization . . . . .	74
<b>14</b>	<b>Hardware-Feedback Interface for scheduling on Intel Hardware</b>	<b>77</b>
14.1	Overview . . . . .	77
14.2	The Hardware Feedback Interface . . . . .	77
14.3	Implementation details for Linux . . . . .	78
14.4	References . . . . .	78
<b>15</b>	<b>x86 IOMMU Support</b>	<b>79</b>
15.1	Basic stuff . . . . .	79
15.2	How is an I/O Virtual Address (IOVA) generated? . . . . .	80
15.3	Intel Specific Notes . . . . .	80
15.4	AMD Specific Notes . . . . .	80
15.5	Fault reporting . . . . .	81
15.6	Kernel Log Samples . . . . .	81
<b>16</b>	<b>Intel(R) TXT Overview</b>	<b>83</b>
16.1	Trusted Boot Project Overview . . . . .	83
16.2	Value Proposition for Linux or “Why should you care?” . . . . .	84
16.3	How Does it Work? . . . . .	84
16.4	Configuring the System . . . . .	86
<b>17</b>	<b>AMD Memory Encryption</b>	<b>87</b>

<b>18</b>	<b>AMD HSMP interface</b>	<b>89</b>
18.1	HSMP device . . . . .	89
18.2	An example . . . . .	90
<b>19</b>	<b>Intel Trust Domain Extensions (TDX)</b>	<b>91</b>
19.1	New TDX Exceptions . . . . .	91
19.2	#VE on Memory Accesses . . . . .	93
19.3	Linux #VE handler . . . . .	93
19.4	MMIO handling . . . . .	94
19.5	Shared Memory Conversions . . . . .	94
19.6	References . . . . .	95
<b>20</b>	<b>Page Table Isolation (PTI)</b>	<b>97</b>
20.1	Overview . . . . .	97
20.2	Page Table Management . . . . .	97
20.3	Overhead . . . . .	98
20.4	Possible Future Work . . . . .	99
20.5	Testing . . . . .	99
20.6	Debugging . . . . .	99
<b>21</b>	<b>Microarchitectural Data Sampling (MDS) mitigation</b>	<b>101</b>
21.1	Overview . . . . .	101
21.2	Exposure assumptions . . . . .	102
21.3	Mitigation strategy . . . . .	102
21.4	Kernel internal mitigation modes . . . . .	103
21.5	Mitigation points . . . . .	103
<b>22</b>	<b>The Linux Microcode Loader</b>	<b>105</b>
22.1	Early load microcode . . . . .	105
22.2	Late loading . . . . .	106
22.3	Builtin microcode . . . . .	107
<b>23</b>	<b>User Interface for Resource Control feature</b>	<b>109</b>
23.1	Info directory . . . . .	110
23.2	Resource alloc and monitor groups . . . . .	111
23.3	Notes on cache occupancy monitoring and control . . . . .	113
23.4	Memory bandwidth Allocation and monitoring . . . . .	114
23.5	Cache Pseudo-Locking . . . . .	116
23.6	Examples for RDT Monitoring along with allocation usage . . . . .	127
23.7	Intel RDT Errata . . . . .	129
<b>24</b>	<b>TSX Async Abort (TAA) mitigation</b>	<b>131</b>
24.1	Overview . . . . .	131
24.2	Mitigation strategy . . . . .	131
24.3	Kernel internal mitigation modes . . . . .	131
<b>25</b>	<b>Bus lock detection and handling</b>	<b>135</b>
25.1	Problem . . . . .	135
25.2	Detection . . . . .	135
25.3	Software handling . . . . .	136
25.4	Usages . . . . .	136
25.5	Guidance . . . . .	136

<b>26</b>	<b>USB Legacy support</b>	<b>139</b>
<b>27</b>	<b>i386 Support</b>	<b>141</b>
27.1	IO-APIC . . . . .	141
<b>28</b>	<b>x86_64 Support</b>	<b>145</b>
28.1	AMD64 Specific Boot Options . . . . .	145
28.2	General note on [U]EFI x86_64 support . . . . .	150
28.3	Memory Management . . . . .	151
28.4	5-level paging . . . . .	154
28.5	Fake NUMA For CPUsets . . . . .	156
28.6	Firmware support for CPU hotplug under Linux/x86-64 . . . . .	157
28.7	Configurable sysfs parameters for the x86-64 machine check code . . . . .	157
28.8	Using FS and GS segments in user space applications . . . . .	158
<b>29</b>	<b>In-Field Scan</b>	<b>163</b>
29.1	Introduction . . . . .	163
29.2	IFS Image . . . . .	163
29.3	IFS Image Loading . . . . .	163
29.4	Running tests . . . . .	163
29.5	Driver design choices . . . . .	164
<b>30</b>	<b>Shared Virtual Addressing (SVA) with ENQCMD</b>	<b>167</b>
30.1	Background . . . . .	167
30.2	Shared Hardware Workqueues . . . . .	167
30.3	ENQCMD . . . . .	168
30.4	Process Address Space Tagging . . . . .	168
30.5	PASID Management . . . . .	168
30.6	PASID Life Cycle Management . . . . .	169
30.7	Relationships . . . . .	169
30.8	FAQ . . . . .	170
30.9	References . . . . .	171
<b>31</b>	<b>Software Guard eXtensions (SGX)</b>	<b>173</b>
31.1	Overview . . . . .	173
31.2	Enclave Page Cache . . . . .	173
31.3	Application interface . . . . .	174
31.4	ksgxd . . . . .	177
31.5	Launch Control . . . . .	178
31.6	Encryption engines . . . . .	178
31.7	Usage Models . . . . .	179
31.8	Impact of Potential Kernel SGX Bugs . . . . .	179
31.9	Virtual EPC . . . . .	179
<b>32</b>	<b>Feature status on x86 architecture</b>	<b>181</b>
<b>33</b>	<b>x86-specific ELF Auxiliary Vectors</b>	<b>183</b>
33.1	Introduction . . . . .	183
33.2	The exposed auxiliary vectors . . . . .	184
<b>34</b>	<b>Using XSTATE features in user space applications</b>	<b>185</b>
34.1	Using dynamically enabled XSTATE features in user space applications . . . . .	185
34.2	Dynamic features in signal frames . . . . .	186







## **THE LINUX/X86 BOOT PROTOCOL**

On the x86 platform, the Linux kernel uses a rather complicated boot convention. This has evolved partially due to historical aspects, as well as the desire in the early days to have the kernel itself be a bootable image, the complicated PC memory model and due to changed expectations in the PC industry caused by the effective demise of real-mode DOS as a mainstream operating system.

Currently, the following versions of the Linux/x86 boot protocol exist.

Old kernels	zImage/Image support only. Some very early kernels may not even support a command line.
Protocol 2.00	(Kernel 1.3.73) Added bzImage and initrd support, as well as a formalized way to communicate between the boot loader and the kernel. setup.S made relocatable, although the traditional setup area still assumed writable.
Protocol 2.01	(Kernel 1.3.76) Added a heap overrun warning.
Protocol 2.02	(Kernel 2.4.0-test3-pre3) New command line protocol. Lower the conventional memory ceiling. No overwrite of the traditional setup area, thus making booting safe for systems which use the EBDA from SMM or 32-bit BIOS entry points. zImage deprecated but still supported.
Protocol 2.03	(Kernel 2.4.18-pre1) Explicitly makes the highest possible initrd address available to the bootloader.
Protocol 2.04	(Kernel 2.6.14) Extend the syssize field to four bytes.
Protocol 2.05	(Kernel 2.6.20) Make protected mode kernel relocatable. Introduce relocatable_kernel and kernel_alignment fields.
Protocol 2.06	(Kernel 2.6.22) Added a field that contains the size of the boot command line.
Protocol 2.07	(Kernel 2.6.24) Added paravirtualised boot protocol. Introduced hardware_subarch and hardware_subarch_data and KEEP_SEGMENTS flag in load_flags.
Protocol 2.08	(Kernel 2.6.26) Added crc32 checksum and ELF format payload. Introduced payload_offset and payload_length fields to aid in locating the payload.
Protocol 2.09	(Kernel 2.6.26) Added a field of 64-bit physical pointer to single linked list of struct setup_data.
Protocol 2.10	(Kernel 2.6.31) Added a protocol for relaxed alignment beyond the kernel_alignment added, new init_size and pref_address fields. Added extended boot loader IDs.
Protocol 2.11	(Kernel 3.6) Added a field for offset of EFI handover protocol entry point.
Protocol 2.12	(Kernel 3.8) Added the xloadflags field and extension fields to struct boot_params for loading bzImage and ramdisk above 4G in 64bit.
Protocol 2.13	(Kernel 3.14) Support 32- and 64-bit flags being set in xloadflags to support booting a 64-bit kernel from 32-bit EFI
Protocol 2.14	BURNT BY INCORRECT COMMIT ae7e1238e68f2a472a125673ab506d49158c1889 (x86/boot: Add ACPI RSDP address to setup_header) DO NOT USE!!! ASSUME SAME AS 2.13.
Protocol 2.15	(Kernel 5.5) Added the kernel_info and kernel_info.setup_type_max.

**Note:** The protocol version number should be changed only if the setup header is changed. There is no need to update the version number if boot\_params or kernel\_info are changed. Additionally, it is recommended to use xloadflags (in this case the protocol version number should not be updated either) or kernel\_info to communicate supported Linux kernel features to the boot loader. Due to very limited space available in the original setup header every update to it should be considered with great care. Starting from the protocol 2.15 the primary way to communicate things to the boot loader is the kernel\_info.

### 1.1 Memory Layout

The traditional memory map for the kernel loader, used for Image or zImage kernels, typically looks like:

0A0000				
	+-----+			
		Reserved for BIOS		Do not use. Reserved for BIOS EBDA.
09A000	+-----+			
		Command line		
		Stack/heap		For use by the kernel real-mode code.
098000	+-----+			
		Kernel setup		The kernel real-mode code.
090200	+-----+			
		Kernel boot sector		The kernel legacy boot sector.
090000	+-----+			
		Protected-mode kernel		The bulk of the kernel image.
010000	+-----+			
		Boot loader		<- Boot sector entry point 0000:7C00
001000	+-----+			
		Reserved for MBR/BIOS		
000800	+-----+			
		Typically used by MBR		
000600	+-----+			
		BIOS use only		
000000	+-----+			

When using bzImage, the protected-mode kernel was relocated to 0x100000 (“high memory”), and the kernel real-mode block (boot sector, setup, and stack/heap) was made relocatable to any address between 0x10000 and end of low memory. Unfortunately, in protocols 2.00 and 2.01 the 0x90000+ memory range is still used internally by the kernel; the 2.02 protocol resolves that problem.

It is desirable to keep the “memory ceiling” - the highest point in low memory touched by the boot loader - as low as possible, since some newer BIOSes have begun to allocate some rather large amounts of memory, called the Extended BIOS Data Area, near the top of low memory. The boot loader should use the “INT 12h” BIOS call to verify how much low memory is available.

Unfortunately, if INT 12h reports that the amount of memory is too low, there is usually nothing the boot loader can do but to report an error to the user. The boot loader should therefore be designed to take up as little space in low memory as it reasonably can. For zImage or old bzImage kernels, which need data written into the 0x90000 segment, the boot loader should make sure not to use memory above the 0x9A000 point; too many BIOSes will break above that point.

For a modern bzImage kernel with boot protocol version >= 2.02, a memory layout like the following is suggested:

	~		Protected-mode kernel		~
100000		+-----+			
			I/O memory hole		
0A0000		+-----+			

	Reserved for BIOS	Leave as much as possible unused
	~	
	Command line	(Can also be below the X+10000_
→mark)		
X+10000	+-----+	
	Stack/heap	For use by the kernel real-mode_
→code.		
X+08000	+-----+	
	Kernel setup	The kernel real-mode code.
	Kernel boot sector	The kernel legacy boot sector.
X	+-----+	
	Boot loader	<- Boot sector entry point_
→0000:7C00		
001000	+-----+	
	Reserved for MBR/BIOS	
000800	+-----+	
	Typically used by MBR	
000600	+-----+	
	BIOS use only	
000000	+-----+	
... where the address X is as low as the design of the boot loader permits.		

## 1.2 The Real-Mode Kernel Header

In the following text, and anywhere in the kernel boot sequence, “a sector” refers to 512 bytes. It is independent of the actual sector size of the underlying medium.

The first step in loading a Linux kernel should be to load the real-mode code (boot sector and setup code) and then examine the following header at offset 0x01f1. The real-mode code can total up to 32K, although the boot loader may choose to load only the first two sectors (1K) and then examine the bootup sector size.

The header looks like:

Offset/Size	Proto	Name	Meaning
01F1/1	ALL(1)	setup_sects	The size of the setup in sectors
01F2/2	ALL	root_flags	If set, the root is mounted readonly
01F4/4	2.04+(2)	syssize	The size of the 32-bit code in 16-byte paras
01F8/2	ALL	ram_size	DO NOT USE - for bootsect.S use only
01FA/2	ALL	vid_mode	Video mode control
01FC/2	ALL	root_dev	Default root device number
01FE/2	ALL	boot_flag	0xAA55 magic number
0200/2	2.00+	jump	Jump instruction
0202/4	2.00+	header	Magic signature “HdrS”
0206/2	2.00+	version	Boot protocol version supported
0208/4	2.00+	realmode_swch	Boot loader hook (see below)
020C/2	2.00+	start_sys_seg	The load-low segment (0x1000) (obsolete)

continues on next

Table 1 – continued from previous page

Offset/Size	Proto	Name	Meaning
020E/2	2.00+	kernel_version	Pointer to kernel version string
0210/1	2.00+	type_of_loader	Boot loader identifier
0211/1	2.00+	loadflags	Boot protocol option flags
0212/2	2.00+	setup_move_size	Move to high memory size (used with hooks)
0214/4	2.00+	code32_start	Boot loader hook (see below)
0218/4	2.00+	ramdisk_image	initrd load address (set by boot loader)
021C/4	2.00+	ramdisk_size	initrd size (set by boot loader)
0220/4	2.00+	bootsect_kludge	DO NOT USE - for bootsect.S use only
0224/2	2.01+	heap_end_ptr	Free memory after setup end
0226/1	2.02+(3)	ext_loader_ver	Extended boot loader version
0227/1	2.02+(3)	ext_loader_type	Extended boot loader ID
0228/4	2.02+	cmd_line_ptr	32-bit pointer to the kernel command line
022C/4	2.03+	initrd_addr_max	Highest legal initrd address
0230/4	2.05+	kernel_alignment	Physical addr alignment required for kernel
0234/1	2.05+	relocatable_kernel	Whether kernel is relocatable or not
0235/1	2.10+	min_alignment	Minimum alignment, as a power of two
0236/2	2.12+	xloadflags	Boot protocol option flags
0238/4	2.06+	cmdline_size	Maximum size of the kernel command line
023C/4	2.07+	hardware_subarch	Hardware subarchitecture
0240/8	2.07+	hardware_subarch_data	Subarchitecture-specific data
0248/4	2.08+	payload_offset	Offset of kernel payload
024C/4	2.08+	payload_length	Length of kernel payload
0250/8	2.09+	setup_data	64-bit physical pointer to linked list of struct setup_data
0258/8	2.10+	pref_address	Preferred loading address
0260/4	2.10+	init_size	Linear memory required during initialization
0264/4	2.11+	handover_offset	Offset of handover entry point
0268/4	2.15+	kernel_info_offset	Offset of the kernel_info

**Note:**

- (1) For backwards compatibility, if the setup\_sects field contains 0, the real value is 4.
- (2) For boot protocol prior to 2.04, the upper two bytes of the syssize field are unusable, which means the size of a bzImage kernel cannot be determined.
- (3) Ignored, but safe to set, for boot protocols 2.02-2.09.

If the “HdrS” (0x53726448) magic number is not found at offset 0x202, the boot protocol version is “old”. Loading an old kernel, the following parameters should be assumed:

```
Image type = zImage
initrd not supported
Real-mode kernel must be located at 0x90000.
```

Otherwise, the “version” field contains the protocol version, e.g. protocol version 2.01 will contain 0x0201 in this field. When setting fields in the header, you must make sure only to set fields supported by the protocol version in use.

## 1.3 Details of Header Fields

For each field, some are information from the kernel to the bootloader (“read”), some are expected to be filled out by the bootloader (“write”), and some are expected to be read and modified by the bootloader (“modify”).

All general purpose boot loaders should write the fields marked (obligatory). Boot loaders who want to load the kernel at a nonstandard address should fill in the fields marked (reloc); other boot loaders can ignore those fields.

The byte order of all fields is littleendian (this is x86, after all.)

Field name:	setup_sects
Type:	read
Offset/size:	0x1f1/1
Protocol:	ALL

The size of the setup code in 512-byte sectors. If this field is 0, the real value is 4. The real-mode code consists of the boot sector (always one 512-byte sector) plus the setup code.

Field name:	root_flags
Type:	modify (optional)
Offset/size:	0x1f2/2
Protocol:	ALL

If this field is nonzero, the root defaults to readonly. The use of this field is deprecated; use the “ro” or “rw” options on the command line instead.

Field name:	syssize
Type:	read
Offset/size:	0x1f4/4 (protocol 2.04+) 0x1f4/2 (protocol ALL)
Protocol:	2.04+

The size of the protected-mode code in units of 16-byte paragraphs. For protocol versions older than 2.04 this field is only two bytes wide, and therefore cannot be trusted for the size of a kernel if the LOAD\_HIGH flag is set.

Field name:	ram_size
Type:	kernel internal
Offset/size:	0x1f8/2
Protocol:	ALL

This field is obsolete.

Field name:	vid_mode
Type:	modify (obligatory)
Offset/size:	0x1fa/2

Please see the section on SPECIAL COMMAND LINE OPTIONS.

Field name:	root_dev
Type:	modify (optional)
Offset/size:	0x1fc/2
Protocol:	ALL

The default root device device number. The use of this field is deprecated, use the “root=” option on the command line instead.

Field name:	boot_flag
Type:	read
Offset/size:	0x1fe/2
Protocol:	ALL

Contains 0xAA55. This is the closest thing old Linux kernels have to a magic number.

Field name:	jump
Type:	read
Offset/size:	0x200/2
Protocol:	2.00+

Contains an x86 jump instruction, 0xEB followed by a signed offset relative to byte 0x202. This can be used to determine the size of the header.

Field name:	header
Type:	read
Offset/size:	0x202/4
Protocol:	2.00+

Contains the magic number “HdrS” (0x53726448).

Field name:	version
Type:	read
Offset/size:	0x206/2
Protocol:	2.00+

Contains the boot protocol version, in (major << 8)+minor format, e.g. 0x0204 for version 2.04, and 0x0a11 for a hypothetical version 10.17.

Field name:	realmode_swch
Type:	modify (optional)
Offset/size:	0x208/4
Protocol:	2.00+

Boot loader hook (see ADVANCED BOOT LOADER HOOKS below.)

Field name:	start_sys_seg
Type:	read
Offset/size:	0x20c/2
Protocol:	2.00+

The load low segment (0x1000). Obsolete.

Field name:	kernel_version
Type:	read
Offset/size:	0x20e/2
Protocol:	2.00+

If set to a nonzero value, contains a pointer to a NUL-terminated human-readable kernel version number string, less 0x200. This can be used to display the kernel version to the user. This value should be less than (0x200\*setup\_sects).

For example, if this value is set to 0x1c00, the kernel version number string can be found at offset 0x1e00 in the kernel file. This is a valid value if and only if the “setup\_sects” field contains the value 15 or higher, as:

```
0x1c00 < 15*0x200 (= 0x1e00) but
0x1c00 >= 14*0x200 (= 0x1c00)
```

```
0x1c00 >> 9 = 14, So the minimum value for setup_secs is 15.
```

Field name:	type_of_loader
Type:	write (obligatory)
Offset/size:	0x210/1
Protocol:	2.00+

If your boot loader has an assigned id (see table below), enter 0xTV here, where T is an identifier for the boot loader and V is a version number. Otherwise, enter 0xFF here.

For boot loader IDs above T = 0xD, write T = 0xE to this field and write the extended ID minus 0x10 to the ext\_loader\_type field. Similarly, the ext\_loader\_ver field can be used to provide more than four bits for the bootloader version.

For example, for T = 0x15, V = 0x234, write:

```
type_of_loader <- 0xE4
ext_loader_type <- 0x05
ext_loader_ver <- 0x23
```

Assigned boot loader ids (hexadecimal):



0	LILO (0x00 reserved for pre-2.00 bootloader)
1	Loadlin
2	bootsect-loader (0x20, all other values reserved)
3	Syslinux
4	Etherboot/gPXE/iPXE
5	ELILO
7	GRUB
8	U-Boot
9	Xen
A	Gujin
B	Qemu
C	Arcturus Networks uCbootloader
D	kexec-tools
E	Extended (see ext_loader_type)
F	Special (0xFF = undefined)
10	Reserved
11	Minimal Linux Bootloader < <a href="http://sebastian-plotz.blogspot.de">http://sebastian-plotz.blogspot.de</a> >
12	OVMF UEFI virtualization stack

Please contact <[hpa@zytor.com](mailto:hpa@zytor.com)> if you need a bootloader ID value assigned.

Field name:	loadflags
Type:	modify (obligatory)
Offset/size:	0x211/1
Protocol:	2.00+

This field is a bitmask.

Bit 0 (read): LOADED\_HIGH

- If 0, the protected-mode code is loaded at 0x10000.
- If 1, the protected-mode code is loaded at 0x100000.

Bit 1 (kernel internal): KASLR\_FLAG

- Used internally by the compressed kernel to communicate KASLR status to kernel proper.
  - If 1, KASLR enabled.
  - If 0, KASLR disabled.

Bit 5 (write): QUIET\_FLAG

- If 0, print early messages.
- If 1, suppress early messages.

This requests to the kernel (decompressor and early kernel) to not write early messages that require accessing the display hardware directly.

Bit 6 (obsolete): KEEP\_SEGMENTS

Protocol: 2.07+

- This flag is obsolete.

Bit 7 (write): CAN\_USE\_HEAP

Set this bit to 1 to indicate that the value entered in the `heap_end_ptr` is valid. If this field is clear, some setup code functionality will be disabled.

Field name:	setup_move_size
Type:	modify (obligatory)
Offset/size:	0x212/2
Protocol:	2.00-2.01

When using protocol 2.00 or 2.01, if the real mode kernel is not loaded at 0x90000, it gets moved there later in the loading sequence. Fill in this field if you want additional data (such as the kernel command line) moved in addition to the real-mode kernel itself.

The unit is bytes starting with the beginning of the boot sector.

This field is can be ignored when the protocol is 2.02 or higher, or if the real-mode code is loaded at 0x90000.

Field name:	code32_start
Type:	modify (optional, reloc)
Offset/size:	0x214/4
Protocol:	2.00+

The address to jump to in protected mode. This defaults to the load address of the kernel, and can be used by the boot loader to determine the proper load address.

This field can be modified for two purposes:

1. as a boot loader hook (see Advanced Boot Loader Hooks below.)
2. if a bootloader which does not install a hook loads a relocatable kernel at a non-standard address it will have to modify this field to point to the load address.

Field name:	ramdisk_image
Type:	write (obligatory)
Offset/size:	0x218/4
Protocol:	2.00+

The 32-bit linear address of the initial ramdisk or ramfs. Leave at zero if there is no initial ramdisk/ramfs.

Field name:	ramdisk_size
Type:	write (obligatory)
Offset/size:	0x21c/4
Protocol:	2.00+

Size of the initial ramdisk or ramfs. Leave at zero if there is no initial ramdisk/ramfs.

Field name:	bootsect_kludge
Type:	kernel internal
Offset/size:	0x220/4
Protocol:	2.00+

This field is obsolete.

Field name:	heap_end_ptr
Type:	write (obligatory)
Offset/size:	0x224/2
Protocol:	2.01+

Set this field to the offset (from the beginning of the real-mode code) of the end of the setup stack/heap, minus 0x0200.

Field name:	ext_loader_ver
Type:	write (optional)
Offset/size:	0x226/1
Protocol:	2.02+

This field is used as an extension of the version number in the `type_of_loader` field. The total version number is considered to be  $(\text{type\_of\_loader} \& 0x0f) + (\text{ext\_loader\_ver} < 4)$ .

The use of this field is boot loader specific. If not written, it is zero.

Kernels prior to 2.6.31 did not recognize this field, but it is safe to write for protocol version 2.02 or higher.

Field name:	ext_loader_type
Type:	write (obligatory if $(\text{type\_of\_loader} \& 0xf0) == 0xe0$ )
Offset/size:	0x227/1
Protocol:	2.02+

This field is used as an extension of the type number in `type_of_loader` field. If the type in `type_of_loader` is 0xE, then the actual type is  $(\text{ext\_loader\_type} + 0x10)$ .

This field is ignored if the type in `type_of_loader` is not 0xE.

Kernels prior to 2.6.31 did not recognize this field, but it is safe to write for protocol version 2.02 or higher.

Field name:	cmd_line_ptr
Type:	write (obligatory)
Offset/size:	0x228/4
Protocol:	2.02+

Set this field to the linear address of the kernel command line. The kernel command line can be located anywhere between the end of the setup heap and 0xA0000; it does not have to be located in the same 64K segment as the real-mode code itself.

Fill in this field even if your boot loader does not support a command line, in which case you can point this to an empty string (or better yet, to the string "auto".) If this field is left at zero, the kernel will assume that your boot loader does not support the 2.02+ protocol.

Field name:	initrd_addr_max
Type:	read
Offset/size:	0x22c/4
Protocol:	2.03+

The maximum address that may be occupied by the initial ramdisk/ramfs contents. For boot protocols 2.02 or earlier, this field is not present, and the maximum address is 0x37FFFFFF. (This address is defined as the address of the highest safe byte, so if your ramdisk is exactly 131072 bytes long and this field is 0x37FFFFFF, you can start your ramdisk at 0x37FE0000.)

Field name:	kernel_alignment
Type:	read/modify (reloc)
Offset/size:	0x230/4
Protocol:	2.05+ (read), 2.10+ (modify)

Alignment unit required by the kernel (if `relocatable_kernel` is true.) A relocatable kernel that is loaded at an alignment incompatible with the value in this field will be realigned during kernel initialization.

Starting with protocol version 2.10, this reflects the kernel alignment preferred for optimal performance; it is possible for the loader to modify this field to permit a lesser alignment. See the `min_alignment` and `pref_address` field below.

Field name:	relocatable_kernel
Type:	read (reloc)
Offset/size:	0x234/1
Protocol:	2.05+

If this field is nonzero, the protected-mode part of the kernel can be loaded at any address that satisfies the `kernel_alignment` field. After loading, the boot loader must set the `code32_start` field to point to the loaded code, or to a boot loader hook.

Field name:	min_alignment
Type:	read (reloc)
Offset/size:	0x235/1
Protocol:	2.10+

This field, if nonzero, indicates as a power of two the minimum alignment required, as opposed to preferred, by the kernel to boot. If a boot loader makes use of this field, it should update the `kernel_alignment` field with the alignment unit desired; typically:

```
kernel_alignment = 1 << min_alignment
```

There may be a considerable performance cost with an excessively misaligned kernel. Therefore, a loader should typically try each power-of-two alignment from `kernel_alignment` down to this alignment.

Field name:	xloadflags
Type:	read
Offset/size:	0x236/2
Protocol:	2.12+

This field is a bitmask.

Bit 0 (read): XLF\_KERNEL\_64

- If 1, this kernel has the legacy 64-bit entry point at 0x200.

Bit 1 (read): XLF\_CAN\_BE\_LOADED\_ABOVE\_4G

- If 1, kernel/boot\_params/cmdline/ramdisk can be above 4G.

Bit 2 (read): XLF\_EFI\_HANDOVER\_32

- If 1, the kernel supports the 32-bit EFI handoff entry point given at handover\_offset.

Bit 3 (read): XLF\_EFI\_HANDOVER\_64

- If 1, the kernel supports the 64-bit EFI handoff entry point given at handover\_offset + 0x200.

Bit 4 (read): XLF\_EFI\_KEXEC

- If 1, the kernel supports kexec EFI boot with EFI runtime support.

Field name:	cmdline_size
Type:	read
Offset/size:	0x238/4
Protocol:	2.06+

The maximum size of the command line without the terminating zero. This means that the command line can contain at most cmdline\_size characters. With protocol version 2.05 and earlier, the maximum size was 255.

Field name:	hardware_subarch
Type:	write (optional, defaults to x86/PC)
Offset/size:	0x23c/4
Protocol:	2.07+

In a paravirtualized environment the hardware low level architectural pieces such as interrupt handling, page table handling, and accessing process control registers needs to be done differently.

This field allows the bootloader to inform the kernel we are in one of those environments.

0x00000000	The default x86/PC environment
0x00000001	lguest
0x00000002	Xen
0x00000003	Moorestown MID
0x00000004	CE4100 TV Platform

Field name:	hardware_subarch_data
Type:	write (subarch-dependent)
Offset/size:	0x240/8
Protocol:	2.07+

A pointer to data that is specific to hardware subarch. This field is currently unused for the default x86/PC environment, do not modify.

Field name:	payload_offset
Type:	read
Offset/size:	0x248/4
Protocol:	2.08+

If non-zero then this field contains the offset from the beginning of the protected-mode code to the payload.

The payload may be compressed. The format of both the compressed and uncompressed data should be determined using the standard magic numbers. The currently supported compression formats are gzip (magic numbers 1F 8B or 1F 9E), bzip2 (magic number 42 5A), LZMA (magic number 5D 00), XZ (magic number FD 37), LZ4 (magic number 02 21) and ZSTD (magic number 28 B5). The uncompressed payload is currently always ELF (magic number 7F 45 4C 46).

Field name:	payload_length
Type:	read
Offset/size:	0x24c/4
Protocol:	2.08+

The length of the payload.

Field name:	setup_data
Type:	write (special)
Offset/size:	0x250/8
Protocol:	2.09+

The 64-bit physical pointer to NULL terminated single linked list of struct `setup_data`. This is used to define a more extensible boot parameters passing mechanism. The definition of struct `setup_data` is as follow:

```
struct setup_data {
    u64 next;
    u32 type;
    u32 len;
    u8  data[0];
};
```

Where, the next is a 64-bit physical pointer to the next node of linked list, the next field of the last node is 0; the type is used to identify the contents of data; the len is the length of data field; the data holds the real payload.

This list may be modified at a number of points during the bootup process. Therefore,

when modifying this list one should always make sure to consider the case where the linked list already contains entries.

The `setup_data` is a bit awkward to use for extremely large data objects, both because the `setup_data` header has to be adjacent to the data object and because it has a 32-bit length field. However, it is important that intermediate stages of the boot process have a way to identify which chunks of memory are occupied by kernel data.

Thus `setup_indirect` struct and `SETUP_INDIRECT` type were introduced in protocol 2.15:

```
struct setup_indirect {
    __u32 type;
    __u32 reserved; /* Reserved, must be set to zero. */
    __u64 len;
    __u64 addr;
};
```

The type member is a `SETUP_INDIRECT | SETUP_*` type. However, it cannot be `SETUP_INDIRECT` itself since making the `setup_indirect` a tree structure could require a lot of stack space in something that needs to parse it and stack space can be limited in boot contexts.

Let's give an example how to point to `SETUP_E820_EXT` data using `setup_indirect`. In this case `setup_data` and `setup_indirect` will look like this:

```
struct setup_data {
    __u64 next = 0 or <addr_of_next_setup_data_struct>;
    __u32 type = SETUP_INDIRECT;
    __u32 len = sizeof(setup_indirect);
    __u8 data[sizeof(setup_indirect)] = struct setup_indirect {
        __u32 type = SETUP_INDIRECT | SETUP_E820_EXT;
        __u32 reserved = 0;
        __u64 len = <len_of_SETUP_E820_EXT_data>;
        __u64 addr = <addr_of_SETUP_E820_EXT_data>;
    }
}
```

---

**Note:** `SETUP_INDIRECT | SETUP_NONE` objects cannot be properly distinguished from `SETUP_INDIRECT` itself. So, this kind of objects cannot be provided by the bootloaders.

---

Field name:	<code>pref_address</code>
Type:	read (reloc)
Offset/size:	0x258/8
Protocol:	2.10+

This field, if nonzero, represents a preferred load address for the kernel. A relocating bootloader should attempt to load at this address if possible.

A non-relocatable kernel will unconditionally move itself and to run at this address.

Field name:	init_size
Type:	read
Offset/size:	0x260/4

This field indicates the amount of linear contiguous memory starting at the kernel runtime start address that the kernel needs before it is capable of examining its memory map. This is not the same thing as the total amount of memory the kernel needs to boot, but it can be used by a relocating boot loader to help select a safe load address for the kernel.

The kernel runtime start address is determined by the following algorithm:

```
if (relocatable_kernel)
runtime_start = align_up(load_address, kernel_alignment)
else
runtime_start = pref_address
```

Field name:	handover_offset
Type:	read
Offset/size:	0x264/4

This field is the offset from the beginning of the kernel image to the EFI handover protocol entry point. Boot loaders using the EFI handover protocol to boot the kernel should jump to this offset.

See EFI HANDOVER PROTOCOL below for more details.

Field name:	kernel_info_offset
Type:	read
Offset/size:	0x268/4
Protocol:	2.15+

This field is the offset from the beginning of the kernel image to the `kernel_info`. The `kernel_info` structure is embedded in the Linux image in the uncompressed protected mode region.

## 1.4 The kernel\_info

The relationships between the headers are analogous to the various data sections:

```
setup_header = .data boot_params/setup_data = .bss
```

What is missing from the above list? That's right:

```
kernel_info = .rodata
```

We have been (ab)using `.data` for things that could go into `.rodata` or `.bss` for a long time, for lack of alternatives and – especially early on – inertia. Also, the BIOS stub is responsible for creating `boot_params`, so it isn't available to a BIOS-based loader (`setup_data` is, though).

`setup_header` is permanently limited to 144 bytes due to the reach of the 2-byte jump field, which doubles as a length field for the structure, combined with the size of the “hole” in struct



boot\_params that a protected-mode loader or the BIOS stub has to copy it into. It is currently 119 bytes long, which leaves us with 25 very precious bytes. This isn't something that can be fixed without revising the boot protocol entirely, breaking backwards compatibility.

boot\_params proper is limited to 4096 bytes, but can be arbitrarily extended by adding setup\_data entries. It cannot be used to communicate properties of the kernel image, because it is .bss and has no image-provided content.

kernel\_info solves this by providing an extensible place for information about the kernel image. It is readonly, because the kernel cannot rely on a bootloader copying its contents anywhere, but that is OK; if it becomes necessary it can still contain data items that an enabled bootloader would be expected to copy into a setup\_data chunk.

All kernel\_info data should be part of this structure. Fixed size data have to be put before kernel\_info\_var\_len\_data label. Variable size data have to be put after kernel\_info\_var\_len\_data label. Each chunk of variable size data has to be prefixed with header/magic and its size, e.g.:

```
kernel_info:
    .ascii  "LToP"          /* Header, Linux top (structure). */
    .long   kernel_info_var_len_data - kernel_info
    .long   kernel_info_end - kernel_info
    .long   0x01234567      /* Some fixed size data for the bootloaders. */
kernel_info_var_len_data:
example_struct:                /* Some variable size data for the bootloaders.
→ */
    .ascii  "0123"          /* Header/Magic. */
    .long   example_struct_end - example_struct
    .ascii  "Struct"
    .long   0x89012345
example_struct_end:
example_strings:               /* Some variable size data for the bootloaders.
→ */
    .ascii  "ABCD"          /* Header/Magic. */
    .long   example_strings_end - example_strings
    .asciz  "String_0"
    .asciz  "String_1"
example_strings_end:
kernel_info_end:
```

This way the kernel\_info is self-contained blob.

---

**Note:** Each variable size data header/magic can be any 4-character string, without 0 at the end of the string, which does not collide with existing variable length data headers/magics.

---

## 1.5 Details of the kernel\_info Fields

Field name:	header
Offset/size:	0x0000/4

Contains the magic number “LToP” (0x506f544c).

Field name:	size
Offset/size:	0x0004/4

This field contains the size of the kernel\_info including kernel\_info.header. It does not count kernel\_info.kernel\_info\_var\_len\_data size. This field should be used by the bootloaders to detect supported fixed size fields in the kernel\_info and beginning of kernel\_info.kernel\_info\_var\_len\_data.

Field name:	size_total
Offset/size:	0x0008/4

This field contains the size of the kernel\_info including kernel\_info.header and kernel\_info.kernel\_info\_var\_len\_data.

Field name:	setup_type_max
Offset/size:	0x000c/4

This field contains maximal allowed type for setup\_data and setup\_indirect structs.

## 1.6 The Image Checksum

From boot protocol version 2.08 onwards the CRC-32 is calculated over the entire file using the characteristic polynomial 0x04C11DB7 and an initial remainder of 0xffffffff. The checksum is appended to the file; therefore the CRC of the file up to the limit specified in the syssize field of the header is always 0.

## 1.7 The Kernel Command Line

The kernel command line has become an important way for the boot loader to communicate with the kernel. Some of its options are also relevant to the boot loader itself, see “special command line options” below.

The kernel command line is a null-terminated string. The maximum length can be retrieved from the field cmdline\_size. Before protocol version 2.06, the maximum was 255 characters. A string that is too long will be automatically truncated by the kernel.

If the boot protocol version is 2.02 or later, the address of the kernel command line is given by the header field cmd\_line\_ptr (see above.) This address can be anywhere between the end of the setup heap and 0xA0000.

If the protocol version is *not* 2.02 or higher, the kernel command line is entered using the following protocol:

- At offset 0x0020 (word), “cmd\_line\_magic”, enter the magic number 0xA33F.
- At offset 0x0022 (word), “cmd\_line\_offset”, enter the offset of the kernel command line (relative to the start of the real-mode kernel).
- The kernel command line *must* be within the memory region covered by setup\_move\_size, so you may need to adjust this field.

## 1.8 Memory Layout of The Real-Mode Code

The real-mode code requires a stack/heap to be set up, as well as memory allocated for the kernel command line. This needs to be done in the real-mode accessible memory in bottom megabyte.

It should be noted that modern machines often have a sizable Extended BIOS Data Area (EBDA). As a result, it is advisable to use as little of the low megabyte as possible.

Unfortunately, under the following circumstances the 0x90000 memory segment has to be used:

- When loading a zImage kernel ((loadflags & 0x01) == 0).
- When loading a 2.01 or earlier boot protocol kernel.

---

**Note:** For the 2.00 and 2.01 boot protocols, the real-mode code can be loaded at another address, but it is internally relocated to 0x90000. For the “old” protocol, the real-mode code must be loaded at 0x90000.

---

When loading at 0x90000, avoid using memory above 0x9a000.

For boot protocol 2.02 or higher, the command line does not have to be located in the same 64K segment as the real-mode setup code; it is thus permitted to give the stack/heap the full 64K segment and locate the command line above it.

The kernel command line should not be located below the real-mode code, nor should it be located in high memory.

## 1.9 Sample Boot Configuration

As a sample configuration, assume the following layout of the real mode segment.

When loading below 0x90000, use the entire segment:

0x0000-0x7fff	Real mode kernel
0x8000-0xdfff	Stack and heap
0xe000-0xffff	Kernel command line

When loading at 0x90000 OR the protocol version is 2.01 or earlier:

0x0000-0x7fff	Real mode kernel
0x8000-0x97ff	Stack and heap
0x9800-0x9fff	Kernel command line

Such a boot loader should enter the following fields in the header:

```
unsigned long base_ptr; /* base address for real-mode segment */

if ( setup_sects == 0 ) {
    setup_sects = 4;
}

if ( protocol >= 0x0200 ) {
    type_of_loader = <type code>;
    if ( loading_initrd ) {
        ramdisk_image = <initrd_address>;
        ramdisk_size = <initrd_size>;
    }

    if ( protocol >= 0x0202 && loadflags & 0x01 )
        heap_end = 0xe000;
    else
        heap_end = 0x9800;

    if ( protocol >= 0x0201 ) {
        heap_end_ptr = heap_end - 0x200;
        loadflags |= 0x80; /* CAN_USE_HEAP */
    }

    if ( protocol >= 0x0202 ) {
        cmd_line_ptr = base_ptr + heap_end;
        strcpy(cmd_line_ptr, cmdline);
    } else {
        cmd_line_magic = 0xA33F;
        cmd_line_offset = heap_end;
        setup_move_size = heap_end + strlen(cmdline)+1;
        strcpy(base_ptr+cmd_line_offset, cmdline);
    }
} else {
    /* Very old kernel */

    heap_end = 0x9800;

    cmd_line_magic = 0xA33F;
    cmd_line_offset = heap_end;

    /* A very old kernel MUST have its real-mode code
       loaded at 0x90000 */

    if ( base_ptr != 0x90000 ) {
        /* Copy the real-mode kernel */
        memcpy(0x90000, base_ptr, (setup_sects+1)*512);
        base_ptr = 0x90000; /* Relocated */
    }
}
```

```

strcpy(0x90000+cmd_line_offset, cmdline);

/* It is recommended to clear memory up to the 32K mark */
memset(0x90000 + (setup_sects+1)*512, 0,
       (64-(setup_sects+1))*512);
}

```

## 1.10 Loading The Rest of The Kernel

The 32-bit (non-real-mode) kernel starts at offset  $(\text{setup\_sects}+1)*512$  in the kernel file (again, if  $\text{setup\_sects} == 0$  the real value is 4.) It should be loaded at address 0x10000 for Image/zImage kernels and 0x100000 for bzImage kernels.

The kernel is a bzImage kernel if the protocol  $\geq 2.00$  and the 0x01 bit (LOAD\_HIGH) in the loadflags field is set:

```

is_bzImage = (protocol >= 0x0200) && (loadflags & 0x01);
load_address = is_bzImage ? 0x100000 : 0x10000;

```

Note that Image/zImage kernels can be up to 512K in size, and thus use the entire 0x10000-0x90000 range of memory. This means it is pretty much a requirement for these kernels to load the real-mode part at 0x90000. bzImage kernels allow much more flexibility.

## 1.11 Special Command Line Options

If the command line provided by the boot loader is entered by the user, the user may expect the following command line options to work. They should normally not be deleted from the kernel command line even though not all of them are actually meaningful to the kernel. Boot loader authors who need additional command line options for the boot loader itself should get them registered in Documentation/admin-guide/kernel-parameters.rst to make sure they will not conflict with actual kernel options now or in the future.

**vga=<mode>** <mode> here is either an integer (in C notation, either decimal, octal, or hexadecimal) or one of the strings “normal” (meaning 0xFFFF), “ext” (meaning 0xFFFE) or “ask” (meaning 0xFFFD). This value should be entered into the vid\_mode field, as it is used by the kernel before the command line is parsed.

**mem=<size>** <size> is an integer in C notation optionally followed by (case insensitive) K, M, G, T, P or E (meaning << 10, << 20, << 30, << 40, << 50 or << 60). This specifies the end of memory to the kernel. This affects the possible placement of an initrd, since an initrd should be placed near end of memory. Note that this is an option to *both* the kernel and the bootloader!

**initrd=<file>** An initrd should be loaded. The meaning of <file> is obviously bootloader-dependent, and some boot loaders (e.g. LILO) do not have such a command.

In addition, some boot loaders add the following options to the user-specified command line:

**BOOT\_IMAGE=<file>** The boot image which was loaded. Again, the meaning of <file> is obviously bootloader-dependent.

**auto** The kernel was booted without explicit user intervention.

If these options are added by the boot loader, it is highly recommended that they are located *first*, before the user-specified or configuration-specified command line. Otherwise, “init=/bin/sh” gets confused by the “auto” option.

## 1.12 Running the Kernel

The kernel is started by jumping to the kernel entry point, which is located at *segment* offset 0x20 from the start of the real mode kernel. This means that if you loaded your real-mode kernel code at 0x90000, the kernel entry point is 9020:0000.

At entry, ds = es = ss should point to the start of the real-mode kernel code (0x9000 if the code is loaded at 0x90000), sp should be set up properly, normally pointing to the top of the heap, and interrupts should be disabled. Furthermore, to guard against bugs in the kernel, it is recommended that the boot loader sets fs = gs = ds = es = ss.

In our example from above, we would do:

```
/* Note: in the case of the "old" kernel protocol, base_ptr must
   be == 0x90000 at this point; see the previous sample code */

seg = base_ptr >> 4;

cli(); /* Enter with interrupts disabled! */

/* Set up the real-mode kernel stack */
_SS = seg;
_SP = heap_end;

_DS = _ES = _FS = _GS = seg;
jmp_far(seg+0x20, 0); /* Run the kernel */
```

If your boot sector accesses a floppy drive, it is recommended to switch off the floppy motor before running the kernel, since the kernel boot leaves interrupts off and thus the motor will not be switched off, especially if the loaded kernel has the floppy driver as a demand-loaded module!

## 1.13 Advanced Boot Loader Hooks

If the boot loader runs in a particularly hostile environment (such as LOADLIN, which runs under DOS) it may be impossible to follow the standard memory location requirements. Such a boot loader may use the following hooks that, if set, are invoked by the kernel at the appropriate time. The use of these hooks should probably be considered an absolutely last resort!

IMPORTANT: All the hooks are required to preserve %esp, %ebp, %esi and %edi across invocation.

**realmode\_swch:** A 16-bit real mode far subroutine invoked immediately before entering protected mode. The default routine disables NMI, so your routine should probably do so, too.

**code32\_start:** A 32-bit flat-mode routine *jumped* to immediately after the transition to protected mode, but before the kernel is uncompressed. No segments, except CS, are guaranteed to be set up (current kernels do, but older ones do not); you should set them up to BOOT\_DS (0x18) yourself.

After completing your hook, you should jump to the address that was in this field before your boot loader overwrote it (relocated, if appropriate.)

## 1.14 32-bit Boot Protocol

For machine with some new BIOS other than legacy BIOS, such as EFI, LinuxBIOS, etc, and kexec, the 16-bit real mode setup code in kernel based on legacy BIOS can not be used, so a 32-bit boot protocol needs to be defined.

In 32-bit boot protocol, the first step in loading a Linux kernel should be to setup the boot parameters (struct boot\_params, traditionally known as “zero page”). The memory for struct boot\_params should be allocated and initialized to all zero. Then the setup header from offset 0x01f1 of kernel image on should be loaded into struct boot\_params and examined. The end of setup header can be calculated as follow:

0x0202 + byte value at offset 0x0201

In addition to read/modify/write the setup header of the struct boot\_params as that of 16-bit boot protocol, the boot loader should also fill the additional fields of the struct boot\_params as described in chapter [Zero Page](#).

After setting up the struct boot\_params, the boot loader can load the 32/64-bit kernel in the same way as that of 16-bit boot protocol.

In 32-bit boot protocol, the kernel is started by jumping to the 32-bit kernel entry point, which is the start address of loaded 32/64-bit kernel.

At entry, the CPU must be in 32-bit protected mode with paging disabled; a GDT must be loaded with the descriptors for selectors \_\_BOOT\_CS(0x10) and \_\_BOOT\_DS(0x18); both descriptors must be 4G flat segment; \_\_BOOT\_CS must have execute/read permission, and \_\_BOOT\_DS must have read/write permission; CS must be \_\_BOOT\_CS and DS, ES, SS must be \_\_BOOT\_DS; interrupt must be disabled; %esi must hold the base address of the struct boot\_params; %ebp, %edi and %ebx must be zero.

## 1.15 64-bit Boot Protocol

For machine with 64bit cpus and 64bit kernel, we could use 64bit bootloader and we need a 64-bit boot protocol.

In 64-bit boot protocol, the first step in loading a Linux kernel should be to setup the boot parameters (struct boot\_params, traditionally known as “zero page”). The memory for struct boot\_params could be allocated anywhere (even above 4G) and initialized to all zero. Then, the setup header at offset 0x01f1 of kernel image on should be loaded into struct boot\_params and examined. The end of setup header can be calculated as follows:

0x0202 + byte value at offset 0x0201

In addition to read/modify/write the setup header of the struct `boot_params` as that of 16-bit boot protocol, the boot loader should also fill the additional fields of the struct `boot_params` as described in chapter *Zero Page*.

After setting up the struct `boot_params`, the boot loader can load 64-bit kernel in the same way as that of 16-bit boot protocol, but kernel could be loaded above 4G.

In 64-bit boot protocol, the kernel is started by jumping to the 64-bit kernel entry point, which is the start address of loaded 64-bit kernel plus 0x200.

At entry, the CPU must be in 64-bit mode with paging enabled. The range with `setup_header.init_size` from start address of loaded kernel and zero page and command line buffer get ident mapping; a GDT must be loaded with the descriptors for selectors `__BOOT_CS(0x10)` and `__BOOT_DS(0x18)`; both descriptors must be 4G flat segment; `__BOOT_CS` must have execute/read permission, and `__BOOT_DS` must have read/write permission; CS must be `__BOOT_CS` and DS, ES, SS must be `__BOOT_DS`; interrupt must be disabled; `%rsi` must hold the base address of the struct `boot_params`.

### 1.16 EFI Handover Protocol (deprecated)

This protocol allows boot loaders to defer initialisation to the EFI boot stub. The boot loader is required to load the kernel/initrd(s) from the boot media and jump to the EFI handover protocol entry point which is `hdr->handover_offset` bytes from the beginning of `startup_{32,64}`.

The boot loader **MUST** respect the kernel's PE/COFF metadata when it comes to section alignment, the memory footprint of the executable image beyond the size of the file itself, and any other aspect of the PE/COFF header that may affect correct operation of the image as a PE/COFF binary in the execution context provided by the EFI firmware.

The function prototype for the handover entry point looks like this:

```
efi_main(void *handle, efi_system_table_t *table, struct boot_params *bp)
```

'handle' is the EFI image handle passed to the boot loader by the EFI firmware, 'table' is the EFI system table - these are the first two arguments of the "handoff state" as described in section 2.3 of the UEFI specification. 'bp' is the boot loader-allocated boot params.

The boot loader *must* fill out the following fields in bp:

```
- hdr.cmd_line_ptr
- hdr.ramdisk_image (if applicable)
- hdr.ramdisk_size  (if applicable)
```

All other fields should be zero.

**NOTE: The EFI Handover Protocol is deprecated in favour of the ordinary PE/COFF entry point, combined with the `LINUX_EFI_INITRD_MEDIA_GUID` based initrd loading protocol (refer to [0] for an example of the bootloader side of this), which removes the need for any knowledge on the part of the EFI bootloader regarding the internal representation of `boot_params` or any requirements/limitations regarding the placement of the command line and ramdisk in memory, or the placement of the kernel image itself.**

[0] <https://github.com/u-boot/u-boot/commit/ec80b4735a593961fe701cc3a5d717d4739b0fd0>



## DEVICETREE BOOTING

There is one single 32bit entry point to the kernel at `code32_start`, the decompressor (the real mode entry point goes to the same 32bit entry point once it switched into protected mode). That entry point supports one calling convention which is documented in *The Linux/x86 Boot Protocol*. The physical pointer to the device-tree block is passed via `setup_data` which requires at least boot protocol 2.09. The type filed is defined as

```
#define SETUP_DTB 2
```

This device-tree is used as an extension to the “boot page”. As such it does not parse / consider data which is already covered by the boot page. This includes memory size, reserved ranges, command line arguments or `initrd` address. It simply holds information which can not be retrieved otherwise like interrupt routing or a list of devices behind an I2C bus.



## **X86 FEATURE FLAGS**

### **3.1 Introduction**

On x86, flags appearing in `/proc/cpuinfo` have an `X86_FEATURE` definition in `arch/x86/include/asm/cpufeatures.h`. If the kernel cares about a feature or KVM want to expose the feature to a KVM guest, it can and should have an `X86_FEATURE_*` defined. These flags represent hardware features as well as software features.

If users want to know if a feature is available on a given system, they try to find the flag in `/proc/cpuinfo`. If a given flag is present, it means that the kernel supports it and is currently making it available. If such flag represents a hardware feature, it also means that the hardware supports it.

If the expected flag does not appear in `/proc/cpuinfo`, things are murkier. Users need to find out the reason why the flag is missing and find the way how to enable it, which is not always easy. There are several factors that can explain missing flags: the expected feature failed to enable, the feature is missing in hardware, platform firmware did not enable it, the feature is disabled at build or run time, an old kernel is in use, or the kernel does not support the feature and thus has not enabled it. In general, `/proc/cpuinfo` shows features which the kernel supports. For a full list of CPUID flags which the CPU supports, use `tools/arch/x86/kcpuid`.

### **3.2 How are feature flags created?**

#### **3.2.1 a: Feature flags can be derived from the contents of CPUID leaves.**

These feature definitions are organized mirroring the layout of CPUID leaves and grouped in words with offsets as mapped in `enum cpuid_leafs` in `cpufeatures.h` (see `arch/x86/include/asm/cpufeatures.h` for details). If a feature is defined with a `X86_FEATURE_<name>` definition in `cpufeatures.h`, and if it is detected at run time, the flags will be displayed accordingly in `/proc/cpuinfo`. For example, the flag “`avx2`” comes from `X86_FEATURE_AVX2` in `cpufeatures.h`.

### **3.2.2 b: Flags can be from scattered CPUID-based features.**

Hardware features enumerated in sparsely populated CPUID leaves get software-defined values. Still, CPUID needs to be queried to determine if a given feature is present. This is done in `init_scattered_cpuid_features()`. For instance, `X86_FEATURE_CQM_LLC` is defined as `11*32 + 0` and its presence is checked at runtime in the respective CPUID leaf [`EAX=f`, `ECX=0`] bit `EDX[1]`.

The intent of scattering CPUID leaves is to not bloat `struct cpuinfo_x86.x86_capability[]` unnecessarily. For instance, the CPUID leaf [`EAX=7`, `ECX=0`] has 30 features and is dense, but the CPUID leaf [`EAX=7`, `EAX=1`] has only one feature and would waste 31 bits of space in the `x86_capability[]` array. Since there is a `struct cpuinfo_x86` for each possible CPU, the wasted memory is not trivial.

### **3.2.3 c: Flags can be created synthetically under certain conditions for hardware features.**

Examples of conditions include whether certain features are present in `MSR_IA32_CORE_CAPS` or specific CPU models are identified. If the needed conditions are met, the features are enabled by the `set_cpu_cap` or `setup_force_cpu_cap` macros. For example, if bit 5 is set in `MSR_IA32_CORE_CAPS`, the feature `X86_FEATURE_SPLIT_LOCK_DETECT` will be enabled and “`split_lock_detect`” will be displayed. The flag “`ring3mwait`” will be displayed only when running on `INTEL_FAM6_XEON_PHI_[KNL|KNM]` processors.

### **3.2.4 d: Flags can represent purely software features.**

These flags do not represent hardware features. Instead, they represent a software feature implemented in the kernel. For example, Kernel Page Table Isolation is purely software feature and its feature flag `X86_FEATURE_PTI` is also defined in `cpufeatures.h`.

## **3.3 Naming of Flags**

The script `arch/x86/kernel/cpu/mkcapflags.sh` processes the `#define X86_FEATURE_<name>` from `cpufeatures.h` and generates the `x86_cap/bug_flags[]` arrays in `kernel/cpu/capflags.c`. The names in the resulting `x86_cap/bug_flags[]` are used to populate `/proc/cpuinfo`. The naming of flags in the `x86_cap/bug_flags[]` are as follows:

### **3.3.1 a: The name of the flag is from the string in `X86_FEATURE_<name>` by default.**

By default, the flag `<name>` in `/proc/cpuinfo` is extracted from the respective `X86_FEATURE_<name>` in `cpufeatures.h`. For example, the flag “`avx2`” is from `X86_FEATURE_AVX2`.

### 3.3.2 b: The naming can be overridden.

If the comment on the line for the `#define X86_FEATURE_*` starts with a double-quote character (`"`), the string inside the double-quote characters will be the name of the flags. For example, the flag `"sse4_1"` comes from the comment `"sse4_1"` following the `X86_FEATURE_XMM4_1` definition.

There are situations in which overriding the displayed name of the flag is needed. For instance, `/proc/cpuinfo` is a userspace interface and must remain constant. If, for some reason, the naming of `X86_FEATURE_<name>` changes, one shall override the new naming with the name already used in `/proc/cpuinfo`.

### 3.3.3 c: The naming override can be `"`, which means it will not appear in `/proc/cpuinfo`.

The feature shall be omitted from `/proc/cpuinfo` if it does not make sense for the feature to be exposed to userspace. For example, `X86_FEATURE_ALWAYS` is defined in `cpufeatures.h` but that flag is an internal kernel feature used in the alternative runtime patching functionality. So, its name is overridden with `"`. Its flag will not appear in `/proc/cpuinfo`.

## 3.4 Flags are missing when one or more of these happen

### 3.4.1 a: The hardware does not enumerate support for it.

For example, when a new kernel is running on old hardware or the feature is not enabled by boot firmware. Even if the hardware is new, there might be a problem enabling the feature at run time, the flag will not be displayed.

### 3.4.2 b: The kernel does not know about the flag.

For example, when an old kernel is running on new hardware.

### 3.4.3 c: The kernel disabled support for it at compile-time.

For example, if 5-level-paging is not enabled when building (i.e., `CONFIG_X86_5LEVEL` is not selected) the flag `"la57"` will not show up<sup>1</sup>. Even though the feature will still be detected via CPUID, the kernel disables it by clearing via `setup_clear_cpu_cap(X86_FEATURE_LA57)`.

---

<sup>1</sup> 5-level paging uses linear address of 57 bits.

### 3.4.4 d: The feature is disabled at boot-time.

A feature can be disabled either using a command-line parameter or because it failed to be enabled. The command-line parameter `clearcpuid=` can be used to disable features using the feature number as defined in `/arch/x86/include/asm/cpufeatures.h`. For instance, User Mode Instruction Protection can be disabled using `clearcpuid=514`. The number 514 is calculated from `#define X86_FEATURE_UMIP (16*32 + 2)`.

In addition, there exists a variety of custom command-line parameters that disable specific features. The list of parameters includes, but is not limited to, `nofsgsbase`, `nosgx`, `noxsave`, etc. 5-level paging can also be disabled using `"no5lvl"`.

### 3.4.5 e: The feature was known to be non-functional.

The feature was known to be non-functional because a dependency was missing at runtime. For example, AVX flags will not show up if XSAVE feature is disabled since they depend on XSAVE feature. Another example would be broken CPUs and them missing microcode patches. Due to that, the kernel decides not to enable a feature.

## **X86 TOPOLOGY**

This documents and clarifies the main aspects of x86 topology modelling and representation in the kernel. Update/change when doing changes to the respective code.

The architecture-agnostic topology definitions are in `Documentation/admin-guide/cputopology.rst`. This file holds x86-specific differences/specialities which must not necessarily apply to the generic definitions. Thus, the way to read up on Linux topology on x86 is to start with the generic one and look at this one in parallel for the x86 specifics.

Needless to say, code should use the generic functions - this file is *only* here to *document* the inner workings of x86 topology.

Started by Thomas Gleixner <[tglx@linutronix.de](mailto:tglx@linutronix.de)> and Borislav Petkov <[bp@alien8.de](mailto:bp@alien8.de)>.

The main aim of the topology facilities is to present adequate interfaces to code which needs to know/query/use the structure of the running system wrt threads, cores, packages, etc.

The kernel does not care about the concept of physical sockets because a socket has no relevance to software. It's an electromechanical component. In the past a socket always contained a single package (see below), but with the advent of Multi Chip Modules (MCM) a socket can hold more than one package. So there might be still references to sockets in the code, but they are of historical nature and should be cleaned up.

The topology of a system is described in the units of:

- packages
- cores
- threads

### **4.1 Package**

Packages contain a number of cores plus shared resources, e.g. DRAM controller, shared caches etc.

Modern systems may also use the term 'Die' for package.

AMD nomenclature for package is 'Node'.

Package-related topology information in the kernel:

- `cpuinfo_x86.x86_max_cores`:

The number of cores in a package. This information is retrieved via CPUID.

- `cpuinfo_x86.x86_max_dies`:

The number of dies in a package. This information is retrieved via CPUID.

- `cpuinfo_x86.cpu_die_id`:

The physical ID of the die. This information is retrieved via CPUID.

- `cpuinfo_x86.phys_proc_id`:

The physical ID of the package. This information is retrieved via CPUID and deduced from the APIC IDs of the cores in the package.

Modern systems use this value for the socket. There may be multiple packages within a socket. This value may differ from `cpu_die_id`.

- `cpuinfo_x86.logical_proc_id`:

The logical ID of the package. As we do not trust BIOSes to enumerate the packages in a consistent way, we introduced the concept of logical package ID so we can sanely calculate the number of maximum possible packages in the system and have the packages enumerated linearly.

- `topology_max_packages()`:

The maximum possible number of packages in the system. Helpful for per package facilities to preallocate per package information.

- `cpu_llc_id`:

A per-CPU variable containing:

- On Intel, the first APIC ID of the list of CPUs sharing the Last Level Cache
- On AMD, the Node ID or Core Complex ID containing the Last Level Cache. In general, it is a number identifying an LLC uniquely on the system.

## 4.2 Cores

A core consists of 1 or more threads. It does not matter whether the threads are SMT- or CMT-type threads.

AMDs nomenclature for a CMT core is “Compute Unit”. The kernel always uses “core”.

Core-related topology information in the kernel:

- `smp_num_siblings`:

The number of threads in a core. The number of threads in a package can be calculated by:

$$\text{threads\_per\_package} = \text{cpuinfo\_x86.x86\_max\_cores} * \text{smp\_num\_siblings}$$



## 4.3 Threads

A thread is a single scheduling unit. It's the equivalent to a logical Linux CPU.

AMDs nomenclature for CMT threads is "Compute Unit Core". The kernel always uses "thread".

Thread-related topology information in the kernel:

- `topology_core_cpumask()`:  
The cpumask contains all online threads in the package to which a thread belongs.  
The number of online threads is also printed in `/proc/cpuinfo` "siblings."
- `topology_sibling_cpumask()`:  
The cpumask contains all online threads in the core to which a thread belongs.
- `topology_logical_package_id()`:  
The logical package ID to which a thread belongs.
- `topology_physical_package_id()`:  
The physical package ID to which a thread belongs.
- `topology_core_id()`:  
The ID of the core to which a thread belongs. It is also printed in `/proc/cpuinfo` "core\_id."

## 4.4 System topology examples

**Note:** The alternative Linux CPU enumeration depends on how the BIOS enumerates the threads. Many BIOSes enumerate all threads 0 first and then all threads 1. That has the "advantage" that the logical Linux CPU numbers of threads 0 stay the same whether threads are enabled or not. That's merely an implementation detail and has no practical impact.

### 1) Single Package, Single Core:

```
[package 0] -> [core 0] -> [thread 0] -> Linux CPU 0
```

### 2) Single Package, Dual Core

#### a) One thread per core:

```
[package 0] -> [core 0] -> [thread 0] -> Linux CPU 0
              -> [core 1] -> [thread 0] -> Linux CPU 1
```

#### b) Two threads per core:

```
[package 0] -> [core 0] -> [thread 0] -> Linux CPU 0
              -> [thread 1] -> Linux CPU 1
              -> [core 1] -> [thread 0] -> Linux CPU 2
              -> [thread 1] -> Linux CPU 3
```

Alternative enumeration:

```
[package 0] -> [core 0] -> [thread 0] -> Linux CPU 0
              -> [thread 1] -> Linux CPU 2
              -> [core 1] -> [thread 0] -> Linux CPU 1
              -> [thread 1] -> Linux CPU 3
```

AMD nomenclature for CMT systems:

```
[node 0] -> [Compute Unit 0] -> [Compute Unit Core 0] -> Linux CPU 0
              -> [Compute Unit Core 1] -> Linux CPU 1
              -> [Compute Unit 1] -> [Compute Unit Core 0] -> Linux CPU 2
              -> [Compute Unit Core 1] -> Linux CPU 3
```

#### 4) Dual Package, Dual Core

##### a) One thread per core:

```
[package 0] -> [core 0] -> [thread 0] -> Linux CPU 0
              -> [core 1] -> [thread 0] -> Linux CPU 1

[package 1] -> [core 0] -> [thread 0] -> Linux CPU 2
              -> [core 1] -> [thread 0] -> Linux CPU 3
```

##### b) Two threads per core:

```
[package 0] -> [core 0] -> [thread 0] -> Linux CPU 0
              -> [thread 1] -> Linux CPU 1
              -> [core 1] -> [thread 0] -> Linux CPU 2
              -> [thread 1] -> Linux CPU 3

[package 1] -> [core 0] -> [thread 0] -> Linux CPU 4
              -> [thread 1] -> Linux CPU 5
              -> [core 1] -> [thread 0] -> Linux CPU 6
              -> [thread 1] -> Linux CPU 7
```

Alternative enumeration:

```
[package 0] -> [core 0] -> [thread 0] -> Linux CPU 0
              -> [thread 1] -> Linux CPU 4
              -> [core 1] -> [thread 0] -> Linux CPU 1
              -> [thread 1] -> Linux CPU 5

[package 1] -> [core 0] -> [thread 0] -> Linux CPU 2
              -> [thread 1] -> Linux CPU 6
              -> [core 1] -> [thread 0] -> Linux CPU 3
              -> [thread 1] -> Linux CPU 7
```

AMD nomenclature for CMT systems:

```
[node 0] -> [Compute Unit 0] -> [Compute Unit Core 0] -> Linux CPU 0
              -> [Compute Unit Core 1] -> Linux CPU 1
              -> [Compute Unit 1] -> [Compute Unit Core 0] -> Linux CPU 2
              -> [Compute Unit Core 1] -> Linux CPU 3
```

```
[node 1] -> [Compute Unit 0] -> [Compute Unit Core 0] -> Linux CPU 4
          -> [Compute Unit Core 1] -> Linux CPU 5
          -> [Compute Unit 1] -> [Compute Unit Core 0] -> Linux CPU 6
          -> [Compute Unit Core 1] -> Linux CPU 7
```



## **KERNEL LEVEL EXCEPTION HANDLING**

Commentary by Joerg Pommnitz <[joerg@raleigh.ibm.com](mailto:joerg@raleigh.ibm.com)>

When a process runs in kernel mode, it often has to access user mode memory whose address has been passed by an untrusted program. To protect itself the kernel has to verify this address.

In older versions of Linux this was done with the `int verify_area(int type, const void * addr, unsigned long size)` function (which has since been replaced by `access_ok()`).

This function verified that the memory area starting at address 'addr' and of size 'size' was accessible for the operation specified in type (read or write). To do this, `verify_read` had to look up the virtual memory area (vma) that contained the address `addr`. In the normal case (correctly working program), this test was successful. It only failed for a few buggy programs. In some kernel profiling tests, this normally unneeded verification used up a considerable amount of time.

To overcome this situation, Linus decided to let the virtual memory hardware present in every Linux-capable CPU handle this test.

How does this work?

Whenever the kernel tries to access an address that is currently not accessible, the CPU generates a page fault exception and calls the page fault handler:

```
void exc_page_fault(struct pt_regs *regs, unsigned long error_code)
```

in `arch/x86/mm/fault.c`. The parameters on the stack are set up by the low level assembly glue in `arch/x86/entry/entry_32.S`. The parameter `regs` is a pointer to the saved registers on the stack, `error_code` contains a reason code for the exception.

`exc_page_fault()` first obtains the inaccessible address from the CPU control register CR2. If the address is within the virtual address space of the process, the fault probably occurred, because the page was not swapped in, write protected or something similar. However, we are interested in the other case: the address is not valid, there is no vma that contains this address. In this case, the kernel jumps to the `bad_area` label.

There it uses the address of the instruction that caused the exception (i.e. `regs->eip`) to find an address where the execution can continue (fixup). If this search is successful, the fault handler modifies the return address (again `regs->eip`) and returns. The execution will continue at the address in `fixup`.

Where does `fixup` point to?

Since we jump to the contents of `fixup`, `fixup` obviously points to executable code. This code is hidden inside the user access macros. I have picked the `get_user()` macro defined in `arch/x86/include/asm/uaccess.h` as an example. The definition is somewhat hard to follow, so

let's peek at the code generated by the preprocessor and the compiler. I selected the `get_user()` call in `drivers/char/sysrq.c` for a detailed examination.

The original code in `sysrq.c` line 587:

```
get_user(c, buf);
```

The preprocessor output (edited to become somewhat readable):

```
(
{
long __gu_err = - 14 , __gu_val = 0;
const __typeof__(*( ( buf ) )) *__gu_addr = ((buf));
if (((((0 + current_set[0])->tss.segment) == 0x18 ) ||
((sizeof(*(buf))) <= 0xC0000000UL) &&
((unsigned long)(__gu_addr ) <= 0xC0000000UL - (sizeof(*(buf))))))
do {
__gu_err = 0;
switch ((sizeof(*(buf)))) {
case 1:
__asm__ __volatile__(
"1:      mov" "b" " " %2,% " "b" "1\n"
"2:\n"
".section .fixup,\"ax\"\n"
"3:      movl %3,%0\n"
"      xor" "b" " " % "b" "1,% " "b" "1\n"
"      jmp 2b\n"
".section __ex_table,\"a\"\n"
"      .align 4\n"
"      .long 1b,3b\n"
".text"
: "=r"(__gu_err), "=q" (__gu_val): "m"(*(struct _
→_large_struct *)
( __gu_addr )) , "i"(- 14 ), "0"(__gu_err
→));
break;
case 2:
__asm__ __volatile__(
"1:      mov" "w" " " %2,% " "w" "1\n"
"2:\n"
".section .fixup,\"ax\"\n"
"3:      movl %3,%0\n"
"      xor" "w" " " % "w" "1,% " "w" "1\n"
"      jmp 2b\n"
".section __ex_table,\"a\"\n"
"      .align 4\n"
"      .long 1b,3b\n"
".text"
: "=r"(__gu_err), "=r" (__gu_val) : "m"(*(struct
→_large_struct *)
( __gu_addr )) , "i"(- 14 ), "0"(__gu_err
→));
break;
case 4:
```

```

    __asm__ __volatile__(
        "1:      mov" "l" " " %2,%" "" "1\n"
        "2:\n"
        ".section .fixup,\"ax\"\n"
        "3:      movl %3,%0\n"
        "        xor" "l" " " %" "" "1,%" "" "1\n"
        "        jmp 2b\n"
        ".section __ex_table,\"a\"\n"
        "        .align 4\n"
        "        .long 1b,3b\n"
        ".text"
        : "=r"(__gu_err), "=r" (__gu_val) : "m"(*(struct_
→__large_struct *))
        ( __gu_addr )) ), "i"(- 14 ), "0"(__gu_err));
        break;
    default:
        (__gu_val) = __get_user_bad();
    }
} while (0) ;
((c)) = (__typeof__(*((buf))))__gu_val;
__gu_err;
}
);

```

WOW! Black GCC/assembly magic. This is impossible to follow, so let's see what code gcc generates:

```

>      xorl %edx,%edx
>      movl current_set,%eax
>      cmpl $24,788(%eax)
>      je .L1424
>      cmpl $-1073741825,64(%esp)
>      ja .L1423
> .L1424:
>      movl %edx,%eax
>      movl 64(%esp),%ebx
> #APP
> 1:      movb (%ebx),%dl          /* this is the actual user access */
> 2:
> .section .fixup,"ax"
> 3:      movl $-14,%eax
>      xorb %dl,%dl
>      jmp 2b
> .section __ex_table,"a"
>      .align 4
>      .long 1b,3b
> .text
> #NO_APP
> .L1423:
>      movzbl %dl,%esi

```

The optimizer does a good job and gives us something we can actually understand. Can we? The actual user access is quite obvious. Thanks to the unified address space we can just access

the address in user memory. But what does the .section stuff do?????

To understand this we have to look at the final kernel:

```
> objdump --section-headers vmlinux
>
> vmlinux:      file format elf32-i386
>
> Sections:
> Idx Name          Size      VMA      LMA      File off  Algn
>  0 .text          00098f40  c0100000  c0100000  00001000  2**4
>                CONTENTS, ALLOC, LOAD, READONLY, CODE
>  1 .fixup          000016bc  c0198f40  c0198f40  00099f40  2**0
>                CONTENTS, ALLOC, LOAD, READONLY, CODE
>  2 .rodata         0000f127  c019a5fc  c019a5fc  0009b5fc  2**2
>                CONTENTS, ALLOC, LOAD, READONLY, DATA
>  3 __ex_table      000015c0  c01a9724  c01a9724  000aa724  2**2
>                CONTENTS, ALLOC, LOAD, READONLY, DATA
>  4 .data           0000ea58  c01abcf0  c01abcf0  000abcf0  2**4
>                CONTENTS, ALLOC, LOAD, DATA
>  5 .bss            00018e21  c01ba748  c01ba748  000ba748  2**2
>                ALLOC
>  6 .comment        00000ec4  00000000  00000000  000ba748  2**0
>                CONTENTS, READONLY
>  7 .note           00001068  00000ec4  00000ec4  000bb60c  2**0
>                CONTENTS, READONLY
```

There are obviously 2 non standard ELF sections in the generated object file. But first we want to find out what happened to our code in the final kernel executable:

```
> objdump --disassemble --section=.text vmlinux
>
> c017e785 <do_con_write+c1> xorl    %edx,%edx
> c017e787 <do_con_write+c3> movl    0xc01c7bec,%eax
> c017e78c <do_con_write+c8> cmpl    $0x18,0x314(%eax)
> c017e793 <do_con_write+cf> je      c017e79f <do_con_write+db>
> c017e795 <do_con_write+d1> cmpl    $0xbfffffff,0x40(%esp,1)
> c017e79d <do_con_write+d9> ja      c017e7a7 <do_con_write+e3>
> c017e79f <do_con_write+db> movl    %edx,%eax
> c017e7a1 <do_con_write+dd> movl    0x40(%esp,1),%ebx
> c017e7a5 <do_con_write+e1> movb    (%ebx),%dl
> c017e7a7 <do_con_write+e3> movzbl %dl,%esi
```

The whole user memory access is reduced to 10 x86 machine instructions. The instructions bracketed in the .section directives are no longer in the normal execution path. They are located in a different section of the executable file:

```
> objdump --disassemble --section=.fixup vmlinux
>
> c0199ff5 <.fixup+10b5> movl    $0xffffffff2,%eax
> c0199ffa <.fixup+10ba> xorb     %dl,%dl
> c0199ffc <.fixup+10bc> jmp     c017e7a7 <do_con_write+e3>
```



And finally:

```
> objdump --full-contents --section=__ex_table vmlinux
>
> c01aa7c4 93c017c0 e09f19c0 97c017c0 99c017c0 .....
> c01aa7d4 f6c217c0 e99f19c0 a5e717c0 f59f19c0 .....
> c01aa7e4 080a18c0 01a019c0 0a0a18c0 04a019c0 .....
```

or in human readable byte order:

```
> c01aa7c4 c017c093 c0199fe0 c017c097 c017c099 .....
> c01aa7d4 c017c2f6 c0199fe9 c017e7a5 c0199ff5 .....
                                ~~~~~
                                this is the interesting part!
> c01aa7e4 c0180a08 c019a001 c0180a0a c019a004 .....
```

What happened? The assembly directives:

```
.section .fixup,"ax"
.section __ex_table,"a"
```

told the assembler to move the following code to the specified sections in the ELF object file. So the instructions:

```
3:      movl $-14,%eax
      xorb %dl,%dl
      jmp 2b
```

ended up in the `.fixup` section of the object file and the addresses:

```
.long 1b,3b
```

ended up in the `__ex_table` section of the object file. 1b and 3b are local labels. The local label 1b (1b stands for next label 1 backward) is the address of the instruction that might fault, i.e. in our case the address of the label 1 is c017e7a5: the original assembly code: `> 1: movb (%ebx),%dl` and linked in `vmlinux` : `> c017e7a5 <do_con write+e1> movb (%ebx),%dl`

The local label 3 (backwards again) is the address of the code to handle the fault, in our case the actual value is c0199ff5: the original assembly code: > 3: movl \$-14,%eax and linked in vmlinux : > c0199ff5 <.fixup+10b5> movl \$0xffffffff2,%eax

If the fixup was able to handle the exception, control flow may be returned to the instruction after the one that triggered the fault, ie. local label 2b.

The assembly code:

```
> .section __ex_table,"a"
>         .align 4
>         .long 1b,3b
```

becomes the value pair:

```
> c01aa7d4 c017c2f6 c0199fe9 c017e7a5 c0199ff5 .....
                                ^this is ^this is
                                1b       3b
```

c017e7a5,c0199ff5 in the exception table of the kernel.

So, what actually happens if a fault from kernel mode with no suitable vma occurs?

1. access to invalid address:

```
> c017e7a5 <do_con_write+e1> movb    (%ebx),%dl
```

2. MMU generates exception
3. CPU calls `exc_page_fault()`
4. `exc_page_fault()` calls `do_user_addr_fault()`
5. `do_user_addr_fault()` calls `kernelmode_fixup_or_oops()`
6. `kernelmode_fixup_or_oops()` calls `fixup_exception()` (`regs->eip == c017e7a5`);
7. `fixup_exception()` calls `search_exception_tables()`
8. `search_exception_tables()` looks up the address c017e7a5 in the exception table (i.e. the contents of the ELF section `__ex_table`) and returns the address of the associated fault handle code c0199ff5.
9. `fixup_exception()` modifies its own return address to point to the fault handle code and returns.
10. execution continues in the fault handling code.
11.
  - a) EAX becomes -EFAULT (`== -14`)
  - b) DL becomes zero (the value we “read” from user space)
  - c) execution continues at local label 2 (address of the instruction immediately after the faulting user access).

The steps 8a to 8c in a certain way emulate the faulting instruction.

That’s it, mostly. If you look at our example, you might ask why we set EAX to -EFAULT in the exception handler code. Well, the `get_user()` macro actually returns a value: 0, if the user access was successful, -EFAULT on failure. Our original code did not test this return value, however the inline assembly code in `get_user()` tries to return -EFAULT. GCC selected EAX to return this value.

NOTE: Due to the way that the exception table is built and needs to be ordered, only use exceptions for code in the `.text` section. Any other section will cause the exception table to not be sorted correctly, and the exceptions will fail.

Things changed when 64-bit support was added to x86 Linux. Rather than double the size of the exception table by expanding the two entries from 32-bits to 64 bits, a clever trick was used to store addresses as relative offsets from the table itself. The assembly code changed from:

```
.long 1b,3b
to:
```

```
.long (from) - .  
.long (to) - .
```

and the C-code that uses these values converts back to absolute addresses like this:

```
ex_insn_addr(const struct exception_table_entry *x)  
{  
    return (unsigned long)&x->insn + x->insn;  
}
```

In v4.6 the exception table entry was expanded with a new field “handler”. This is also 32-bits wide and contains a third relative function pointer which points to one of:

- 1) **int ex\_handler\_default(const struct exception\_table\_entry \*fixup)** This is legacy case that just jumps to the fixup code
- 2) **int ex\_handler\_fault(const struct exception\_table\_entry \*fixup)** This case provides the fault number of the trap that occurred at entry->insn. It is used to distinguish page faults from machine check.

More functions can easily be added.

CONFIG\_BUILDTIME\_TABLE\_SORT allows the \_\_ex\_table section to be sorted post link of the kernel image, via a host utility scripts/sorttable. It will set the symbol main\_extable\_sort\_needed to 0, avoiding sorting the \_\_ex\_table section at boot time. With the exception table sorted, at runtime when an exception occurs we can quickly lookup the \_\_ex\_table entry via binary search.

This is not just a boot time optimization, some architectures require this table to be sorted in order to handle exceptions relatively early in the boot process. For example, i386 makes use of this form of exception handling before paging support is even enabled!



## **KERNEL STACKS**

### **6.1 Kernel stacks on x86-64 bit**

Most of the text from Keith Owens, hacked by AK

x86\_64 page size (PAGE\_SIZE) is 4K.

Like all other architectures, x86\_64 has a kernel stack for every active thread. These thread stacks are THREAD\_SIZE (2\*PAGE\_SIZE) big. These stacks contain useful data as long as a thread is alive or a zombie. While the thread is in user space the kernel stack is empty except for the thread\_info structure at the bottom.

In addition to the per thread stacks, there are specialized stacks associated with each CPU. These stacks are only used while the kernel is in control on that CPU; when a CPU returns to user space the specialized stacks contain no useful data. The main CPU stacks are:

- Interrupt stack. IRQ\_STACK\_SIZE

Used for external hardware interrupts. If this is the first external hardware interrupt (i.e. not a nested hardware interrupt) then the kernel switches from the current task to the interrupt stack. Like the split thread and interrupt stacks on i386, this gives more room for kernel interrupt processing without having to increase the size of every per thread stack.

The interrupt stack is also used when processing a softirq.

Switching to the kernel interrupt stack is done by software based on a per CPU interrupt nest counter. This is needed because x86-64 "IST" hardware stacks cannot nest without races.

x86\_64 also has a feature which is not available on i386, the ability to automatically switch to a new stack for designated events such as double fault or NMI, which makes it easier to handle these unusual events on x86\_64. This feature is called the Interrupt Stack Table (IST). There can be up to 7 IST entries per CPU. The IST code is an index into the Task State Segment (TSS). The IST entries in the TSS point to dedicated stacks; each stack can be a different size.

An IST is selected by a non-zero value in the IST field of an interrupt-gate descriptor. When an interrupt occurs and the hardware loads such a descriptor, the hardware automatically sets the new stack pointer based on the IST value, then invokes the interrupt handler. If the interrupt came from user mode, then the interrupt handler prologue will switch back to the per-thread stack. If software wants to allow nested IST interrupts then the handler must adjust the IST values on entry to and exit from the interrupt handler. (This is occasionally done, e.g. for debug exceptions.)

Events with different IST codes (i.e. with different stacks) can be nested. For example, a debug interrupt can safely be interrupted by an NMI. arch/x86\_64/kernel/entry.S::paranoidentry ad-

justs the stack pointers on entry to and exit from all IST events, in theory allowing IST events with the same code to be nested. However in most cases, the stack size allocated to an IST assumes no nesting for the same code. If that assumption is ever broken then the stacks will become corrupt.

The currently assigned IST stacks are:

- `ESTACK_DF`. `EXCEPTION_STKSZ (PAGE_SIZE)`.

Used for interrupt 8 - Double Fault Exception (`#DF`).

Invoked when handling one exception causes another exception. Happens when the kernel is very confused (e.g. kernel stack pointer corrupt). Using a separate stack allows the kernel to recover from it well enough in many cases to still output an oops.

- `ESTACK_NMI`. `EXCEPTION_STKSZ (PAGE_SIZE)`.

Used for non-maskable interrupts (NMI).

NMI can be delivered at any time, including when the kernel is in the middle of switching stacks. Using IST for NMI events avoids making assumptions about the previous state of the kernel stack.

- `ESTACK_DB`. `EXCEPTION_STKSZ (PAGE_SIZE)`.

Used for hardware debug interrupts (interrupt 1) and for software debug interrupts (`INT3`).

When debugging a kernel, debug interrupts (both hardware and software) can occur at any time. Using IST for these interrupts avoids making assumptions about the previous state of the kernel stack.

To handle nested `#DB` correctly there exist two instances of DB stacks. On `#DB` entry the IST stackpointer for `#DB` is switched to the second instance so a nested `#DB` starts from a clean stack. The nested `#DB` switches the IST stackpointer to a guard hole to catch triple nesting.

- `ESTACK_MCE`. `EXCEPTION_STKSZ (PAGE_SIZE)`.

Used for interrupt 18 - Machine Check Exception (`#MC`).

MCE can be delivered at any time, including when the kernel is in the middle of switching stacks. Using IST for MCE events avoids making assumptions about the previous state of the kernel stack.

For more details see the Intel IA32 or AMD AMD64 architecture manuals.

## 6.2 Printing backtraces on x86

The question about the ‘?’ preceding function names in an x86 stacktrace keeps popping up, here’s an indepth explanation. It helps if the reader stares at `print_context_stack()` and the whole machinery in and around `arch/x86/kernel/dumpstack.c`.

Adapted from Ingo’s mail, Message-ID: <20150521101614.GA10889@gmail.com>:

We always scan the full kernel stack for return addresses stored on the kernel stack(s)<sup>1</sup>, from stack top to stack bottom, and print out anything that ‘looks like’ a kernel text address.

---

<sup>1</sup> For things like IRQ and IST stacks, we also scan those stacks, in the right order, and try to cross from one stack into another reconstructing the call chain. This works most of the time.

If it fits into the frame pointer chain, we print it without a question mark, knowing that it's part of the real backtrace.

If the address does not fit into our expected frame pointer chain we still print it, but we print a '?'. It can mean two things:

- either the address is not part of the call chain: it's just stale values on the kernel stack, from earlier function calls. This is the common case.
- or it is part of the call chain, but the frame pointer was not set up properly within the function, so we don't recognize it.

This way we will always print out the real call chain (plus a few more entries), regardless of whether the frame pointer was set up correctly or not - but in most cases we'll get the call chain right as well. The entries printed are strictly in stack order, so you can deduce more information from that as well.

The most important property of this method is that we never lose information: we always strive to print all addresses on the stack(s) that look like kernel text addresses, so if debug information is wrong, we still print out the real call chain as well - just with more question marks than ideal.





## **KERNEL ENTRIES**

This file documents some of the kernel entries in `arch/x86/entry/entry_64.S`. A lot of this explanation is adapted from an email from Ingo Molnar:

<https://lore.kernel.org/r/20110529191055.GC9835%40elte.hu>

The x86 architecture has quite a few different ways to jump into kernel code. Most of these entry points are registered in `arch/x86/kernel/traps.c` and implemented in `arch/x86/entry/entry_64.S` for 64-bit, `arch/x86/entry/entry_32.S` for 32-bit and finally `arch/x86/entry/entry_64_compat.S` which implements the 32-bit compatibility syscall entry points and thus provides for 32-bit processes the ability to execute syscalls when running on 64-bit kernels.

The IDT vector assignments are listed in `arch/x86/include/asm/irq_vectors.h`.

Some of these entries are:

- `system_call`: syscall instruction from 64-bit code.
- `entry_INT80_compat`: int 0x80 from 32-bit or 64-bit code; compat syscall either way.
- `entry_INT80_compat`, `ia32_sysenter`: syscall and sysenter from 32-bit code
- `interrupt`: An array of entries. Every IDT vector that doesn't explicitly point somewhere else gets set to the corresponding value in `interrupts`. These point to a whole array of magically-generated functions that make their way to `do_IRQ` with the interrupt number as a parameter.
- APIC interrupts: Various special-purpose interrupts for things like TLB shutdown.
- Architecturally-defined exceptions like `divide_error`.

There are a few complexities here. The different x86-64 entries have different calling conventions. The syscall and sysenter instructions have their own peculiar calling conventions. Some of the IDT entries push an error code onto the stack; others don't. IDT entries using the IST alternative stack mechanism need their own magic to get the stack frames right. (You can find some documentation in the AMD APM, Volume 2, Chapter 8 and the Intel SDM, Volume 3, Chapter 6.)

Dealing with the `swapgs` instruction is especially tricky. `Swapgs` toggles whether `gs` is the kernel `gs` or the user `gs`. The `swapgs` instruction is rather fragile: it must nest perfectly and only in single depth, it should only be used if entering from user mode to kernel mode and then when returning to user-space, and precisely so. If we mess that up even slightly, we crash.

So when we have a secondary entry, already in kernel mode, we *must not* use `SWAPGS` blindly - nor must we forget doing a `SWAPGS` when it's not switched/swapped yet.

Now, there's a secondary complication: there's a cheap way to test which mode the CPU is in and an expensive way.

The cheap way is to pick this info off the entry frame on the kernel stack, from the CS of the ptregs area of the kernel stack:

```
xorl %ebx,%ebx
testl $3,CS+8(%rsp)
je error_kernelspace
SWAPGS
```

The expensive (paranoid) way is to read back the MSR\_GS\_BASE value (which is what SWAPGS modifies):

```
movl $1,%ebx
movl $MSR_GS_BASE,%ecx
rdmsr
testl %edx,%edx
js 1f /* negative -> in kernel */
SWAPGS
xorl %ebx,%ebx
1: ret
```

If we are at an interrupt or user-trap/gate-alike boundary then we can use the faster check: the stack will be a reliable indicator of whether SWAPGS was already done: if we see that we are a secondary entry interrupting kernel mode execution, then we know that the GS base has already been switched. If it says that we interrupted user-space execution then we must do the SWAPGS.

But if we are in an NMI/MCE/DEBUG/whatever super-atomic entry context, which might have triggered right after a normal entry wrote CS to the stack but before we executed SWAPGS, then the only safe way to check for GS is the slower method: the RDMSR.

Therefore, super-atomic entries (except NMI, which is handled separately) must use idtentry with paranoid=1 to handle gsbases correctly. This triggers three main behavior changes:

- Interrupt entry will use the slower gsbases check.
- Interrupt entry from user mode will switch off the IST stack.
- Interrupt exit to kernel mode will not attempt to reschedule.

We try to only use IST entries and the paranoid entry code for vectors that absolutely need the more expensive check for the GS base - and we generate all 'normal' entry points with the regular (faster) paranoid=0 variant.

## EARLY PRINTK

Mini-HOWTO for using the earlyprintk=dbgp boot option with a USB2 Debug port key and a debug cable, on x86 systems.

You need two computers, the 'USB debug key' special gadget and two USB cables, connected like this:

[host/target] <-----> [USB debug key] <-----> [client/console]

### 8.1 Hardware requirements

- a) Host/target system needs to have USB debug port capability.

You can check this capability by looking at a 'Debug port' bit in the lspci -vvv output:

```
# lspci -vvv
...
00:1d.7 USB Controller: Intel Corporation 82801H (ICH8 Family) USB2 EHCI
↳ Controller #1 (rev 03) (prog-if 20 [EHCI])
    Subsystem: Lenovo ThinkPad T61
    Control: I/O- Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr-
↳ Stepping- SERR+ FastB2B- DisINTx-
    Status: Cap+ 66MHz- UDF- FastB2B+ ParErr- DEVSEL=medium >TAbort-
↳ <TAbort- <MAbort- >SERR- <PERR- INTx-
    Latency: 0
    Interrupt: pin D routed to IRQ 19
    Region 0: Memory at fe227000 (32-bit, non-prefetchable) [size=1K]
    Capabilities: [50] Power Management version 2
        Flags: PMEClk- DSI- D1- D2- AuxCurrent=375mA PME(D0+,D1-,
↳ D2-,D3hot+,D3cold+)
            Status: D0 PME-Enable- DSel=0 DScale=0 PME+
    Capabilities: [58] Debug port: BAR=1 offset=00a0
        ^^^^^^^^^^ <===== [ HERE ]
    Kernel driver in use: ehci_hcd
    Kernel modules: ehci-hcd
...
```

---

**Note:** If your system does not list a debug port capability then you probably won't be

able to use the USB debug key.

---

- b) You also need a NetChip USB debug cable/key:

<http://www.plxtech.com/products/NET2000/NET20DC/default.asp>

This is a small blue plastic connector with two USB connections; it draws power from its USB connections.

- c) You need a second client/console system with a high speed USB 2.0 port.
- d) The NetChip device must be plugged directly into the physical debug port on the “host/target” system. You cannot use a USB hub in between the physical debug port and the “host/target” system.

The EHCI debug controller is bound to a specific physical USB port and the NetChip device will only work as an early printk device in this port. The EHCI host controllers are electrically wired such that the EHCI debug controller is hooked up to the first physical port and there is no way to change this via software. You can find the physical port through experimentation by trying each physical port on the system and rebooting. Or you can try and use `lsusb` or look at the kernel info messages emitted by the usb stack when you plug a usb device into various ports on the “host/target” system.

Some hardware vendors do not expose the usb debug port with a physical connector and if you find such a device send a complaint to the hardware vendor, because there is no reason not to wire this port into one of the physically accessible ports.

- e) It is also important to note, that many versions of the NetChip device require the “client/console” system to be plugged into the right hand side of the device (with the product logo facing up and readable left to right). The reason being is that the 5 volt power supply is taken from only one side of the device and it must be the side that does not get rebooted.

## 8.2 Software requirements

- a) On the host/target system:

You need to enable the following kernel config option:

`CONFIG_EARLY_PRINTKDBG=y`

And you need to add the boot command line: “earlyprintk=dbgp”.

---

**Note:** If you are using Grub, append it to the ‘kernel’ line in `/etc/grub.conf`. If you are using Grub2 on a BIOS firmware system, append it to the ‘linux’ line in `/boot/grub2/grub.cfg`. If you are using Grub2 on an EFI firmware system, append it to the ‘linux’ or ‘linuxefi’ line in `/boot/grub2/grub.cfg` or `/boot/efi/EFI/<distro>/grub.cfg`.

---

On systems with more than one EHCI debug controller you must specify the correct EHCI debug controller number. The ordering comes from the PCI bus enumeration of the EHCI controllers. The default with no number argument

is “0” or the first EHCI debug controller. To use the second EHCI debug controller, you would use the command line: “earlyprintk=dbgp1”

**Note:** normally earlyprintk console gets turned off once the regular console is alive - use “earlyprintk=dbgp,keep” to keep this channel open beyond early bootup. This can be useful for debugging crashes under Xorg, etc.

b) On the client/console system:

You should enable the following kernel config option:

```
CONFIG_USB_SERIAL_DEBUG=y
```

On the next bootup with the modified kernel you should get a /dev/ttyUSBx device(s).

Now this channel of kernel messages is ready to be used: start your favorite terminal emulator (minicom, etc.) and set it up to use /dev/ttyUSB0 - or use a raw ‘cat /dev/ttyUSBx’ to see the raw output.

c) On Nvidia Southbridge based systems: the kernel will try to probe and find out which port has a debug device connected.

## 8.3 Testing

You can test the output by using earlyprintk=dbgp,keep and provoking kernel messages on the host/target system. You can provoke a harmless kernel message by for example doing:

```
echo h > /proc/sysrq-trigger
```

On the host/target system you should see this help line in “dmesg” output:

```
SysRq : HELP : loglevel(0-9) reBoot Crashdump terminate-all-tasks(E) memory-
↳ full-oom-kill(F) kill-all-tasks(I) saK show-backtrace-all-active-cpus(L)
↳ show-memory-usage(M) nice-all-RT-tasks(N) powerOff show-registers(P) show-
↳ all-timers(Q) unRaw Sync show-task-states(T) Unmount show-blocked-tasks(W)
↳ dump-ftrace-buffer(Z)
```

On the client/console system do:

```
cat /dev/ttyUSB0
```

And you should see the help line above displayed shortly after you’ve provoked it on the host system.

If it does not work then please ask about it on the [linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org) mailing list or contact the x86 maintainers.



## **ORC UNWINDER**

### **9.1 Overview**

The kernel `CONFIG_UNWINDER_ORC` option enables the ORC unwinder, which is similar in concept to a DWARF unwinder. The difference is that the format of the ORC data is much simpler than DWARF, which in turn allows the ORC unwinder to be much simpler and faster.

The ORC data consists of unwind tables which are generated by `objtool`. They contain out-of-band data which is used by the in-kernel ORC unwinder. `Objtool` generates the ORC data by first doing compile-time stack metadata validation (`CONFIG_STACK_VALIDATION`). After analyzing all the code paths of a `.o` file, it determines information about the stack state at each instruction address in the file and outputs that information to the `.orc_unwind` and `.orc_unwind_ip` sections.

The per-object ORC sections are combined at link time and are sorted and post-processed at boot time. The unwinder uses the resulting data to correlate instruction addresses with their stack states at run time.

### **9.2 ORC vs frame pointers**

With frame pointers enabled, GCC adds instrumentation code to every function in the kernel. The kernel's `.text` size increases by about 3.2%, resulting in a broad kernel-wide slowdown. Measurements by Mel Gorman<sup>1</sup> have shown a slowdown of 5-10% for some workloads.

In contrast, the ORC unwinder has no effect on text size or runtime performance, because the debuginfo is out of band. So if you disable frame pointers and enable the ORC unwinder, you get a nice performance improvement across the board, and still have reliable stack traces.

Ingo Molnar says:

“Note that it’s not just a performance improvement, but also an instruction cache locality improvement: 3.2% `.text` savings almost directly transform into a similarly sized reduction in cache footprint. That can transform to even higher speedups for workloads whose cache locality is borderline.”

Another benefit of ORC compared to frame pointers is that it can reliably unwind across interrupts and exceptions. Frame pointer based unwinds can sometimes skip the caller of the interrupted function, if it was a leaf function or if the interrupt hit before the frame pointer was saved.

---

<sup>1</sup> <https://lore.kernel.org/r/20170602104048.jkkzssljsompjdw@su.se>

The main disadvantage of the ORC unwinder compared to frame pointers is that it needs more memory to store the ORC unwind tables: roughly 2-4MB depending on the kernel config.

## 9.3 ORC vs DWARF

ORC debuginfo's advantage over DWARF itself is that it's much simpler. It gets rid of the complex DWARF CFI state machine and also gets rid of the tracking of unnecessary registers. This allows the unwinder to be much simpler, meaning fewer bugs, which is especially important for mission critical oops code.

The simpler debuginfo format also enables the unwinder to be much faster than DWARF, which is important for perf and lockdep. In a basic performance test by Jiri Slaby<sup>2</sup>, the ORC unwinder was about 20x faster than an out-of-tree DWARF unwinder. (Note: That measurement was taken before some performance tweaks were added, which doubled performance, so the speedup over DWARF may be closer to 40x.)

The ORC data format does have a few downsides compared to DWARF. ORC unwind tables take up ~50% more RAM (+1.3MB on an x86 defconfig kernel) than DWARF-based eh\_frame tables.

Another potential downside is that, as GCC evolves, it's conceivable that the ORC data may end up being *too* simple to describe the state of the stack for certain optimizations. But IMO this is unlikely because GCC saves the frame pointer for any unusual stack adjustments it does, so I suspect we'll really only ever need to keep track of the stack pointer and the frame pointer between call frames. But even if we do end up having to track all the registers DWARF tracks, at least we will still be able to control the format, e.g. no complex state machines.

## 9.4 ORC unwind table generation

The ORC data is generated by objtool. With the existing compile-time stack metadata validation feature, objtool already follows all code paths, and so it already has all the information it needs to be able to generate ORC data from scratch. So it's an easy step to go from stack validation to ORC data generation.

It should be possible to instead generate the ORC data with a simple tool which converts DWARF to ORC data. However, such a solution would be incomplete due to the kernel's extensive use of asm, inline asm, and special sections like exception tables.

That could be rectified by manually annotating those special code paths using GNU assembler .cfi annotations in .S files, and homegrown annotations for inline asm in .c files. But asm annotations were tried in the past and were found to be unmaintainable. They were often incorrect/incomplete and made the code harder to read and keep updated. And based on looking at glibc code, annotating inline asm in .c files might be even worse.

Objtool still needs a few annotations, but only in code which does unusual things to the stack like entry code. And even then, far fewer annotations are needed than what DWARF would need, so they're much more maintainable than DWARF CFI annotations.

So the advantages of using objtool to generate ORC data are that it gives more accurate debuginfo, with very few annotations. It also insulates the kernel from toolchain bugs which can

---

<sup>2</sup> <https://lore.kernel.org/r/d2ca5435-6386-29b8-db87-7f227c2b713a@suse.cz>



be very painful to deal with in the kernel since we often have to workaround issues in older versions of the toolchain for years.

The downside is that the unwinder now becomes dependent on objtool's ability to reverse engineer GCC code flow. If GCC optimizations become too complicated for objtool to follow, the ORC data generation might stop working or become incomplete. (It's worth noting that livepatch already has such a dependency on objtool's ability to follow GCC code flow.)

If newer versions of GCC come up with some optimizations which break objtool, we may need to revisit the current implementation. Some possible solutions would be asking GCC to make the optimizations more palatable, or having objtool use DWARF as an additional input, or creating a GCC plugin to assist objtool with its analysis. But for now, objtool follows GCC code quite well.

## 9.5 Unwinder implementation details

Objtool generates the ORC data by integrating with the compile-time stack metadata validation feature, which is described in detail in `tools/objtool/Documentation/stack-validation.txt`. After analyzing all the code paths of a `.o` file, it creates an array of `orc_entry` structs, and a parallel array of instruction addresses associated with those structs, and writes them to the `.orc_unwind` and `.orc_unwind_ip` sections respectively.

The ORC data is split into the two arrays for performance reasons, to make the searchable part of the data (`.orc_unwind_ip`) more compact. The arrays are sorted in parallel at boot time.

Performance is further improved by the use of a fast lookup table which is created at runtime. The fast lookup table associates a given address with a range of indices for the `.orc_unwind` table, so that only a small subset of the table needs to be searched.

## 9.6 Etymology

Orcs, fearsome creatures of medieval folklore, are the Dwarves' natural enemies. Similarly, the ORC unwinder was created in opposition to the complexity and slowness of DWARF.

"Although Orcs rarely consider multiple solutions to a problem, they do excel at getting things done because they are creatures of action, not thought."<sup>3</sup> Similarly, unlike the esoteric DWARF unwinder, the voracious ORC unwinder wastes no time or siloconic effort decoding variable-length zero-extended unsigned-integer byte-coded state-machine-based debug information entries.

Similar to how Orcs frequently unravel the well-intentioned plans of their adversaries, the ORC unwinder frequently unravels stacks with brutal, unyielding efficiency.

ORC stands for Oops Rewind Capability.

---

<sup>3</sup> <http://dustin.wikidot.com/half-orcs-and-orcs>



## ZERO PAGE

The additional fields in struct boot\_params as a part of 32-bit boot protocol of kernel. These should be filled by bootloader or 16-bit real-mode setup code of the kernel. References/settings to it mainly are in:

```
arch/x86/include/uapi/asm/bootparam.h
```

Offset/Size	Proto	Name	Meaning
000/040	ALL	screen_info	Text mode or frame buffer information (struct screen_info)
040/014	ALL	apm_bios_info	APM BIOS information (struct apm_bios_info)
058/008	ALL	tboot_addr	Physical address of tboot shared page
060/010	ALL	ist_info	Intel SpeedStep (IST) BIOS support information (struct ist_info)
070/008	ALL	acpi_rsdp_addr	Physical address of ACPI RSDP table
080/010	ALL	hd0_info	hd0 disk parameter, OBSOLETE!!
090/010	ALL	hd1_info	hd1 disk parameter, OBSOLETE!!
0A0/010	ALL	sys_desc_table	System description table (struct sys_desc_table), OBSOLETE!!
0B0/010	ALL	olpc_ofw_header	OLPC's OpenFirmware CIF and friends
0C0/004	ALL	ext_ramdisk_image	ramdisk_image high 32bits
0C4/004	ALL	ext_ramdisk_size	ramdisk_size high 32bits
0C8/004	ALL	ext_cmd_line_ptr	cmd_line_ptr high 32bits
13C/004	ALL	cc_blob_address	Physical address of Confidential Computing blob
140/080	ALL	edid_info	Video mode setup (struct edid_info)
1C0/020	ALL	efi_info	EFI 32 information (struct efi_info)
1E0/004	ALL	alt_mem_k	Alternative mem check, in KB
1E4/004	ALL	scratch	Scratch field for the kernel setup code
1E8/001	ALL	e820_entries	Number of entries in e820_table (below)
1E9/001	ALL	eddbuf_entries	Number of entries in eddbuf (below)
1EA/001	ALL	edd_mbr_sig_buf_entries	Number of entries in edd_mbr_sig_buffer (below)
1EB/001	ALL	kbd_status	Numlock is enabled
1EC/001	ALL	secure_boot	Secure boot is enabled in the firmware
1EF/001	ALL	sentinel	Used to detect broken bootloaders
290/040	ALL	edd_mbr_sig_buffer	EDD MBR signatures
2D0/A00	ALL	e820_table	E820 memory map table (array of struct e820_entry)
D00/1EC	ALL	eddbuf	EDD data (array of struct edd_info)



## **THE TLB**

When the kernel unmaps or modified the attributes of a range of memory, it has two choices:

1. Flush the entire TLB with a two-instruction sequence. This is a quick operation, but it causes collateral damage: TLB entries from areas other than the one we are trying to flush will be destroyed and must be refilled later, at some cost.
2. Use the `invlpg` instruction to invalidate a single page at a time. This could potentially cost many more instructions, but it is a much more precise operation, causing no collateral damage to other TLB entries.

Which method to do depends on a few things:

1. The size of the flush being performed. A flush of the entire address space is obviously better performed by flushing the entire TLB than doing  $2^{48}/\text{PAGE\_SIZE}$  individual flushes.
2. The contents of the TLB. If the TLB is empty, then there will be no collateral damage caused by doing the global flush, and all of the individual flush will have ended up being wasted work.
3. The size of the TLB. The larger the TLB, the more collateral damage we do with a full flush. So, the larger the TLB, the more attractive an individual flush looks. Data and instructions have separate TLBs, as do different page sizes.
4. The microarchitecture. The TLB has become a multi-level cache on modern CPUs, and the global flushes have become more expensive relative to single-page flushes.

There is obviously no way the kernel can know all these things, especially the contents of the TLB during a given flush. The sizes of the flush will vary greatly depending on the workload as well. There is essentially no “right” point to choose.

You may be doing too many individual invalidations if you see the `invlpg` instruction (or instructions `_near_it`) show up high in profiles. If you believe that individual invalidations being called too often, you can lower the tunable:

```
/sys/kernel/debug/x86/tlb_single_page_flush_ceiling
```

This will cause us to do the global flush for more cases. Lowering it to 0 will disable the use of the individual flushes. Setting it to 1 is a very conservative setting and it should never need to be 0 under normal circumstances.

Despite the fact that a single individual flush on x86 is guaranteed to flush a full 2MB<sup>1</sup>, `hugetlbfs` always uses the full flushes. THP is treated exactly the same as normal memory.

---

<sup>1</sup> A footnote in Intel’s SDM “4.10.4.2 Recommended Invalidation” says: “One execution of `INVLPG` is sufficient even for a page with size greater than 4 KBytes.”

You might see `invlpg` inside of `flush_tlb_mm_range()` show up in profiles, or you can use the `trace_tlb_flush()` tracepoints. to determine how long the flush operations are taking.

Essentially, you are balancing the cycles you spend doing `invlpg` with the cycles that you spend refilling the TLB later.

You can measure how expensive TLB refills are by using performance counters and ‘perf stat’, like this:

```
perf stat -e
cpu/event=0x8,umask=0x84,name=dtlb_load_misses_walk_duration/,
cpu/event=0x8,umask=0x82,name=dtlb_load_misses_walk_completed/,
cpu/event=0x49,umask=0x4,name=dtlb_store_misses_walk_duration/,
cpu/event=0x49,umask=0x2,name=dtlb_store_misses_walk_completed/,
cpu/event=0x85,umask=0x4,name=itlb_misses_walk_duration/,
cpu/event=0x85,umask=0x2,name=itlb_misses_walk_completed/
```

That works on an IvyBridge-era CPU (i5-3320M). Different CPUs may have differently-named counters, but they should at least be there in some form. You can use `pmu-tools` ‘`ocperf list`’ (<https://github.com/andikleen/pmu-tools>) to find the right counters for a given CPU.

## MTRR (MEMORY TYPE RANGE REGISTER) CONTROL

### Authors

- Richard Gooch <[rgooch@atnf.csiro.au](mailto:rgooch@atnf.csiro.au)> - 3 Jun 1999
- Luis R. Rodriguez <[mcgrof@do-not-panic.com](mailto:mcgrof@do-not-panic.com)> - April 9, 2015

### 12.1 Phasing out MTRR use

MTRR use is replaced on modern x86 hardware with PAT. Direct MTRR use by drivers on Linux is now completely phased out, device drivers should use `arch_phys_wc_add()` in combination with `ioremap_wc()` to make MTRR effective on non-PAT systems while a no-op but equally effective on PAT enabled systems.

Even if Linux does not use MTRRs directly, some x86 platform firmware may still set up MTRRs early before booting the OS. They do this as some platform firmware may still have implemented access to MTRRs which would be controlled and handled by the platform firmware directly. An example of platform use of MTRRs is through the use of SMI handlers, one case could be for fan control, the platform code would need uncachable access to some of its fan control registers. Such platform access does not need any Operating System MTRR code in place other than `mtrr_type_lookup()` to ensure any OS specific mapping requests are aligned with platform MTRR setup. If MTRRs are only set up by the platform firmware code though and the OS does not make any specific MTRR mapping requests `mtrr_type_lookup()` should always return `MTRR_TYPE_INVALID`.

For details refer to *PAT (Page Attribute Table)*.

---

**Tip:** On Intel P6 family processors (Pentium Pro, Pentium II and later) the Memory Type Range Registers (MTRRs) may be used to control processor access to memory ranges. This is most useful when you have a video (VGA) card on a PCI or AGP bus. Enabling write-combining allows bus write transfers to be combined into a larger transfer before bursting over the PCI/AGP bus. This can increase performance of image write operations 2.5 times or more.

The Cyrix 6x86, 6x86MX and M II processors have Address Range Registers (ARRs) which provide a similar functionality to MTRRs. For these, the ARRs are used to emulate the MTRRs.

The AMD K6-2 (stepping 8 and above) and K6-3 processors have two MTRRs. These are supported. The AMD Athlon family provide 8 Intel style MTRRs.

The Centaur C6 (WinChip) has 8 MCRs, allowing write-combining. These are supported.

The VIA Cyrix III and VIA C3 CPUs offer 8 Intel style MTRRs.

The CONFIG\_MTRR option creates a /proc/mtrr file which may be used to manipulate your MTRRs. Typically the X server should use this. This should have a reasonably generic interface so that similar control registers on other processors can be easily supported.

---

There are two interfaces to /proc/mtrr: one is an ASCII interface which allows you to read and write. The other is an ioctl() interface. The ASCII interface is meant for administration. The ioctl() interface is meant for C programs (i.e. the X server). The interfaces are described below, with sample commands and C code.

## 12.2 Reading MTRRs from the shell

```
% cat /proc/mtrr
reg00: base=0x00000000 ( 0MB), size= 128MB: write-back, count=1
reg01: base=0x08000000 ( 128MB), size= 64MB: write-back, count=1
```

Creating MTRRs from the C-shell:

```
# echo "base=0xf8000000 size=0x400000 type=write-combining" >| /proc/mtrr
```

or if you use bash:

```
# echo "base=0xf8000000 size=0x400000 type=write-combining" >| /proc/mtrr
```

And the result thereof:

```
% cat /proc/mtrr
reg00: base=0x00000000 ( 0MB), size= 128MB: write-back, count=1
reg01: base=0x08000000 ( 128MB), size= 64MB: write-back, count=1
reg02: base=0xf8000000 (3968MB), size= 4MB: write-combining, count=1
```

This is for video RAM at base address 0xf8000000 and size 4 megabytes. To find out your base address, you need to look at the output of your X server, which tells you where the linear framebuffer address is. A typical line that you may get is:

```
(--) S3: PCI: 968 rev 0, Linear FB @ 0xf8000000
```

Note that you should only use the value from the X server, as it may move the framebuffer base address, so the only value you can trust is that reported by the X server.

To find out the size of your framebuffer (what, you don't actually know?), the following line will tell you:

```
(--) S3: videoram: 4096k
```

That's 4 megabytes, which is 0x400000 bytes (in hexadecimal). A patch is being written for XFree86 which will make this automatic: in other words the X server will manipulate /proc/mtrr using the ioctl() interface, so users won't have to do anything. If you use a commercial X server, lobby your vendor to add support for MTRRs.



## 12.3 Creating overlapping MTRRs

```
%echo "base=0xf0000000 size=0x1000000 type=write-combining" >/proc/mtrr
%echo "base=0xf0000000 size=0x1000 type=uncachable" >/proc/mtrr
```

And the results:

```
% cat /proc/mtrr
reg00: base=0x00000000 ( 0MB), size= 64MB: write-back, count=1
reg01: base=0xf0000000 (4016MB), size= 16MB: write-combining, count=1
reg02: base=0xf0000000 (4016MB), size= 4kB: uncachable, count=1
```

Some cards (especially Voodoo Graphics boards) need this 4 kB area excluded from the beginning of the region because it is used for registers.

NOTE: You can only create type=uncachable region, if the first region that you created is type=write-combining.

## 12.4 Removing MTRRs from the C-shel

```
% echo "disable=2" >| /proc/mtrr
```

or using bash:

```
% echo "disable=2" >| /proc/mtrr
```

## 12.5 Reading MTRRs from a C program using ioctl()'s

```
/* mtrr-show.c

Source file for mtrr-show (example program to show MTRRs using ioctl()'s)

Copyright (C) 1997-1998 Richard Gooch

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

```
Richard Gooch may be reached by email at  rgooch@atnf.csiro.au
The postal address is:
    Richard Gooch, c/o ATNF, P. O. Box 76, Epping, N.S.W., 2121, Australia.
*/

/*
This program will use an ioctl() on /proc/mtrr to show the current MTRR
settings. This is an alternative to reading /proc/mtrr.

Written by      Richard Gooch    17-DEC-1997

Last updated by Richard Gooch    2-MAY-1998

*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <asm/mtrr.h>

#define TRUE 1
#define FALSE 0
#define ERRSTRING strerror (errno)

static char *mtrr_strings[MTRR_NUM_TYPES] =
{
    "uncachable",           /* 0 */
    "write-combining",      /* 1 */
    "?",                    /* 2 */
    "?",                    /* 3 */
    "write-through",        /* 4 */
    "write-protect",        /* 5 */
    "write-back",           /* 6 */
};

int main ()
{
    int fd;
    struct mtrr_gentry gentry;

    if ( ( fd = open ("/proc/mtrr", O_RDONLY, 0) ) == -1 )
    {
        if (errno == ENOENT)
        {
```

```

    fputs ("/proc/mtrr not found: not supported or you don't have a PPro?\n",
    stderr);
    exit (1);
}
fprintf (stderr, "Error opening /proc/mtrr\t%s\n", ERRSTRING);
exit (2);
}
for (gentry.regnum = 0; ioctl (fd, MTRRIOC_GET_ENTRY, &gentry) == 0;
++gentry.regnum)
{
if (gentry.size < 1)
{
    fprintf (stderr, "Register: %u disabled\n", gentry.regnum);
    continue;
}
fprintf (stderr, "Register: %u base: 0x%lx size: 0x%lx type: %s\n",
    gentry.regnum, gentry.base, gentry.size,
    mtrr_strings[gentry.type]);
}
if (errno == EINVAL) exit (0);
fprintf (stderr, "Error doing ioctl(2) on /dev/mtrr\t%s\n", ERRSTRING);
exit (3);
} /* End Function main */

```

## 12.6 Creating MTRRs from a C programme using ioctl()'s

```

/* mtrr-add.c

Source file for mtrr-add (example programme to add an MTRRs using ioctl())

Copyright (C) 1997-1998 Richard Gooch

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Richard Gooch may be reached by email at rgooch@atnf.csiro.au
The postal address is:

```

```
Richard Gooch, c/o ATNF, P. O. Box 76, Epping, N.S.W., 2121, Australia.
*/

/*
This programme will use an ioctl() on /proc/mtrr to add an entry. The first
available mtrr is used. This is an alternative to writing /proc/mtrr.

Written by      Richard Gooch    17-DEC-1997

Last updated by Richard Gooch    2-MAY-1998

*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <asm/mtrr.h>

#define TRUE 1
#define FALSE 0
#define ERRSTRING strerror (errno)

static char *mtrr_strings[MTRR_NUM_TYPES] =
{
    "uncachable",           /* 0 */
    "write-combining",      /* 1 */
    "?",                    /* 2 */
    "?",                    /* 3 */
    "write-through",        /* 4 */
    "write-protect",        /* 5 */
    "write-back",           /* 6 */
};

int main (int argc, char **argv)
{
    int fd;
    struct mtrr_sentry sentry;

    if (argc != 4)
    {
        fprintf (stderr, "Usage:\tmtrr-add base size type\n");
        exit (1);
    }
}
```

```

    sentry.base = strtoul (argv[1], NULL, 0);
    sentry.size = strtoul (argv[2], NULL, 0);
    for (sentry.type = 0; sentry.type < MTRR_NUM_TYPES; ++sentry.type)
    {
    if (strcmp (argv[3], mtrr_strings[sentry.type]) == 0) break;
    }
    if (sentry.type >= MTRR_NUM_TYPES)
    {
    fprintf (stderr, "Illegal type: \"%s\\\"\\n\", argv[3]);
    exit (2);
    }
    if ( ( fd = open ("/proc/mtrr", O_WRONLY, 0) ) == -1 )
    {
    if (errno == ENOENT)
    {
    fputs ("/proc/mtrr not found: not supported or you don't have a PPro?\\n",
    stderr);
    exit (3);
    }
    fprintf (stderr, "Error opening /proc/mtrr\\t%s\\n", ERRSTRING);
    exit (4);
    }
    if (ioctl (fd, MTRRIOC_ADD_ENTRY, &sentry) == -1)
    {
    fprintf (stderr, "Error doing ioctl(2) on /dev/mtrr\\t%s\\n", ERRSTRING);
    exit (5);
    }
    fprintf (stderr, "Sleeping for 5 seconds so you can see the new entry\\n");
    sleep (5);
    close (fd);
    fputs ("I've just closed /proc/mtrr so now the new entry should be gone\\n",
    stderr);
} /* End Function main */

```



## **PAT (PAGE ATTRIBUTE TABLE)**

x86 Page Attribute Table (PAT) allows for setting the memory attribute at the page level granularity. PAT is complementary to the MTRR settings which allows for setting of memory types over physical address ranges. However, PAT is more flexible than MTRR due to its capability to set attributes at page level and also due to the fact that there are no hardware limitations on number of such attribute settings allowed. Added flexibility comes with guidelines for not having memory type aliasing for the same physical memory with multiple virtual addresses.

PAT allows for different types of memory attributes. The most commonly used ones that will be supported at this time are:

WB	Write-back
UC	Uncached
WC	Write-combined
WT	Write-through
UC-	Uncached Minus

### **13.1 PAT APIs**

There are many different APIs in the kernel that allows setting of memory attributes at the page level. In order to avoid aliasing, these interfaces should be used thoughtfully. Below is a table of interfaces available, their intended usage and their memory attribute relationships. Internally, these APIs use a `reserve_memtype()/free_memtype()` interface on the physical address range to avoid any aliasing.

API	RAM	ACPI,...	Reserved/Holes
ioremap	-	UC-	UC-
ioremap_cache	-	WB	WB
ioremap_uc	-	UC	UC
ioremap_wc	-	-	WC
ioremap_wt	-	-	WT
set_memory_uc, set_memory_wb	UC-	-	-
set_memory_wc, set_memory_wb	WC	-	-
set_memory_wt, set_memory_wb	WT	-	-
pci sysfs resource	-	-	UC-
pci sysfs re- source_wc is IORE- SOURCE_PREFETCH	-	-	WC
pci proc !PCI- IOC_WRITE_COMBINE	-	-	UC-
pci proc PCI- IOC_WRITE_COMBINE	-	-	WC
/dev/mem read-write	-	WB/WC/UC-	WB/WC/UC-
/dev/mem mmap SYNC flag	-	UC-	UC-
/dev/mem mmap !SYNC flag and any alias to this area	-	WB/WC/UC-  (from existing alias)	WB/WC/UC- (from existing alias)
/dev/mem mmap !SYNC flag no alias to this area and MTRR says WB	-	WB	WB
/dev/mem mmap !SYNC flag no alias to this area and MTRR says !WB	-	-	UC-

## 13.2 Advanced APIs for drivers

A. Exporting pages to users with `remap_pfn_range`, `io_remap_pfn_range`, `vmf_insert_pfn`.

Drivers wanting to export some pages to userspace do it by using mmap interface and a combination of:

- 1) `pgprot_noncached()`
- 2) `io_remap_pfn_range()` or `remap_pfn_range()` or `vmf_insert_pfn()`

With PAT support, a new API `pgprot_writecombine` is being added. So, drivers can continue to use the above sequence, with either `pgprot_noncached()` or `pgprot_writecombine()` in step 1, followed by step 2.



In addition, step 2 internally tracks the region as UC or WC in memtype list in order to ensure no conflicting mapping.

Note that this set of APIs only works with IO (non RAM) regions. If driver wants to export a RAM region, it has to do set\_memory\_uc() or set\_memory\_wc() as step 0 above and also track the usage of those pages and use set\_memory\_wb() before the page is freed to free pool.

13.3 MTRR effects on PAT / non-PAT systems

The following table provides the effects of using write-combining MTRRs when using ioremap\*() calls on x86 for both non-PAT and PAT systems. Ideally mtrr\_add() usage will be phased out in favor of arch\_phys\_wc\_add() which will be a no-op on PAT enabled systems. The region over which a arch\_phys\_wc\_add() is made, should already have been ioremapped with WC attributes or PAT entries, this can be done by using ioremap\_wc() / set\_memory\_wc(). Devices which combine areas of IO memory desired to remain uncachable with areas where write-combining is desirable should consider use of ioremap\_uc() followed by set\_memory\_wc() to white-list effective write-combined areas. Such use is nevertheless discouraged as the effective memory type is considered implementation defined, yet this strategy can be used as last resort on devices with size-constrained regions where otherwise MTRR write-combining would otherwise not be effective.

MTRR		Non-PAT	PAT	Linux ioremap value	Effective memory type	
					Non-PAT	PAT
		PAT				
		PCD				
		PWT				
WC	000	WB		_PAGE_CACHE_MODE_WB	WC	WC
WC	001	WC		_PAGE_CACHE_MODE_WC	WC*	WC
WC	010	UC-		_PAGE_CACHE_MODE_UC_MINUS	WC*	UC
WC	011	UC		_PAGE_CACHE_MODE_UC	UC	UC
(*) denotes implementation defined and is discouraged						

**Note:** - in the above table mean “Not suggested usage for the API”. Some of the -’s are strictly enforced by the kernel. Some others are not really enforced today, but may be enforced in future.

For ioremap and pci access through /sys or /proc - The actual type returned can be more restrictive, in case of any existing aliasing for that address. For example: If there is an existing uncached mapping, a new ioremap\_wc can return uncached mapping in place of write-combine requested.

set\_memory\_[uc|wc|wt] and set\_memory\_wb should be used in pairs, where driver will first make a region uc, wc or wt and switch it back to wb after use.

Over time writes to /proc/mtrr will be deprecated in favor of using PAT based interfaces. Users writing to /proc/mtrr are suggested to use above interfaces.

Drivers should use `ioremap_[uc|wc]` to access PCI BARs with `[uc|wc]` access types.

Drivers should use `set_memory_[uc|wc|wt]` to set access type for RAM ranges.

### 13.4 PAT debugging

With `CONFIG_DEBUG_FS` enabled, PAT memtype list can be examined by:

```
# mount -t debugfs debugfs /sys/kernel/debug
# cat /sys/kernel/debug/x86/pat_memtype_list
PAT memtype list:
uncached-minus @ 0x7fadf000-0x7fae0000
uncached-minus @ 0x7fb19000-0x7fb1a000
uncached-minus @ 0x7fb1a000-0x7fb1b000
uncached-minus @ 0x7fb1b000-0x7fb1c000
uncached-minus @ 0x7fb1c000-0x7fb1d000
uncached-minus @ 0x7fb1d000-0x7fb1e000
uncached-minus @ 0x7fb1e000-0x7fb25000
uncached-minus @ 0x7fb25000-0x7fb26000
uncached-minus @ 0x7fb26000-0x7fb27000
uncached-minus @ 0x7fb27000-0x7fb28000
uncached-minus @ 0x7fb28000-0x7fb2e000
uncached-minus @ 0x7fb2e000-0x7fb2f000
uncached-minus @ 0x7fb2f000-0x7fb30000
uncached-minus @ 0x7fb31000-0x7fb32000
uncached-minus @ 0x80000000-0x90000000
```

This list shows physical address ranges and various PAT settings used to access those physical address ranges.

Another, more verbose way of getting PAT related debug messages is with “debugpat” boot parameter. With this parameter, various debug messages are printed to dmesg log.

### 13.5 PAT Initialization

The following table describes how PAT is initialized under various configurations. The PAT MSR must be updated by Linux in order to support WC and WT attributes. Otherwise, the PAT MSR has the value programmed in it by the firmware. Note, Xen enables WC attribute in the PAT MSR for guests.

MTRR	PAT	Call Sequence	PAT State	PAT MSR
E	E	MTRR -> PAT init	Enabled	OS
E	D	MTRR -> PAT init	Disabled	•
D	E	MTRR -> PAT disable	Disabled	BIOS
D	D	MTRR -> PAT disable	Disabled	•
•	np/E	PAT -> PAT disable	Disabled	BIOS
•	np/D	PAT -> PAT disable	Disabled	•
E	!P/E	MTRR -> PAT init	Disabled	BIOS
D	!P/E	MTRR -> PAT disable	Disabled	BIOS
!M	!P/E	MTRR stub -> PAT disable	Disabled	BIOS

## Legend

E	Feature enabled in CPU
D	Feature disabled/unsupported in CPU
np	“nopat” boot option specified
!P	CONFIG_X86_PAT option unset
!M	CONFIG_MTRR option unset
Enabled	PAT state set to enabled
Disabled	PAT state set to disabled
OS	PAT initializes PAT MSR with OS setting
BIOS	PAT keeps PAT MSR with BIOS setting



## HARDWARE-FEEDBACK INTERFACE FOR SCHEDULING ON INTEL HARDWARE

### 14.1 Overview

Intel has described the Hardware Feedback Interface (HFI) in the Intel 64 and IA-32 Architectures Software Developer's Manual (Intel SDM) Volume 3 Section 14.6<sup>1</sup>.

The HFI gives the operating system a performance and energy efficiency capability data for each CPU in the system. Linux can use the information from the HFI to influence task placement decisions.

### 14.2 The Hardware Feedback Interface

The Hardware Feedback Interface provides to the operating system information about the performance and energy efficiency of each CPU in the system. Each capability is given as a unit-less quantity in the range [0-255]. Higher values indicate higher capability. Energy efficiency and performance are reported in separate capabilities. Even though on some systems these two metrics may be related, they are specified as independent capabilities in the Intel SDM.

These capabilities may change at runtime as a result of changes in the operating conditions of the system or the action of external factors. The rate at which these capabilities are updated is specific to each processor model. On some models, capabilities are set at boot time and never change. On others, capabilities may change every tens of milliseconds. For instance, a remote mechanism may be used to lower Thermal Design Power. Such change can be reflected in the HFI. Likewise, if the system needs to be throttled due to excessive heat, the HFI may reflect reduced performance on specific CPUs.

The kernel or a userspace policy daemon can use these capabilities to modify task placement decisions. For instance, if either the performance or energy capabilities of a given logical processor becomes zero, it is an indication that the hardware recommends to the operating system to not schedule any tasks on that processor for performance or energy efficiency reasons, respectively.

---

<sup>1</sup> <https://www.intel.com/sdm>

## 14.3 Implementation details for Linux

The infrastructure to handle thermal event interrupts has two parts. In the Local Vector Table of a CPU's local APIC, there exists a register for the Thermal Monitor Register. This register controls how interrupts are delivered to a CPU when the thermal monitor generates an interrupt. Further details can be found in the Intel SDM Vol. 3 Section 10.5<sup>Page 77, 1</sup>.

The thermal monitor may generate interrupts per CPU or per package. The HFI generates package-level interrupts. This monitor is configured and initialized via a set of machine-specific registers. Specifically, the HFI interrupt and status are controlled via designated bits in the `IA32_PACKAGE_THERM_INTERRUPT` and `IA32_PACKAGE_THERM_STATUS` registers, respectively. There exists one HFI table per package. Further details can be found in the Intel SDM Vol. 3 Section 14.9<sup>Page 77, 1</sup>.

The hardware issues an HFI interrupt after updating the HFI table and is ready for the operating system to consume it. CPUs receive such interrupt via the thermal entry in the Local APIC's Local Vector Table.

When servicing such interrupt, the HFI driver parses the updated table and relays the update to userspace using the thermal notification framework. Given that there may be many HFI updates every second, the updates relayed to userspace are throttled at a rate of `CONFIG_HZ` jiffies.

## 14.4 References

## **X86 IOMMU SUPPORT**

The architecture specs can be obtained from the below locations.

- Intel: <http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf>
- AMD: [https://www.amd.com/system/files/TechDocs/48882\\_IOMMU.pdf](https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf)

This guide gives a quick cheat sheet for some basic understanding.

### **15.1 Basic stuff**

ACPI enumerates and lists the different IOMMUs on the platform, and device scope relationships between devices and which IOMMU controls them.

Some ACPI Keywords:

- DMAR - Intel DMA Remapping table
- DRHD - Intel DMA Remapping Hardware Unit Definition
- RMRR - Intel Reserved Memory Region Reporting Structure
- IVRS - AMD I/O Virtualization Reporting Structure
- IVDB - AMD I/O Virtualization Definition Block
- IVHD - AMD I/O Virtualization Hardware Definition

#### **15.1.1 What is Intel RMRR?**

There are some devices the BIOS controls, for e.g USB devices to perform PS2 emulation. The regions of memory used for these devices are marked reserved in the e820 map. When we turn on DMA translation, DMA to those regions will fail. Hence BIOS uses RMRR to specify these regions along with devices that need to access these regions. OS is expected to setup unity mappings for these regions for these devices to access these regions.

### **15.1.2 What is AMD IVRS?**

The architecture defines an ACPI-compatible data structure called an I/O Virtualization Reporting Structure (IVRS) that is used to convey information related to I/O virtualization to system software. The IVRS describes the configuration and capabilities of the IOMMUs contained in the platform as well as information about the devices that each IOMMU virtualizes.

The IVRS provides information about the following:

- IOMMUs present in the platform including their capabilities and proper configuration
- System I/O topology relevant to each IOMMU
- Peripheral devices that cannot be otherwise enumerated
- Memory regions used by SMI/SMM, platform firmware, and platform hardware. These are generally exclusion ranges to be configured by system software.

## **15.2 How is an I/O Virtual Address (IOVA) generated?**

Well behaved drivers call `dma_map_*`() calls before sending command to device that needs to perform DMA. Once DMA is completed and mapping is no longer required, driver performs `dma_unmap_*`() calls to unmap the region.

## **15.3 Intel Specific Notes**

### **15.3.1 Graphics Problems?**

If you encounter issues with graphics devices, you can try adding option `intel_iommu=igfx_off` to turn off the integrated graphics engine. If this fixes anything, please ensure you file a bug reporting the problem.

### **15.3.2 Some exceptions to IOVA**

Interrupt ranges are not address translated, (0xf0000000 - 0xf000ffff). The same is true for peer to peer transactions. Hence we reserve the address from PCI MMIO ranges so they are not allocated for IOVA addresses.

## **15.4 AMD Specific Notes**

### **15.4.1 Graphics Problems?**

If you encounter issues with integrated graphics devices, you can try adding option `iommu=pt` to the kernel command line use a 1:1 mapping for the IOMMU. If this fixes anything, please ensure you file a bug reporting the problem.



## 15.5 Fault reporting

When errors are reported, the IOMMU signals via an interrupt. The fault reason and device that caused it is printed on the console.

## 15.6 Kernel Log Samples

### 15.6.1 Intel Boot Messages

Something like this gets printed indicating presence of DMAR tables in ACPI:

```
ACPI: DMAR (v001 A M I OEMDMAR 0x00000001 MSFT 0x00000097) @
→0x000000007f5b5ef0
```

When DMAR is being processed and initialized by ACPI, prints DMAR locations and any RMRR's processed:

```
ACPI DMAR:Host address width 36
ACPI DMAR:DRHD (flags: 0x00000000)base: 0x00000000fed90000
ACPI DMAR:DRHD (flags: 0x00000000)base: 0x00000000fed91000
ACPI DMAR:DRHD (flags: 0x00000001)base: 0x00000000fed93000
ACPI DMAR:RMRR base: 0x000000000000ed000 end: 0x000000000000effff
ACPI DMAR:RMRR base: 0x000000007f600000 end: 0x000000007fffffff
```

When DMAR is enabled for use, you will notice:

```
PCI-DMA: Using DMAR IOMMU
```

### 15.6.2 Intel Fault reporting

```
DMAR:[DMA Write] Request device [00:02.0] fault addr 6df084000
DMAR:[fault reason 05] PTE Write access is not set
DMAR:[DMA Write] Request device [00:02.0] fault addr 6df084000
DMAR:[fault reason 05] PTE Write access is not set
```

### 15.6.3 AMD Boot Messages

Something like this gets printed indicating presence of the IOMMU:

```
iommu: Default domain type: Translated
iommu: DMA domain TLB invalidation policy: lazy mode
```

### 15.6.4 AMD Fault reporting

```
AMD-Vi: Event logged [IO_PAGE_FAULT domain=0x0007 address=0xffffc02000_  
→ flags=0x0000]  
AMD-Vi: Event logged [IO_PAGE_FAULT device=07:00.0 domain=0x0007_  
→ address=0xffffc02000 flags=0x0000]
```

## INTEL(R) TXT OVERVIEW

Intel's technology for safer computing, Intel(R) Trusted Execution Technology (Intel(R) TXT), defines platform-level enhancements that provide the building blocks for creating trusted platforms.

Intel TXT was formerly known by the code name LaGrande Technology (LT).

Intel TXT in Brief:

- Provides dynamic root of trust for measurement (DRTM)
- Data protection in case of improper shutdown
- Measurement and verification of launched environment

Intel TXT is part of the vPro(TM) brand and is also available some non-vPro systems. It is currently available on desktop systems based on the Q35, X38, Q45, and Q43 Express chipsets (e.g. Dell Optiplex 755, HP dc7800, etc.) and mobile systems based on the GM45, PM45, and GS45 Express chipsets.

For more information, see <http://www.intel.com/technology/security/>. This site also has a link to the Intel TXT MLE Developers Manual, which has been updated for the new released platforms.

Intel TXT has been presented at various events over the past few years, some of which are:

- **LinuxTAG 2008:** <http://www.linuxtag.org/2008/en/conf/events/vp-donnerstag.html>
- **TRUST2008:** [http://www.trust-conference.eu/downloads/Keynote-Speakers/3\\_David-Grawrock\\_The-Front-Door-of-Trusted-Computing.pdf](http://www.trust-conference.eu/downloads/Keynote-Speakers/3_David-Grawrock_The-Front-Door-of-Trusted-Computing.pdf)
- **IDF, Shanghai:** [http://www.prcidf.com.cn/index\\_en.html](http://www.prcidf.com.cn/index_en.html)
- **IDFs 2006, 2007** (I'm not sure if/where they are online)

### 16.1 Trusted Boot Project Overview

Trusted Boot (tboot) is an open source, pre-kernel/VMM module that uses Intel TXT to perform a measured and verified launch of an OS kernel/VMM.

It is hosted on SourceForge at <http://sourceforge.net/projects/tboot>. The mercurial source repo is available at <http://www.bughost.org/repos.hg/tboot.hg>.

Tboot currently supports launching Xen (open source VMM/hypervisor w/ TXT support since v3.2), and now Linux kernels.

## 16.2 Value Proposition for Linux or “Why should you care?”

While there are many products and technologies that attempt to measure or protect the integrity of a running kernel, they all assume the kernel is “good” to begin with. The Integrity Measurement Architecture (IMA) and Linux Integrity Module interface are examples of such solutions.

To get trust in the initial kernel without using Intel TXT, a static root of trust must be used. This bases trust in BIOS starting at system reset and requires measurement of all code executed between system reset through the completion of the kernel boot as well as data objects used by that code. In the case of a Linux kernel, this means all of BIOS, any option ROMs, the bootloader and the boot config. In practice, this is a lot of code/data, much of which is subject to change from boot to boot (e.g. changing NICs may change option ROMs). Without reference hashes, these measurement changes are difficult to assess or confirm as benign. This process also does not provide DMA protection, memory configuration/alias checks and locks, crash protection, or policy support.

By using the hardware-based root of trust that Intel TXT provides, many of these issues can be mitigated. Specifically: many pre-launch components can be removed from the trust chain, DMA protection is provided to all launched components, a large number of platform configuration checks are performed and values locked, protection is provided for any data in the event of an improper shutdown, and there is support for policy-based execution/verification. This provides a more stable measurement and a higher assurance of system configuration and initial state than would be otherwise possible. Since the tboot project is open source, source code for almost all parts of the trust chain is available (excepting SMM and Intel-provided firmware).

## 16.3 How Does it Work?

- Tboot is an executable that is launched by the bootloader as the “kernel” (the binary the bootloader executes).
- It performs all of the work necessary to determine if the platform supports Intel TXT and, if so, executes the GETSEC[SENTER] processor instruction that initiates the dynamic root of trust.
  - If tboot determines that the system does not support Intel TXT or is not configured correctly (e.g. the SINIT AC Module was incorrect), it will directly launch the kernel with no changes to any state.
  - Tboot will output various information about its progress to the terminal, serial port, and/or an in-memory log; the output locations can be configured with a command line switch.
- The GETSEC[SENTER] instruction will return control to tboot and tboot then verifies certain aspects of the environment (e.g. TPM NV lock, e820 table does not have invalid entries, etc.).
- It will wake the APs from the special sleep state the GETSEC[SENTER] instruction had put them in and place them into a wait-for-SIPI state.
  - Because the processors will not respond to an INIT or SIPI when in the TXT environment, it is necessary to create a small VT-x guest for the APs. When they run in this guest, they will simply wait for the INIT-SIPI-SIPI sequence, which will cause

VMEXITs, and then disable VT and jump to the SIPI vector. This approach seemed like a better choice than having to insert special code into the kernel's MP wakeup sequence.

- Tboot then applies an (optional) user-defined launch policy to verify the kernel and initrd.
  - This policy is rooted in TPM NV and is described in the tboot project. The tboot project also contains code for tools to create and provision the policy.
  - Policies are completely under user control and if not present then any kernel will be launched.
  - Policy action is flexible and can include halting on failures or simply logging them and continuing.
- Tboot adjusts the e820 table provided by the bootloader to reserve its own location in memory as well as to reserve certain other TXT-related regions.
- As part of its launch, tboot DMA protects all of RAM (using the VT-d PMRs). Thus, the kernel must be booted with 'intel\_iommu=on' in order to remove this blanket protection and use VT-d's page-level protection.
- Tboot will populate a shared page with some data about itself and pass this to the Linux kernel as it transfers control.
  - The location of the shared page is passed via the boot\_params struct as a physical address.
- The kernel will look for the tboot shared page address and, if it exists, map it.
- As one of the checks/protections provided by TXT, it makes a copy of the VT-d DMARs in a DMA-protected region of memory and verifies them for correctness. The VT-d code will detect if the kernel was launched with tboot and use this copy instead of the one in the ACPI table.
- At this point, tboot and TXT are out of the picture until a shutdown (S<n>)
- In order to put a system into any of the sleep states after a TXT launch, TXT must first be exited. This is to prevent attacks that attempt to crash the system to gain control on reboot and steal data left in memory.
  - The kernel will perform all of its sleep preparation and populate the shared page with the ACPI data needed to put the platform in the desired sleep state.
  - Then the kernel jumps into tboot via the vector specified in the shared page.
  - Tboot will clean up the environment and disable TXT, then use the kernel-provided ACPI information to actually place the platform into the desired sleep state.
  - In the case of S3, tboot will also register itself as the resume vector. This is necessary because it must re-establish the measured environment upon resume. Once the TXT environment has been restored, it will restore the TPM PCRs and then transfer control back to the kernel's S3 resume vector. In order to preserve system integrity across S3, the kernel provides tboot with a set of memory ranges (RAM and RESERVED\_KERN in the e820 table, but not any memory that BIOS might alter over the S3 transition) that tboot will calculate a MAC (message authentication code) over and then seal with the TPM. On resume and once the measured environment has been re-established, tboot will re-calculate the MAC and verify it against the sealed value. Tboot's policy

determines what happens if the verification fails. Note that the c/s 194 of tboot which has the new MAC code supports this.

That's pretty much it for TXT support.

## 16.4 Configuring the System

This code works with 32bit, 32bit PAE, and 64bit (x86\_64) kernels.

In BIOS, the user must enable: TPM, TXT, VT-x, VT-d. Not all BIOSes allow these to be individually enabled/disabled and the screens in which to find them are BIOS-specific.

grub.conf needs to be modified as follows:

```
title Linux 2.6.29-tip w/ tboot
  root (hd0,0)
    kernel /tboot.gz logging=serial,vga,memory
    module /vmlinuz-2.6.29-tip intel_iommu=on ro
      root=LABEL=/ rhgb console=ttyS0,115200 3
    module /initrd-2.6.29-tip.img
    module /Q35_SINIT_17.BIN
```

The kernel option for enabling Intel TXT support is found under the Security top-level menu and is called “Enable Intel(R) Trusted Execution Technology (TXT)”. It is considered EXPERIMENTAL and depends on the generic x86 support (to allow maximum flexibility in kernel build options), since the tboot code will detect whether the platform actually supports Intel TXT and thus whether any of the kernel code is executed.

The Q35\_SINIT\_17.BIN file is what Intel TXT refers to as an Authenticated Code Module. It is specific to the chipset in the system and can also be found on the Trusted Boot site. It is an (unencrypted) module signed by Intel that is used as part of the DRTM process to verify and configure the system. It is signed because it operates at a higher privilege level in the system than any other macrocode and its correct operation is critical to the establishment of the DRTM. The process for determining the correct SINIT ACM for a system is documented in the SINIT-guide.txt file that is on the tboot SourceForge site under the SINIT ACM downloads.

## **AMD MEMORY ENCRYPTION**

Secure Memory Encryption (SME) and Secure Encrypted Virtualization (SEV) are features found on AMD processors.

SME provides the ability to mark individual pages of memory as encrypted using the standard x86 page tables. A page that is marked encrypted will be automatically decrypted when read from DRAM and encrypted when written to DRAM. SME can therefore be used to protect the contents of DRAM from physical attacks on the system.

SEV enables running encrypted virtual machines (VMs) in which the code and data of the guest VM are secured so that a decrypted version is available only within the VM itself. SEV guest VMs have the concept of private and shared memory. Private memory is encrypted with the guest-specific key, while shared memory may be encrypted with hypervisor key. When SME is enabled, the hypervisor key is the same key which is used in SME.

A page is encrypted when a page table entry has the encryption bit set (see below on how to determine its position). The encryption bit can also be specified in the cr3 register, allowing the PGD table to be encrypted. Each successive level of page tables can also be encrypted by setting the encryption bit in the page table entry that points to the next table. This allows the full page table hierarchy to be encrypted. Note, this means that just because the encryption bit is set in cr3, doesn't imply the full hierarchy is encrypted. Each page table entry in the hierarchy needs to have the encryption bit set to achieve that. So, theoretically, you could have the encryption bit set in cr3 so that the PGD is encrypted, but not set the encryption bit in the PGD entry for a PUD which results in the PUD pointed to by that entry to not be encrypted.

When SEV is enabled, instruction pages and guest page tables are always treated as private. All the DMA operations inside the guest must be performed on shared memory. Since the memory encryption bit is controlled by the guest OS when it is operating in 64-bit or 32-bit PAE mode, in all other modes the SEV hardware forces the memory encryption bit to 1.

Support for SME and SEV can be determined through the CPUID instruction. The CPUID function 0x8000001f reports information related to SME:

```
0x8000001f[eax]:  
    Bit[0] indicates support for SME  
    Bit[1] indicates support for SEV  
0x8000001f[ebx]:  
    Bits[5:0]    pagetable bit number used to activate memory  
                  encryption  
    Bits[11:6]   reduction in physical address space, in bits, when  
                  memory encryption is enabled (this only affects  
                  system physical addresses, not guest physical  
                  addresses)
```

If support for SME is present, MSR 0xc00100010 (MSR\_AMD64\_SYSCFG) can be used to determine if SME is enabled and/or to enable memory encryption:

0xc00100010:	
Bit[23]	0 = memory encryption features are disabled 1 = memory encryption features are enabled

If SEV is supported, MSR 0xc0010131 (MSR\_AMD64\_SEV) can be used to determine if SEV is active:

0xc0010131:	
Bit[0]	0 = memory encryption is not active 1 = memory encryption is active

Linux relies on BIOS to set this bit if BIOS has determined that the reduction in the physical address space as a result of enabling memory encryption (see CPUID information above) will not conflict with the address space resource requirements for the system. If this bit is not set upon Linux startup then Linux itself will not set it and memory encryption will not be possible.

The state of SME in the Linux kernel can be documented as follows:

- Supported: The CPU supports SME (determined through CPUID instruction).
- Enabled: Supported and bit 23 of MSR\_AMD64\_SYSCFG is set.
- Active: Supported, Enabled and the Linux kernel is actively applying the encryption bit to page table entries (the SME mask in the kernel is non-zero).

SME can also be enabled and activated in the BIOS. If SME is enabled and activated in the BIOS, then all memory accesses will be encrypted and it will not be necessary to activate the Linux memory encryption support. If the BIOS merely enables SME (sets bit 23 of the MSR\_AMD64\_SYSCFG), then Linux can activate memory encryption by default (CONFIG\_AMD\_MEM\_ENCRYPT\_ACTIVE\_BY\_DEFAULT=y) or by supplying mem\_encrypt=on on the kernel command line. However, if BIOS does not enable SME, then Linux will not be able to activate memory encryption, even if configured to do so by default or the mem\_encrypt=on command line parameter is specified.



## **AMD HSMP INTERFACE**

Newer Fam19h EPYC server line of processors from AMD support system management functionality via HSMP (Host System Management Port).

The Host System Management Port (HSMP) is an interface to provide OS-level software with access to system management functions via a set of mailbox registers.

More details on the interface can be found in chapter “7 Host System Management Port (HSMP)” of the family/model PPR Eg: [https://www.amd.com/system/files/TechDocs/55898\\_B1\\_pub\\_0.50.zip](https://www.amd.com/system/files/TechDocs/55898_B1_pub_0.50.zip)

HSMP interface is supported on EPYC server CPU models only.

### **18.1 HSMP device**

amd\_hsmp driver under the drivers/platforms/x86/ creates miscdevice /dev/hsmp to let user space programs run hsmp mailbox commands.

```
$ ls -al /dev/hsmp crw-r-r- 1 root root 10, 123 Jan 21 21:41 /dev/hsmp
```

#### **Characteristics of the dev node:**

- Write mode is used for running set/configure commands
- Read mode is used for running get/status monitor commands

#### **Access restrictions:**

- Only root user is allowed to open the file in write mode.
- The file can be opened in read mode by all the users.

#### **In-kernel integration:**

- Other subsystems in the kernel can use the exported transport function `hsmp_send_message()`.
- Locking across callers is taken care by the driver.

## 18.2 An example

To access hsmp device from a C program. First, you need to include the headers:

```
#include <linux/amd_hsmp.h>
```

Which defines the supported messages/message IDs.

Next thing, open the device file, as follows:

```
int file;

file = open("/dev/hsmp", O_RDWR);
if (file < 0) {
    /* ERROR HANDLING; you can check errno to see what went wrong */
    exit(1);
}
```

The following IOCTL is defined:

**ioctl(file, HSMP\_IOCTL\_CMD, struct hsmp\_message \*msg)** The argument is a pointer to a:

```
struct hsmp_message {
    __u32    msg_id;                /* Message ID */
    __u16    num_args;              /* Number of input argument_
↳ words in message */
    __u16    response_sz;          /* Number of expected output/
↳ response words */
    __u32    args[HSMP_MAX_MSG_LEN]; /* argument/response buffer */
    __u16    sock_ind;            /* socket number */
};
```

The ioctl would return a non-zero on failure; you can read errno to see what happened. The transaction returns 0 on success.

More details on the interface and message definitions can be found in chapter “7 Host System Management Port (HSMP)” of the respective family/model PPR eg: [https://www.amd.com/system/files/TechDocs/55898\\_B1\\_pub\\_0.50.zip](https://www.amd.com/system/files/TechDocs/55898_B1_pub_0.50.zip)

User space C-APIs are made available by linking against the esmi library, which is provided by the E-SMS project <https://developer.amd.com/e-sms/>. See: [https://github.com/amd/esmi\\_ib\\_library](https://github.com/amd/esmi_ib_library)

## INTEL TRUST DOMAIN EXTENSIONS (TDX)

Intel's Trust Domain Extensions (TDX) protect confidential guest VMs from the host and physical attacks by isolating the guest register state and by encrypting the guest memory. In TDX, a special module running in a special mode sits between the host and the guest and manages the guest/host separation.

Since the host cannot directly access guest registers or memory, much normal functionality of a hypervisor must be moved into the guest. This is implemented using a Virtualization Exception (#VE) that is handled by the guest kernel. A #VE is handled entirely inside the guest kernel, but some require the hypervisor to be consulted.

TDX includes new hypercall-like mechanisms for communicating from the guest to the hypervisor or the TDX module.

### 19.1 New TDX Exceptions

TDX guests behave differently from bare-metal and traditional VMX guests. In TDX guests, otherwise normal instructions or memory accesses can cause #VE or #GP exceptions.

Instructions marked with an "\*" conditionally cause exceptions. The details for these instructions are discussed below.

#### 19.1.1 Instruction-based #VE

- Port I/O (INS, OUTS, IN, OUT)
- HLT
- MONITOR, MWAIT
- WBINVD, INVD
- VMCALL
- RDMSR\*, WRMSR\*
- CPUID\*

### 19.1.2 Instruction-based #GP

- All VMX instructions: INVEPT, INVVPID, VMCLEAR, VMFUNC, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, VMXON
- ENCLS, ENCLU
- GETSEC
- RSM
- ENQCMD
- RDMSR\*,WRMSR\*

### 19.1.3 RDMSR/WRMSR Behavior

MSR access behavior falls into three categories:

- #GP generated
- #VE generated
- “Just works”

In general, the #GP MSRs should not be used in guests. Their use likely indicates a bug in the guest. The guest may try to handle the #GP with a hypercall but it is unlikely to succeed.

The #VE MSRs are typically able to be handled by the hypervisor. Guests can make a hypercall to the hypervisor to handle the #VE.

The “just works” MSRs do not need any special guest handling. They might be implemented by directly passing through the MSR to the hardware or by trapping and handling in the TDX module. Other than possibly being slow, these MSRs appear to function just as they would on bare metal.

### 19.1.4 CPUID Behavior

For some CPUID leaves and sub-leaves, the virtualized bit fields of CPUID return values (in guest EAX/EBX/ECX/EDX) are configurable by the hypervisor. For such cases, the Intel TDX module architecture defines two virtualization types:

- Bit fields for which the hypervisor controls the value seen by the guest TD.
- Bit fields for which the hypervisor configures the value such that the guest TD either sees their native value or a value of 0. For these bit fields, the hypervisor can mask off the native values, but it can not turn *on* values.

A #VE is generated for CPUID leaves and sub-leaves that the TDX module does not know how to handle. The guest kernel may ask the hypervisor for the value with a hypercall.

## 19.2 #VE on Memory Accesses

There are essentially two classes of TDX memory: private and shared. Private memory receives full TDX protections. Its content is protected against access from the hypervisor. Shared memory is expected to be shared between guest and hypervisor and does not receive full TDX protections.

A TD guest is in control of whether its memory accesses are treated as private or shared. It selects the behavior with a bit in its page table entries. This helps ensure that a guest does not place sensitive information in shared memory, exposing it to the untrusted hypervisor.

### 19.2.1 #VE on Shared Memory

Access to shared mappings can cause a #VE. The hypervisor ultimately controls whether a shared memory access causes a #VE, so the guest must be careful to only reference shared pages it can safely handle a #VE. For instance, the guest should be careful not to access shared memory in the #VE handler before it reads the #VE info structure (TDG.VP.VEINFO.GET).

Shared mapping content is entirely controlled by the hypervisor. The guest should only use shared mappings for communicating with the hypervisor. Shared mappings must never be used for sensitive memory content like kernel stacks. A good rule of thumb is that hypervisor-shared memory should be treated the same as memory mapped to userspace. Both the hypervisor and userspace are completely untrusted.

MMIO for virtual devices is implemented as shared memory. The guest must be careful not to access device MMIO regions unless it is also prepared to handle a #VE.

### 19.2.2 #VE on Private Pages

An access to private mappings can also cause a #VE. Since all kernel memory is also private memory, the kernel might theoretically need to handle a #VE on arbitrary kernel memory accesses. This is not feasible, so TDX guests ensure that all guest memory has been “accepted” before memory is used by the kernel.

A modest amount of memory (typically 512M) is pre-accepted by the firmware before the kernel runs to ensure that the kernel can start up without being subjected to a #VE.

The hypervisor is permitted to unilaterally move accepted pages to a “blocked” state. However, if it does this, page access will not generate a #VE. It will, instead, cause a “TD Exit” where the hypervisor is required to handle the exception.

## 19.3 Linux #VE handler

Just like page faults or #GP's, #VE exceptions can be either handled or be fatal. Typically, an unhandled userspace #VE results in a SIGSEGV. An unhandled kernel #VE results in an oops.

Handling nested exceptions on x86 is typically nasty business. A #VE could be interrupted by an NMI which triggers another #VE and hilarity ensues. The TDX #VE architecture anticipated this scenario and includes a feature to make it slightly less nasty.

During `#VE` handling, the TDX module ensures that all interrupts (including NMIs) are blocked. The block remains in place until the guest makes a `TDG.VP.VEINFO.GET TDCALL`. This allows the guest to control when interrupts or a new `#VE` can be delivered.

However, the guest kernel must still be careful to avoid potential `#VE`-triggering actions (discussed above) while this block is in place. While the block is in place, any `#VE` is elevated to a double fault (`#DF`) which is not recoverable.

## 19.4 MMIO handling

In non-TDX VMs, MMIO is usually implemented by giving a guest access to a mapping which will cause a `VMEXIT` on access, and then the hypervisor emulates the access. That is not possible in TDX guests because `VMEXIT` will expose the register state to the host. TDX guests don't trust the host and can't have their state exposed to the host.

In TDX, MMIO regions typically trigger a `#VE` exception in the guest. The guest `#VE` handler then emulates the MMIO instruction inside the guest and converts it into a controlled `TDCALL` to the host, rather than exposing guest state to the host.

MMIO addresses on x86 are just special physical addresses. They can theoretically be accessed with any instruction that accesses memory. However, the kernel instruction decoding method is limited. It is only designed to decode instructions like those generated by `io.h` macros.

MMIO access via other means (like structure overlays) may result in an oops.

## 19.5 Shared Memory Conversions

All TDX guest memory starts out as private at boot. This memory can not be accessed by the hypervisor. However, some kernel users like device drivers might have a need to share data with the hypervisor. To do this, memory must be converted between shared and private. This can be accomplished using some existing memory encryption helpers:

- `set_memory_decrypted()` converts a range of pages to shared.
- `set_memory_encrypted()` converts memory back to private.

Device drivers are the primary user of shared memory, but there's no need to touch every driver. DMA buffers and `ioremap()` do the conversions automatically.

TDX uses `SWIOTLB` for most DMA allocations. The `SWIOTLB` buffer is converted to shared on boot.

For coherent DMA allocation, the DMA buffer gets converted on the allocation. Check `force_dma_unencrypted()` for details.

## 19.6 References

TDX reference material is collected here:

<https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html>





## **PAGE TABLE ISOLATION (PTI)**

### **20.1 Overview**

Page Table Isolation (pti, previously known as KAISER<sup>1</sup>) is a countermeasure against attacks on the shared user/kernel address space such as the “Meltdown” approach<sup>2</sup>.

To mitigate this class of attacks, we create an independent set of page tables for use only when running userspace applications. When the kernel is entered via syscalls, interrupts or exceptions, the page tables are switched to the full “kernel” copy. When the system switches back to user mode, the user copy is used again.

The userspace page tables contain only a minimal amount of kernel data: only what is needed to enter/exit the kernel such as the entry/exit functions themselves and the interrupt descriptor table (IDT). There are a few strictly unnecessary things that get mapped such as the first C function when entering an interrupt (see comments in pti.c).

This approach helps to ensure that side-channel attacks leveraging the paging structures do not function when PTI is enabled. It can be enabled by setting `CONFIG_PAGE_TABLE_ISOLATION=y` at compile time. Once enabled at compile-time, it can be disabled at boot with the ‘nopti’ or ‘pti=’ kernel parameters (see kernel-parameters.txt).

### **20.2 Page Table Management**

When PTI is enabled, the kernel manages two sets of page tables. The first set is very similar to the single set which is present in kernels without PTI. This includes a complete mapping of userspace that the kernel can use for things like `copy_to_user()`.

Although `_complete_`, the user portion of the kernel page tables is crippled by setting the NX bit in the top level. This ensures that any missed kernel->user CR3 switch will immediately crash userspace upon executing its first instruction.

The userspace page tables map only the kernel data needed to enter and exit the kernel. This data is entirely contained in the ‘struct `cpu_entry_area`’ structure which is placed in the fixmap which gives each CPU’s copy of the area a compile-time-fixed virtual address.

For new userspace mappings, the kernel makes the entries in its page tables like normal. The only difference is when the kernel makes entries in the top (PGD) level. In addition to setting

---

<sup>1</sup> <https://gruss.cc/files/kaiser.pdf>

<sup>2</sup> <https://meltdownattack.com/meltdown.pdf>

the entry in the main kernel PGD, a copy of the entry is made in the userspace page tables' PGD.

This sharing at the PGD level also inherently shares all the lower layers of the page tables. This leaves a single, shared set of userspace page tables to manage. One PTE to lock, one set of accessed bits, dirty bits, etc...

## 20.3 Overhead

Protection against side-channel attacks is important. But, this protection comes at a cost:

1. Increased Memory Use
  - a. Each process now needs an order-1 PGD instead of order-0. (Consumes an additional 4k per process).
  - b. The 'cpu\_entry\_area' structure must be 2MB in size and 2MB aligned so that it can be mapped by setting a single PMD entry. This consumes nearly 2MB of RAM once the kernel is decompressed, but no space in the kernel image itself.
2. Runtime Cost
  - a. CR3 manipulation to switch between the page table copies must be done at interrupt, syscall, and exception entry and exit (it can be skipped when the kernel is interrupted, though.) Moves to CR3 are on the order of a hundred cycles, and are required at every entry and exit.
  - b. A "trampoline" must be used for SYSCALL entry. This trampoline depends on a smaller set of resources than the non-PTI SYSCALL entry code, so requires mapping fewer things into the userspace page tables. The downside is that stacks must be switched at entry time.
  - c. Global pages are disabled for all kernel structures not mapped into both kernel and userspace page tables. This feature of the MMU allows different processes to share TLB entries mapping the kernel. Losing the feature means more TLB misses after a context switch. The actual loss of performance is very small, however, never exceeding 1%.
  - d. Process Context IDentifiers (PCID) is a CPU feature that allows us to skip flushing the entire TLB when switching page tables by setting a special bit in CR3 when the page tables are changed. This makes switching the page tables (at context switch, or kernel entry/exit) cheaper. But, on systems with PCID support, the context switch code must flush both the user and kernel entries out of the TLB. The user PCID TLB flush is deferred until the exit to userspace, minimizing the cost. See [intel.com/sdm](http://intel.com/sdm) for the gory PCID/INVPCID details.
  - e. The userspace page tables must be populated for each new process. Even without PTI, the shared kernel mappings are created by copying top-level (PGD) entries into each new process. But, with PTI, there are now *two* kernel mappings: one in the kernel page tables that maps everything and one for the entry/exit structures. At fork(), we need to copy both.
  - f. In addition to the fork()-time copying, there must also be an update to the userspace PGD any time a set\_pgd() is done on a PGD used to map userspace. This ensures that the kernel and userspace copies always map the same userspace memory.
  - g. On systems without PCID support, each CR3 write flushes the entire TLB. That means that each syscall, interrupt or exception flushes the TLB.

- h. INVPCID is a TLB-flushing instruction which allows flushing of TLB entries for non-current PCIDs. Some systems support PCIDs, but do not support INVPCID. On these systems, addresses can only be flushed from the TLB for the current PCID. When flushing a kernel address, we need to flush all PCIDs, so a single kernel address flush will require a TLB-flushing CR3 write upon the next use of every PCID.

## 20.4 Possible Future Work

1. We can be more careful about not actually writing to CR3 unless its value is actually changed.
2. Allow PTI to be enabled/disabled at runtime in addition to the boot-time switching.

## 20.5 Testing

To test stability of PTI, the following test procedure is recommended, ideally doing all of these in parallel:

1. Set `CONFIG_DEBUG_ENTRY=y`
2. Run several copies of all of the tools/testing/selftests/x86/ tests (excluding MPX and protection\_keys) in a loop on multiple CPUs for several minutes. These tests frequently uncover corner cases in the kernel entry code. In general, old kernels might cause these tests themselves to crash, but they should never crash the kernel.
3. Run the ‘perf’ tool in a mode (top or record) that generates many frequent performance monitoring non-maskable interrupts (see “NMI” in /proc/interrupts). This exercises the NMI entry/exit code which is known to trigger bugs in code paths that did not expect to be interrupted, including nested NMIs. Using “-c” boosts the rate of NMIs, and using two -c with separate counters encourages nested NMIs and less deterministic behavior.

```
while true; do perf record -c 10000 -e instructions,cycles -a sleep 10;
  ↪done
```

4. Launch a KVM virtual machine.
5. Run 32-bit binaries on systems supporting the SYSCALL instruction. This has been a lightly-tested code path and needs extra scrutiny.

## 20.6 Debugging

Bugs in PTI cause a few different signatures of crashes that are worth noting here.

- Failures of the selftests/x86 code. Usually a bug in one of the more obscure corners of entry\_64.S
- Crashes in early boot, especially around CPU bringup. Bugs in the trampoline code or mappings cause these.
- Crashes at the first interrupt. Caused by bugs in entry\_64.S, like screwing up a page table switch. Also caused by incorrectly mapping the IRQ handler entry code.

- Crashes at the first NMI. The NMI code is separate from main interrupt handlers and can have bugs that do not affect normal interrupts. Also caused by incorrectly mapping NMI code. NMIs that interrupt the entry code must be very careful and can be the cause of crashes that show up when running perf.
- Kernel crashes at the first exit to userspace. `entry_64.S` bugs, or failing to map some of the exit code.
- Crashes at first interrupt that interrupts userspace. The paths in `entry_64.S` that return to userspace are sometimes separate from the ones that return to the kernel.
- Double faults: overflowing the kernel stack because of page faults upon page faults. Caused by touching non-pti-mapped data in the entry code, or forgetting to switch to kernel CR3 before calling into C functions which are not pti-mapped.
- Userspace segfaults early in boot, sometimes manifesting as `mount(8)` failing to mount the rootfs. These have tended to be TLB invalidation issues. Usually invalidating the wrong PCID, or otherwise missing an invalidation.

## **MICROARCHITECTURAL DATA SAMPLING (MDS) MITIGATION**

### **21.1 Overview**

Microarchitectural Data Sampling (MDS) is a family of side channel attacks on internal buffers in Intel CPUs. The variants are:

- Microarchitectural Store Buffer Data Sampling (MSBDS) (CVE-2018-12126)
- Microarchitectural Fill Buffer Data Sampling (MFBDS) (CVE-2018-12130)
- Microarchitectural Load Port Data Sampling (MLPDS) (CVE-2018-12127)
- Microarchitectural Data Sampling Uncacheable Memory (MDSUM) (CVE-2019-11091)

MSBDS leaks Store Buffer Entries which can be speculatively forwarded to a dependent load (store-to-load forwarding) as an optimization. The forward can also happen to a faulting or assisting load operation for a different memory address, which can be exploited under certain conditions. Store buffers are partitioned between Hyper-Threads so cross thread forwarding is not possible. But if a thread enters or exits a sleep state the store buffer is repartitioned which can expose data from one thread to the other.

MFBDS leaks Fill Buffer Entries. Fill buffers are used internally to manage L1 miss situations and to hold data which is returned or sent in response to a memory or I/O operation. Fill buffers can forward data to a load operation and also write data to the cache. When the fill buffer is deallocated it can retain the stale data of the preceding operations which can then be forwarded to a faulting or assisting load operation, which can be exploited under certain conditions. Fill buffers are shared between Hyper-Threads so cross thread leakage is possible.

MLPDS leaks Load Port Data. Load ports are used to perform load operations from memory or I/O. The received data is then forwarded to the register file or a subsequent operation. In some implementations the Load Port can contain stale data from a previous operation which can be forwarded to faulting or assisting loads under certain conditions, which again can be exploited eventually. Load ports are shared between Hyper-Threads so cross thread leakage is possible.

MDSUM is a special case of MSBDS, MFBDS and MLPDS. An uncacheable load from memory that takes a fault or assist can leave data in a microarchitectural structure that may later be observed using one of the same methods used by MSBDS, MFBDS or MLPDS.

## 21.2 Exposure assumptions

It is assumed that attack code resides in user space or in a guest with one exception. The rationale behind this assumption is that the code construct needed for exploiting MDS requires:

- to control the load to trigger a fault or assist
- to have a disclosure gadget which exposes the speculatively accessed data for consumption through a side channel.
- to control the pointer through which the disclosure gadget exposes the data

The existence of such a construct in the kernel cannot be excluded with 100% certainty, but the complexity involved makes it extremely unlikely.

There is one exception, which is untrusted BPF. The functionality of untrusted BPF is limited, but it needs to be thoroughly investigated whether it can be used to create such a construct.

## 21.3 Mitigation strategy

All variants have the same mitigation strategy at least for the single CPU thread case (SMT off): Force the CPU to clear the affected buffers.

This is achieved by using the otherwise unused and obsolete VERW instruction in combination with a microcode update. The microcode clears the affected CPU buffers when the VERW instruction is executed.

For virtualization there are two ways to achieve CPU buffer clearing. Either the modified VERW instruction or via the L1D Flush command. The latter is issued when L1TF mitigation is enabled so the extra VERW can be avoided. If the CPU is not affected by L1TF then VERW needs to be issued.

If the VERW instruction with the supplied segment selector argument is executed on a CPU without the microcode update there is no side effect other than a small number of pointlessly wasted CPU cycles.

This does not protect against cross Hyper-Thread attacks except for MSBDS which is only exploitable cross Hyper-thread when one of the Hyper-Threads enters a C-state.

The kernel provides a function to invoke the buffer clearing:

```
mds_clear_cpu_buffers()
```

The mitigation is invoked on kernel/userspace, hypervisor/guest and C-state (idle) transitions.

As a special quirk to address virtualization scenarios where the host has the microcode updated, but the hypervisor does not (yet) expose the MD\_CLEAR CPUID bit to guests, the kernel issues the VERW instruction in the hope that it might actually clear the buffers. The state is reflected accordingly.

According to current knowledge additional mitigations inside the kernel itself are not required because the necessary gadgets to expose the leaked data cannot be controlled in a way which allows exploitation from malicious user space or VM guests.

## 21.4 Kernel internal mitigation modes

off	Mitigation is disabled. Either the CPU is not affected or mds=off is supplied on the kernel command line
full	Mitigation is enabled. CPU is affected and MD_CLEAR is advertised in CPUID.
vmwerv	Mitigation is enabled. CPU is affected and MD_CLEAR is not advertised in CPUID. That is mainly for virtualization scenarios where the host has the updated microcode but the hypervisor does not expose MD_CLEAR in CPUID. It's a best effort approach without guarantee.

If the CPU is affected and mds=off is not supplied on the kernel command line then the kernel selects the appropriate mitigation mode depending on the availability of the MD\_CLEAR CPUID bit.

## 21.5 Mitigation points

### 21.5.1 1. Return to user space

When transitioning from kernel to user space the CPU buffers are flushed on affected CPUs when the mitigation is not disabled on the kernel command line. The mitigation is enabled through the static key mds\_user\_clear.

The mitigation is invoked in `prepare_exit_to_usermode()` which covers all but one of the kernel to user space transitions. The exception is when we return from a Non Maskable Interrupt (NMI), which is handled directly in `do_nmi()`.

**(The reason that NMI is special is that `prepare_exit_to_usermode()` can enable IRQs. In NMI context, NMIs are blocked, and we don't want to enable IRQs with NMIs blocked.)**

### 21.5.2 2. C-State transition

When a CPU goes idle and enters a C-State the CPU buffers need to be cleared on affected CPUs when SMT is active. This addresses the repartitioning of the store buffer when one of the Hyper-Threads enters a C-State.

When SMT is inactive, i.e. either the CPU does not support it or all sibling threads are offline CPU buffer clearing is not required.

The idle clearing is enabled on CPUs which are only affected by MSBDS and not by any other MDS variant. The other MDS variants cannot be protected against cross Hyper-Thread attacks because the Fill Buffer and the Load Ports are shared. So on CPUs affected by other variants, the idle clearing would be a window dressing exercise and is therefore not activated.

The invocation is controlled by the static key `mds_idle_clear` which is switched depending on the chosen mitigation mode and the SMT state of the system.

The buffer clear is only invoked before entering the C-State to prevent that stale data from the idling CPU from spilling to the Hyper-Thread sibling after the store buffer got repartitioned and all entries are available to the non idle sibling.

When coming out of idle the store buffer is partitioned again so each sibling has half of it available. The back from idle CPU could be then speculatively exposed to contents of the sibling. The buffers are flushed either on exit to user space or on VMENTER so malicious code in user space or the guest cannot speculatively access them.

The mitigation is hooked into all variants of `halt()/mwait()`, but does not cover the legacy ACPI IO-Port mechanism because the ACPI idle driver has been superseded by the `intel_idle` driver around 2010 and is preferred on all affected CPUs which are expected to gain the `MD_CLEAR` functionality in microcode. Aside of that the IO-Port mechanism is a legacy interface which is only used on older systems which are either not affected or do not receive microcode updates anymore.



## THE LINUX MICROCODE LOADER

### Authors

- Fenghua Yu <[fenghua.yu@intel.com](mailto:fenghua.yu@intel.com)>
- Borislav Petkov <[bp@suse.de](mailto:bp@suse.de)>

The kernel has a x86 microcode loading facility which is supposed to provide microcode loading methods in the OS. Potential use cases are updating the microcode on platforms beyond the OEM End-Of-Life support, and updating the microcode on long-running systems without rebooting.

The loader supports three loading methods:

### 22.1 Early load microcode

The kernel can update microcode very early during boot. Loading microcode early can fix CPU issues before they are observed during kernel boot time.

The microcode is stored in an initrd file. During boot, it is read from it and loaded into the CPU cores.

The format of the combined initrd image is microcode in (uncompressed) cpio format followed by the (possibly compressed) initrd image. The loader parses the combined initrd image during boot.

The microcode files in cpio name space are:

**on Intel:** kernel/x86/microcode/GenuineIntel.bin

**on AMD :** kernel/x86/microcode/AuthenticAMD.bin

During BSP (BootStrapping Processor) boot (pre-SMP), the kernel scans the microcode file in the initrd. If microcode matching the CPU is found, it will be applied in the BSP and later on in all APs (Application Processors).

The loader also saves the matching microcode for the CPU in memory. Thus, the cached microcode patch is applied when CPUs resume from a sleep state.

Here's a crude example how to prepare an initrd with microcode (this is normally done automatically by the distribution, when recreating the initrd, so you don't really have to do it yourself. It is documented here for future reference only).

```
#!/bin/bash

if [ -z "$1" ]; then
    echo "You need to supply an initrd file"
    exit 1
fi

INITRD="$1"

DSTDIR=kernel/x86/microcode
TMPDIR=/tmp/initrd

rm -rf $TMPDIR

mkdir $TMPDIR
cd $TMPDIR
mkdir -p $DSTDIR

if [ -d /lib/firmware/amd-ucode ]; then
    cat /lib/firmware/amd-ucode/microcode_amd*.bin > $DSTDIR/AuthenticAMD.
    ↪bin
fi

if [ -d /lib/firmware/intel-ucode ]; then
    cat /lib/firmware/intel-ucode/* > $DSTDIR/GenuineIntel.bin
fi

find . | cpio -o -H newc > ../ucode.cpio
cd ..
mv $INITRD $INITRD.orig
cat ucode.cpio $INITRD.orig > $INITRD

rm -rf $TMPDIR
```

The system needs to have the microcode packages installed into `/lib/firmware` or you need to fixup the paths above if yours are somewhere else and/or you've downloaded them directly from the processor vendor's site.

## 22.2 Late loading

There are two legacy user space interfaces to load microcode, either through `/dev/cpu/microcode` or through `/sys/devices/system/cpu/microcode/reload` file in sysfs.

The `/dev/cpu/microcode` method is deprecated because it needs a special userspace tool for that.

The easier method is simply installing the microcode packages your distro supplies and running:

```
# echo 1 > /sys/devices/system/cpu/microcode/reload
```

as root.

The loading mechanism looks for microcode blobs in `/lib/firmware/{intel-ucode,amd-ucode}`. The default distro installation packages already put them there.

## 22.3 Builtin microcode

The loader supports also loading of a builtin microcode supplied through the regular builtin firmware method `CONFIG_EXTRA_FIRMWARE`. Only 64-bit is currently supported.

Here's an example:

```
CONFIG_EXTRA_FIRMWARE="intel-ucode/06-3a-09 amd-ucode/microcode_amd_fam15h.bin"
CONFIG_EXTRA_FIRMWARE_DIR="/lib/firmware"
```

This basically means, you have the following tree structure locally:

```
/lib/firmware/
|-- amd-ucode
...
|   |-- microcode_amd_fam15h.bin
...
|-- intel-ucode
...
|   |-- 06-3a-09
...
```

so that the build system can find those files and integrate them into the final kernel image. The early loader finds them and applies them.

Needless to say, this method is not the most flexible one because it requires rebuilding the kernel each time updated microcode from the CPU vendor is available.



## USER INTERFACE FOR RESOURCE CONTROL FEATURE

**Copyright** © 2016 Intel Corporation

### Authors

- Fenghua Yu <fenghua.yu@intel.com>
- Tony Luck <tony.luck@intel.com>
- Vikas Shivappa <vikas.shivappa@intel.com>

Intel refers to this feature as Intel Resource Director Technology(Intel(R) RDT). AMD refers to this feature as AMD Platform Quality of Service(AMD QoS).

This feature is enabled by the CONFIG\_X86\_CPU\_RESCTRL and the x86 /proc/cpuinfo flag bits:

RDT (Resource Director Technology) Allocation	"rdt_a"
CAT (Cache Allocation Technology)	"cat_l3", "cat_l2"
CDP (Code and Data Prioritization)	"cdp_l3", "cdp_l2"
CQM (Cache QoS Monitoring)	"cqm_llc", "cqm_occup_llc"
MBM (Memory Bandwidth Monitoring)	"cqm_mbm_total", "cqm_mbm_local"
MBA (Memory Bandwidth Allocation)	"mba"

To use the feature mount the file system:

```
# mount -t resctrl resctrl [-o cdp[,cdpl2][,mba_MBps]] /sys/fs/resctrl
```

mount options are:

**"cdp"**: Enable code/data prioritization in L3 cache allocations.

**"cdpl2"**: Enable code/data prioritization in L2 cache allocations.

**"mba\_MBps"**: Enable the MBA Software Controller(mba\_sc) to specify MBA bandwidth in MBps

L2 and L3 CDP are controlled separately.

RDT features are orthogonal. A particular system may support only monitoring, only control, or both monitoring and control. Cache pseudo-locking is a unique way of using cache control to "pin" or "lock" data in the cache. Details can be found in "Cache Pseudo-Locking".

The mount succeeds if either of allocation or monitoring is present, but only those files and directories supported by the system will be created. For more details on the behavior of the interface during monitoring and allocation, see the "Resource alloc and monitor groups" section.

## 23.1 Info directory

The 'info' directory contains information about the enabled resources. Each resource has its own subdirectory. The subdirectory names reflect the resource names.

Each subdirectory contains the following files with respect to allocation:

Cache resource(L3/L2) subdirectory contains the following files related to allocation:

**"num\_closids"**: The number of CLOSIDs which are valid for this resource. The kernel uses the smallest number of CLOSIDs of all enabled resources as limit.

**"cbm\_mask"**: The bitmask which is valid for this resource. This mask is equivalent to 100%.

**"min\_cbm\_bits"**: The minimum number of consecutive bits which must be set when writing a mask.

**"shareable\_bits"**: Bitmask of shareable resource with other executing entities (e.g. I/O). User can use this when setting up exclusive cache partitions. Note that some platforms support devices that have their own settings for cache use which can over-ride these bits.

**"bit\_usage"**: Annotated capacity bitmasks showing how all instances of the resource are used. The legend is:

**"0"**: Corresponding region is unused. When the system's resources have been allocated and a "0" is found in "bit\_usage" it is a sign that resources are wasted.

**"H"**: Corresponding region is used by hardware only but available for software use. If a resource has bits set in "shareable\_bits" but not all of these bits appear in the resource groups' schematas then the bits appearing in "shareable\_bits" but no resource group will be marked as "H".

**"X"**: Corresponding region is available for sharing and used by hardware and software. These are the bits that appear in "shareable\_bits" as well as a resource group's allocation.

**"S"**: Corresponding region is used by software and available for sharing.

**"E"**: Corresponding region is used exclusively by one resource group. No sharing allowed.

**"P"**: Corresponding region is pseudo-locked. No sharing allowed.

Memory bandwidth(MB) subdirectory contains the following files with respect to allocation:

**"min\_bandwidth"**: The minimum memory bandwidth percentage which user can request.

**"bandwidth\_gran"**: The granularity in which the memory bandwidth percentage is allocated. The allocated b/w percentage is rounded off to the next control step available on the hardware. The available bandwidth control steps are: min\_bandwidth + N \* bandwidth\_gran.

**"delay\_linear"**: Indicates if the delay scale is linear or non-linear. This field is purely informational only.

**"thread\_throttle\_mode"**: Indicator on Intel systems of how tasks running on threads of a physical core are throttled in cases where they request different memory bandwidth percentages:

**"max"**: the smallest percentage is applied to all threads

**“per-thread”:** bandwidth percentages are directly applied to the threads running on the core

If RDT monitoring is available there will be an “L3\_MON” directory with the following files:

**“num\_rmid”:** The number of RMIDs available. This is the upper bound for how many “CTRL\_MON” + “MON” groups can be created.

**“mon\_features”:** Lists the monitoring events if monitoring is enabled for the resource.

**“max\_threshold\_occupancy”:** Read/write file provides the largest value (in bytes) at which a previously used LLC\_occupancy counter can be considered for re-use.

Finally, in the top level of the “info” directory there is a file named “last\_cmd\_status”. This is reset with every “command” issued via the file system (making new directories or writing to any of the control files). If the command was successful, it will read as “ok”. If the command failed, it will provide more information that can be conveyed in the error returns from file operations. E.g.

```
# echo L3:0=f7 > schemata
bash: echo: write error: Invalid argument
# cat info/last_cmd_status
mask f7 has non-consecutive 1-bits
```

## 23.2 Resource alloc and monitor groups

Resource groups are represented as directories in the resctrl file system. The default group is the root directory which, immediately after mounting, owns all the tasks and cpus in the system and can make full use of all resources.

On a system with RDT control features additional directories can be created in the root directory that specify different amounts of each resource (see “schemata” below). The root and these additional top level directories are referred to as “CTRL\_MON” groups below.

On a system with RDT monitoring the root directory and other top level directories contain a directory named “mon\_groups” in which additional directories can be created to monitor subsets of tasks in the CTRL\_MON group that is their ancestor. These are called “MON” groups in the rest of this document.

Removing a directory will move all tasks and cpus owned by the group it represents to the parent. Removing one of the created CTRL\_MON groups will automatically remove all MON groups below it.

All groups contain the following files:

**“tasks”:** Reading this file shows the list of all tasks that belong to this group. Writing a task id to the file will add a task to the group. If the group is a CTRL\_MON group the task is removed from whichever previous CTRL\_MON group owned the task and also from any MON group that owned the task. If the group is a MON group, then the task must already belong to the CTRL\_MON parent of this group. The task is removed from any previous MON group.

**“cpus”:** Reading this file shows a bitmask of the logical CPUs owned by this group. Writing a mask to this file will add and remove CPUs to/from this group. As with the tasks file a hierarchy is maintained where MON groups may only include CPUs owned by the parent

CTRL\_MON group. When the resource group is in pseudo-locked mode this file will only be readable, reflecting the CPUs associated with the pseudo-locked region.

**“cpus\_list”**: Just like “cpus”, only using ranges of CPUs instead of bitmasks.

When control is enabled all CTRL\_MON groups will also contain:

**“schemata”**: A list of all the resources available to this group. Each resource has its own line and format - see below for details.

**“size”**: Mirrors the display of the “schemata” file to display the size in bytes of each allocation instead of the bits representing the allocation.

**“mode”**: The “mode” of the resource group dictates the sharing of its allocations. A “shareable” resource group allows sharing of its allocations while an “exclusive” resource group does not. A cache pseudo-locked region is created by first writing “pseudo-locksetup” to the “mode” file before writing the cache pseudo-locked region’s schemata to the resource group’s “schemata” file. On successful pseudo-locked region creation the mode will automatically change to “pseudo-locked”.

When monitoring is enabled all MON groups will also contain:

**“mon\_data”**: This contains a set of files organized by L3 domain and by RDT event. E.g. on a system with two L3 domains there will be subdirectories “mon\_L3\_00” and “mon\_L3\_01”. Each of these directories have one file per event (e.g. “llc\_occupancy”, “mbm\_total\_bytes”, and “mbm\_local\_bytes”). In a MON group these files provide a read out of the current value of the event for all tasks in the group. In CTRL\_MON groups these files provide the sum for all tasks in the CTRL\_MON group and all tasks in MON groups. Please see example section for more details on usage.

### 23.2.1 Resource allocation rules

When a task is running the following rules define which resources are available to it:

- 1) If the task is a member of a non-default group, then the schemata for that group is used.
- 2) Else if the task belongs to the default group, but is running on a CPU that is assigned to some specific group, then the schemata for the CPU’s group is used.
- 3) Otherwise the schemata for the default group is used.

### 23.2.2 Resource monitoring rules

- 1) If a task is a member of a MON group, or non-default CTRL\_MON group then RDT events for the task will be reported in that group.
- 2) If a task is a member of the default CTRL\_MON group, but is running on a CPU that is assigned to some specific group, then the RDT events for the task will be reported in that group.
- 3) Otherwise RDT events for the task will be reported in the root level “mon\_data” group.



## 23.3 Notes on cache occupancy monitoring and control

When moving a task from one group to another you should remember that this only affects *new* cache allocations by the task. E.g. you may have a task in a monitor group showing 3 MB of cache occupancy. If you move to a new group and immediately check the occupancy of the old and new groups you will likely see that the old group is still showing 3 MB and the new group zero. When the task accesses locations still in cache from before the move, the h/w does not update any counters. On a busy system you will likely see the occupancy in the old group go down as cache lines are evicted and re-used while the occupancy in the new group rises as the task accesses memory and loads into the cache are counted based on membership in the new group.

The same applies to cache allocation control. Moving a task to a group with a smaller cache partition will not evict any cache lines. The process may continue to use them from the old partition.

Hardware uses CLOSid(Class of service ID) and an RMID(Resource monitoring ID) to identify a control group and a monitoring group respectively. Each of the resource groups are mapped to these IDs based on the kind of group. The number of CLOSid and RMID are limited by the hardware and hence the creation of a “CTRL\_MON” directory may fail if we run out of either CLOSID or RMID and creation of “MON” group may fail if we run out of RMIDs.

### 23.3.1 max\_threshold\_occupancy - generic concepts

Note that an RMID once freed may not be immediately available for use as the RMID is still tagged the cache lines of the previous user of RMID. Hence such RMIDs are placed on limbo list and checked back if the cache occupancy has gone down. If there is a time when system has a lot of limbo RMIDs but which are not ready to be used, user may see an -EBUSY during mkdir.

max\_threshold\_occupancy is a user configurable value to determine the occupancy at which an RMID can be freed.

### 23.3.2 Schemata files - general concepts

Each line in the file describes one resource. The line starts with the name of the resource, followed by specific values to be applied in each of the instances of that resource on the system.

### 23.3.3 Cache IDs

On current generation systems there is one L3 cache per socket and L2 caches are generally just shared by the hyperthreads on a core, but this isn't an architectural requirement. We could have multiple separate L3 caches on a socket, multiple cores could share an L2 cache. So instead of using “socket” or “core” to define the set of logical cpus sharing a resource we use a “Cache ID”. At a given cache level this will be a unique number across the whole system (but it isn't guaranteed to be a contiguous sequence, there may be gaps). To find the ID for each logical CPU look in /sys/devices/system/cpu/cpu\*/cache/index\*/id

### 23.3.4 Cache Bit Masks (CBM)

For cache resources we describe the portion of the cache that is available for allocation using a bitmask. The maximum value of the mask is defined by each cpu model (and may be different for different cache levels). It is found using CPUID, but is also provided in the “info” directory of the resctrl file system in “info/{resource}/cbm\_mask”. Intel hardware requires that these masks have all the ‘1’ bits in a contiguous block. So 0x3, 0x6 and 0xC are legal 4-bit masks with two bits set, but 0x5, 0x9 and 0xA are not. On a system with a 20-bit mask each bit represents 5% of the capacity of the cache. You could partition the cache into four equal parts with masks: 0x1f, 0x3e0, 0x7c00, 0xf8000.

## 23.4 Memory bandwidth Allocation and monitoring

For Memory bandwidth resource, by default the user controls the resource by indicating the percentage of total memory bandwidth.

The minimum bandwidth percentage value for each cpu model is predefined and can be looked up through “info/MB/min\_bandwidth”. The bandwidth granularity that is allocated is also dependent on the cpu model and can be looked up at “info/MB/bandwidth\_gran”. The available bandwidth control steps are:  $\text{min\_bw} + N * \text{bw\_gran}$ . Intermediate values are rounded to the next control step available on the hardware.

The bandwidth throttling is a core specific mechanism on some of Intel SKUs. Using a high bandwidth and a low bandwidth setting on two threads sharing a core may result in both threads being throttled to use the low bandwidth (see “thread\_throttle\_mode”).

The fact that Memory bandwidth allocation(MBA) may be a core specific mechanism where as memory bandwidth monitoring(MBM) is done at the package level may lead to confusion when users try to apply control via the MBA and then monitor the bandwidth to see if the controls are effective. Below are such scenarios:

1. User may *not* see increase in actual bandwidth when percentage values are increased:

This can occur when aggregate L2 external bandwidth is more than L3 external bandwidth. Consider an SKL SKU with 24 cores on a package and where L2 external is 10GBps (hence aggregate L2 external bandwidth is 240GBps) and L3 external bandwidth is 100GBps. Now a workload with ‘20 threads, having 50% bandwidth, each consuming 5GBps’ consumes the max L3 bandwidth of 100GBps although the percentage value specified is only 50% << 100%. Hence increasing the bandwidth percentage will not yield any more bandwidth. This is because although the L2 external bandwidth still has capacity, the L3 external bandwidth is fully used. Also note that this would be dependent on number of cores the benchmark is run on.

2. Same bandwidth percentage may mean different actual bandwidth depending on # of threads:

For the same SKU in #1, a ‘single thread, with 10% bandwidth’ and ‘4 thread, with 10% bandwidth’ can consume upto 10GBps and 40GBps although they have same percentage bandwidth of 10%. This is simply because as threads start using more cores in an rdtgroup, the actual bandwidth may increase or vary although user specified bandwidth percentage is same.

In order to mitigate this and make the interface more user friendly, resctrl added support for specifying the bandwidth in MBps as well. The kernel underneath would use a software feedback mechanism or a “Software Controller(mba\_sc)” which reads the actual bandwidth using MBM counters and adjust the memory bandwidth percentages to ensure:

```
"actual bandwidth < user specified bandwidth".
```

By default, the schemata would take the bandwidth percentage values where as user can switch to the “MBA software controller” mode using a mount option ‘mba\_MBps’. The schemata format is specified in the below sections.

### 23.4.1 L3 schemata file details (code and data prioritization disabled)

With CDP disabled the L3 schemata format is:

```
L3:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...
```

### 23.4.2 L3 schemata file details (CDP enabled via mount option to resctrl)

When CDP is enabled L3 control is split into two separate resources so you can specify independent masks for code and data like this:

```
L3DATA:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...
L3CODE:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...
```

### 23.4.3 L2 schemata file details

CDP is supported at L2 using the ‘cdpl2’ mount option. The schemata format is either:

```
L2:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...
```

or

```
L2DATA:<cache_id0>=<cbm>;<cache_id1>=<cbm>;... L2CODE:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...
```

### 23.4.4 Memory bandwidth Allocation (default mode)

Memory b/w domain is L3 cache.

```
MB:<cache_id0>=bandwidth0;<cache_id1>=bandwidth1;...
```

### 23.4.5 Memory bandwidth Allocation specified in MBps

Memory bandwidth domain is L3 cache.

```
MB:<cache_id0>=bw_MBps0;<cache_id1>=bw_MBps1;...
```

### 23.4.6 Reading/writing the schemata file

Reading the schemata file will show the state of all resources on all domains. When writing you only need to specify those values which you wish to change. E.g.

```
# cat schemata
L3DATA:0=fffff;1=fffff;2=fffff;3=fffff
L3CODE:0=fffff;1=fffff;2=fffff;3=fffff
# echo "L3DATA:2=3c0;" > schemata
# cat schemata
L3DATA:0=fffff;1=fffff;2=3c0;3=fffff
L3CODE:0=fffff;1=fffff;2=fffff;3=fffff
```

## 23.5 Cache Pseudo-Locking

CAT enables a user to specify the amount of cache space that an application can fill. Cache pseudo-locking builds on the fact that a CPU can still read and write data pre-allocated outside its current allocated area on a cache hit. With cache pseudo-locking, data can be preloaded into a reserved portion of cache that no application can fill, and from that point on will only serve cache hits. The cache pseudo-locked memory is made accessible to user space where an application can map it into its virtual address space and thus have a region of memory with reduced average read latency.

The creation of a cache pseudo-locked region is triggered by a request from the user to do so that is accompanied by a schemata of the region to be pseudo-locked. The cache pseudo-locked region is created as follows:

- Create a CAT allocation CLOSNEW with a CBM matching the schemata from the user of the cache region that will contain the pseudo-locked memory. This region must not overlap with any current CAT allocation/CLOS on the system and no future overlap with this cache region is allowed while the pseudo-locked region exists.
- Create a contiguous region of memory of the same size as the cache region.
- Flush the cache, disable hardware prefetchers, disable preemption.
- Make CLOSNEW the active CLOS and touch the allocated memory to load it into the cache.
- Set the previous CLOS as active.
- At this point the closid CLOSNEW can be released - the cache pseudo-locked region is protected as long as its CBM does not appear in any CAT allocation. Even though the cache pseudo-locked region will from this point on not appear in any CBM of any CLOS an application running with any CLOS will be able to access the memory in the pseudo-locked region since the region continues to serve cache hits.
- The contiguous region of memory loaded into the cache is exposed to user-space as a character device.

Cache pseudo-locking increases the probability that data will remain in the cache via carefully configuring the CAT feature and controlling application behavior. There is no guarantee that data is placed in cache. Instructions like INVLD, WBINVD, CLFLUSH, etc. can still evict “locked” data from cache. Power management C-states may shrink or power off cache. Deeper C-states will automatically be restricted on pseudo-locked region creation.

It is required that an application using a pseudo-locked region runs with affinity to the cores (or a subset of the cores) associated with the cache on which the pseudo-locked region resides. A sanity check within the code will not allow an application to map pseudo-locked memory unless it runs with affinity to cores associated with the cache on which the pseudo-locked region resides. The sanity check is only done during the initial `mmap()` handling, there is no enforcement afterwards and the application self needs to ensure it remains affine to the correct cores.

Pseudo-locking is accomplished in two stages:

- 1) During the first stage the system administrator allocates a portion of cache that should be dedicated to pseudo-locking. At this time an equivalent portion of memory is allocated, loaded into allocated cache portion, and exposed as a character device.
- 2) During the second stage a user-space application maps (`mmap()`) the pseudo-locked memory into its address space.

### 23.5.1 Cache Pseudo-Locking Interface

A pseudo-locked region is created using the `resctrl` interface as follows:

- 1) Create a new resource group by creating a new directory in `/sys/fs/resctrl`.
- 2) Change the new resource group's mode to "pseudo-locksetup" by writing "pseudo-locksetup" to the "mode" file.
- 3) Write the schemata of the pseudo-locked region to the "schemata" file. All bits within the schemata should be "unused" according to the "bit\_usage" file.

On successful pseudo-locked region creation the "mode" file will contain "pseudo-locked" and a new character device with the same name as the resource group will exist in `/dev/pseudo_lock`. This character device can be `mmap()`'ed by user space in order to obtain access to the pseudo-locked memory region.

An example of cache pseudo-locked region creation and usage can be found below.

### 23.5.2 Cache Pseudo-Locking Debugging Interface

The pseudo-locking debugging interface is enabled by default (if `CONFIG_DEBUG_FS` is enabled) and can be found in `/sys/kernel/debug/resctrl`.

There is no explicit way for the kernel to test if a provided memory location is present in the cache. The pseudo-locking debugging interface uses the tracing infrastructure to provide two ways to measure cache residency of the pseudo-locked region:

- 1) Memory access latency using the `pseudo_lock_mem_latency` tracepoint. Data from these measurements are best visualized using a hist trigger (see example below). In this test the pseudo-locked region is traversed at a stride of 32 bytes while hardware prefetchers and preemption are disabled. This also provides a substitute visualization of cache hits and misses.
- 2) Cache hit and miss measurements using model specific precision counters if available. Depending on the levels of cache on the system the `pseudo_lock_l2` and `pseudo_lock_l3` tracepoints are available.

When a pseudo-locked region is created a new `debugfs` directory is created for it in `debugfs` as `/sys/kernel/debug/resctrl/<newdir>`. A single write-only file, `pseudo_lock_measure`, is present

in this directory. The measurement of the pseudo-locked region depends on the number written to this debugfs file:

- 1: writing “1” to the `pseudo_lock_measure` file will trigger the latency measurement captured in the `pseudo_lock_mem_latency` tracepoint. See example below.
- 2: writing “2” to the `pseudo_lock_measure` file will trigger the L2 cache residency (cache hits and misses) measurement captured in the `pseudo_lock_l2` tracepoint. See example below.
- 3: writing “3” to the `pseudo_lock_measure` file will trigger the L3 cache residency (cache hits and misses) measurement captured in the `pseudo_lock_l3` tracepoint.

All measurements are recorded with the tracing infrastructure. This requires the relevant tracepoints to be enabled before the measurement is triggered.

### Example of latency debugging interface

In this example a pseudo-locked region named “newlock” was created. Here is how we can measure the latency in cycles of reading from this region and visualize this data with a histogram that is available if `CONFIG_HIST_TRIGGERS` is set:

```
# :> /sys/kernel/debug/tracing/trace
# echo 'hist:keys=latency' > /sys/kernel/debug/tracing/events/resctrl/pseudo_
↳lock_mem_latency/trigger
# echo 1 > /sys/kernel/debug/tracing/events/resctrl/pseudo_lock_mem_latency/
↳enable
# echo 1 > /sys/kernel/debug/resctrl/newlock/pseudo_lock_measure
# echo 0 > /sys/kernel/debug/tracing/events/resctrl/pseudo_lock_mem_latency/
↳enable
# cat /sys/kernel/debug/tracing/events/resctrl/pseudo_lock_mem_latency/hist

# event histogram
#
# trigger info: hist:keys=latency:vals=hitcount:sort=hitcount:size=2048_
↳[active]
#

{ latency:      456 } hitcount:      1
{ latency:      50 } hitcount:     83
{ latency:      36 } hitcount:     96
{ latency:      44 } hitcount:    174
{ latency:      48 } hitcount:    195
{ latency:      46 } hitcount:    262
{ latency:      42 } hitcount:    693
{ latency:      40 } hitcount:   3204
{ latency:      38 } hitcount:   3484

Totals:
  Hits: 8192
  Entries: 9
  Dropped: 0
```

## Example of cache hits/misses debugging

In this example a pseudo-locked region named “newlock” was created on the L2 cache of a platform. Here is how we can obtain details of the cache hits and misses using the platform’s precision counters.

```
# :> /sys/kernel/debug/tracing/trace
# echo 1 > /sys/kernel/debug/tracing/events/resctrl/pseudo_lock_l2/enable
# echo 2 > /sys/kernel/debug/resctrl/newlock/pseudo_lock_measure
# echo 0 > /sys/kernel/debug/tracing/events/resctrl/pseudo_lock_l2/enable
# cat /sys/kernel/debug/tracing/trace

# tracer: nop
#
#          _-----=> irqs-off
#          / _-----=> need-resched
#          | / _-----=> hardirq/softirq
#          || / _-----=> preempt-depth
#          ||| / _-----=> delay
#          TASK-PID   CPU#  ||||   TIMESTAMP   FUNCTION
#          |   |   |   |   |   |   |   |
pseudo_lock_mea-1672  [002]  ....   3132.860500: pseudo_lock_l2: hits=4097 miss=0
```

## Examples for RDT allocation usage

### 1) Example 1

On a two socket machine (one L3 cache per socket) with just four bits for cache bit masks, minimum b/w of 10% with a memory bandwidth granularity of 10%.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir p0 p1
# echo "L3:0=3;1=c\nMB:0=50;1=50" > /sys/fs/resctrl/p0/schemata
# echo "L3:0=3;1=3\nMB:0=50;1=50" > /sys/fs/resctrl/p1/schemata
```

The default resource group is unmodified, so we have access to all parts of all caches (its schemata file reads “L3:0=f;1=f”).

Tasks that are under the control of group “p0” may only allocate from the “lower” 50% on cache ID 0, and the “upper” 50% of cache ID 1. Tasks in group “p1” use the “lower” 50% of cache on both sockets.

Similarly, tasks that are under the control of group “p0” may use a maximum memory b/w of 50% on socket0 and 50% on socket 1. Tasks in group “p1” may also use 50% memory b/w on both sockets. Note that unlike cache masks, memory b/w cannot specify whether these allocations can overlap or not. The allocations specifies the maximum b/w that the group may be able to use and the system admin can configure the b/w accordingly.

If resctrl is using the software controller (mba\_sc) then user can enter the max b/w in MB rather than the percentage values.



```
# echo "L3:0=3;1=c\nMB:0=1024;1=500" > /sys/fs/resctrl/p0/schemata
# echo "L3:0=3;1=3\nMB:0=1024;1=500" > /sys/fs/resctrl/p1/schemata
```

In the above example the tasks in “p1” and “p0” on socket 0 would use a max b/w of 1024MB where as on socket 1 they would use 500MB.

### 2) Example 2

Again two sockets, but this time with a more realistic 20-bit mask.

Two real time tasks pid=1234 running on processor 0 and pid=5678 running on processor 1 on socket 0 on a 2-socket and dual core machine. To avoid noisy neighbors, each of the two real-time tasks exclusively occupies one quarter of L3 cache on socket 0.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
```

First we reset the schemata for the default group so that the “upper” 50% of the L3 cache on socket 0 and 50% of memory b/w cannot be used by ordinary tasks:

```
# echo "L3:0=3ff;1=ffffff\nMB:0=50;1=100" > schemata
```

Next we make a resource group for our first real time task and give it access to the “top” 25% of the cache on socket 0.

```
# mkdir p0
# echo "L3:0=f8000;1=ffffff" > p0/schemata
```

Finally we move our first real time task into this resource group. We also use taskset(1) to ensure the task always runs on a dedicated CPU on socket 0. Most uses of resource groups will also constrain which processors tasks run on.

```
# echo 1234 > p0/tasks
# taskset -cp 1 1234
```

Ditto for the second real time task (with the remaining 25% of cache):

```
# mkdir p1
# echo "L3:0=7c00;1=ffffff" > p1/schemata
# echo 5678 > p1/tasks
# taskset -cp 2 5678
```

For the same 2 socket system with memory b/w resource and CAT L3 the schemata would look like(Assume min\_bandwidth 10 and bandwidth\_gran is 10):

For our first real time task this would request 20% memory b/w on socket 0.

```
# echo -e "L3:0=f8000;1=ffffff\nMB:0=20;1=100" > p0/schemata
```

For our second real time task this would request an other 20% memory b/w on socket 0.

```
# echo -e "L3:0=f8000;1=ffffff\nMB:0=20;1=100" > p0/schemata
```

### 3) Example 3



A single socket system which has real-time tasks running on core 4-7 and non real-time workload assigned to core 0-3. The real-time tasks share text and data, so a per task association is not required and due to interaction with the kernel it's desired that the kernel on these cores shares L3 with the tasks.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
```

First we reset the schemata for the default group so that the “upper” 50% of the L3 cache on socket 0, and 50% of memory bandwidth on socket 0 cannot be used by ordinary tasks:

```
# echo "L3:0=3ff\nMB:0=50" > schemata
```

Next we make a resource group for our real time cores and give it access to the “top” 50% of the cache on socket 0 and 50% of memory bandwidth on socket 0.

```
# mkdir p0
# echo "L3:0=ffc00\nMB:0=50" > p0/schemata
```

Finally we move core 4-7 over to the new group and make sure that the kernel and the tasks running there get 50% of the cache. They should also get 50% of memory bandwidth assuming that the cores 4-7 are SMT siblings and only the real time threads are scheduled on the cores 4-7.

```
# echo F0 > p0/cpus
```

#### 4) Example 4

The resource groups in previous examples were all in the default “shareable” mode allowing sharing of their cache allocations. If one resource group configures a cache allocation then nothing prevents another resource group to overlap with that allocation.

In this example a new exclusive resource group will be created on a L2 CAT system with two L2 cache instances that can be configured with an 8-bit capacity bitmask. The new exclusive resource group will be configured to use 25% of each cache instance.

```
# mount -t resctrl resctrl /sys/fs/resctrl/
# cd /sys/fs/resctrl
```

First, we observe that the default group is configured to allocate to all L2 cache:

```
# cat schemata
L2:0=ff;1=ff
```

We could attempt to create the new resource group at this point, but it will fail because of the overlap with the schemata of the default group:

```
# mkdir p0
# echo 'L2:0=0x3;1=0x3' > p0/schemata
# cat p0/mode
shareable
# echo exclusive > p0/mode
-sh: echo: write error: Invalid argument
```

```
# cat info/last_cmd_status
schemata overlaps
```

To ensure that there is no overlap with another resource group the default resource group's schemata has to change, making it possible for the new resource group to become exclusive.

```
# echo 'L2:0=0xfc;1=0xfc' > schemata
# echo exclusive > p0/mode
# grep . p0/*
p0/cpus:0
p0/mode:exclusive
p0/schemata:L2:0=03;1=03
p0/size:L2:0=262144;1=262144
```

A new resource group will on creation not overlap with an exclusive resource group:

```
# mkdir p1
# grep . p1/*
p1/cpus:0
p1/mode:shareable
p1/schemata:L2:0=fc;1=fc
p1/size:L2:0=786432;1=786432
```

The bit\_usage will reflect how the cache is used:

```
# cat info/L2/bit_usage
0=SSSSSSEE;1=SSSSSSEE
```

A resource group cannot be forced to overlap with an exclusive resource group:

```
# echo 'L2:0=0x1;1=0x1' > p1/schemata
-sh: echo: write error: Invalid argument
# cat info/last_cmd_status
overlaps with exclusive group
```

### Example of Cache Pseudo-Locking

Lock portion of L2 cache from cache id 1 using CBM 0x3. Pseudo-locked region is exposed at /dev/pseudo\_lock/newlock that can be provided to application for argument to mmap().

```
# mount -t resctrl resctrl /sys/fs/resctrl/
# cd /sys/fs/resctrl
```

Ensure that there are bits available that can be pseudo-locked, since only unused bits can be pseudo-locked the bits to be pseudo-locked needs to be removed from the default resource group's schemata:

```
# cat info/L2/bit_usage
0=SSSSSSSS;1=SSSSSSSS
# echo 'L2:1=0xfc' > schemata
```

```
# cat info/L2/bit_usage
0=SSSSSSSS;1=SSSSSS00
```

Create a new resource group that will be associated with the pseudo-locked region, indicate that it will be used for a pseudo-locked region, and configure the requested pseudo-locked region capacity bitmask:

```
# mkdir newlock
# echo pseudo-locksetup > newlock/mode
# echo 'L2:1=0x3' > newlock/schemata
```

On success the resource group's mode will change to pseudo-locked, the bit\_usage will reflect the pseudo-locked region, and the character device exposing the pseudo-locked region will exist:

```
# cat newlock/mode
pseudo-locked
# cat info/L2/bit_usage
0=SSSSSSSS;1=SSSSSSPP
# ls -l /dev/pseudo_lock/newlock
crw----- 1 root root 243, 0 Apr  3 05:01 /dev/pseudo_lock/newlock
```

```
/*
 * Example code to access one page of pseudo-locked cache region
 * from user space.
 */
#define _GNU_SOURCE
#include <fcntl.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

/*
 * It is required that the application runs with affinity to only
 * cores associated with the pseudo-locked region. Here the cpu
 * is hardcoded for convenience of example.
 */
static int cpuid = 2;

int main(int argc, char *argv[])
{
    cpu_set_t cpuset;
    long page_size;
    void *mapping;
    int dev_fd;
    int ret;

    page_size = sysconf(_SC_PAGESIZE);
```

```
CPU_ZERO(&cpuset);
CPU_SET(cpuid, &cpuset);
ret = sched_setaffinity(0, sizeof(cpuset), &cpuset);
if (ret < 0) {
    perror("sched_setaffinity");
    exit(EXIT_FAILURE);
}

dev_fd = open("/dev/pseudo_lock/newlock", O_RDWR);
if (dev_fd < 0) {
    perror("open");
    exit(EXIT_FAILURE);
}

mapping = mmap(0, page_size, PROT_READ | PROT_WRITE, MAP_SHARED,
               dev_fd, 0);
if (mapping == MAP_FAILED) {
    perror("mmap");
    close(dev_fd);
    exit(EXIT_FAILURE);
}

/* Application interacts with pseudo-locked memory @mapping */

ret = munmap(mapping, page_size);
if (ret < 0) {
    perror("munmap");
    close(dev_fd);
    exit(EXIT_FAILURE);
}

close(dev_fd);
exit(EXIT_SUCCESS);
}
```

### 23.5.3 Locking between applications

Certain operations on the resctrl filesystem, composed of read/writes to/from multiple files, must be atomic.

As an example, the allocation of an exclusive reservation of L3 cache involves:

1. Read the cbmmasks from each directory or the per-resource “bit\_usage”
2. Find a contiguous set of bits in the global CBM bitmask that is clear in any of the directory cbmmasks
3. Create a new directory
4. Set the bits found in step 2 to the new directory “schemata” file

If two applications attempt to allocate space concurrently then they can end up allocating the same bits so the reservations are shared instead of exclusive.

To coordinate atomic operations on the resctrlfs and to avoid the problem above, the following locking procedure is recommended:

Locking is based on flock, which is available in libc and also as a shell script command

Write lock:

- A) Take flock(LOCK\_EX) on /sys/fs/resctrl
- B) Read/write the directory structure.
- C) funlock

Read lock:

- A) Take flock(LOCK\_SH) on /sys/fs/resctrl
- B) If success read the directory structure.
- C) funlock

Example with bash:

```
# Atomically read directory structure
$ flock -s /sys/fs/resctrl/ find /sys/fs/resctrl

# Read directory contents and create new subdirectory

$ cat create-dir.sh
find /sys/fs/resctrl/ > output.txt
mask = function-of(output.txt)
mkdir /sys/fs/resctrl/newres/
echo mask > /sys/fs/resctrl/newres/schemata

$ flock /sys/fs/resctrl/ ./create-dir.sh
```

Example with C:

```
/*
 * Example code do take advisory locks
 * before accessing resctrl filesystem
 */
#include <sys/file.h>
#include <stdlib.h>

void resctrl_take_shared_lock(int fd)
{
    int ret;

    /* take shared lock on resctrl filesystem */
    ret = flock(fd, LOCK_SH);
    if (ret) {
        perror("flock");
        exit(-1);
    }
}
```

```
void resctrl_take_exclusive_lock(int fd)
{
    int ret;

    /* release lock on resctrl filesystem */
    ret = flock(fd, LOCK_EX);
    if (ret) {
        perror("flock");
        exit(-1);
    }
}

void resctrl_release_lock(int fd)
{
    int ret;

    /* take shared lock on resctrl filesystem */
    ret = flock(fd, LOCK_UN);
    if (ret) {
        perror("flock");
        exit(-1);
    }
}

void main(void)
{
    int fd, ret;

    fd = open("/sys/fs/resctrl", O_DIRECTORY);
    if (fd == -1) {
        perror("open");
        exit(-1);
    }
    resctrl_take_shared_lock(fd);
    /* code to read directory contents */
    resctrl_release_lock(fd);

    resctrl_take_exclusive_lock(fd);
    /* code to read and write directory contents */
    resctrl_release_lock(fd);
}
```

## 23.6 Examples for RDT Monitoring along with allocation usage

### 23.6.1 Reading monitored data

Reading an event file (for ex: `mon_data/mon_L3_00/llc_occupancy`) would show the current snapshot of LLC occupancy of the corresponding MON group or CTRL\_MON group.

### 23.6.2 Example 1 (Monitor CTRL\_MON group and subset of tasks in CTRL\_MON group)

On a two socket machine (one L3 cache per socket) with just four bits for cache bit masks:

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir p0 p1
# echo "L3:0=3;1=c" > /sys/fs/resctrl/p0/schemata
# echo "L3:0=3;1=3" > /sys/fs/resctrl/p1/schemata
# echo 5678 > p1/tasks
# echo 5679 > p1/tasks
```

The default resource group is unmodified, so we have access to all parts of all caches (its schemata file reads "L3:0=f;1=f").

Tasks that are under the control of group "p0" may only allocate from the "lower" 50% on cache ID 0, and the "upper" 50% of cache ID 1. Tasks in group "p1" use the "lower" 50% of cache on both sockets.

Create monitor groups and assign a subset of tasks to each monitor group.

```
# cd /sys/fs/resctrl/p1/mon_groups
# mkdir m11 m12
# echo 5678 > m11/tasks
# echo 5679 > m12/tasks
```

fetch data (data shown in bytes)

```
# cat m11/mon_data/mon_L3_00/llc_occupancy
16234000
# cat m11/mon_data/mon_L3_01/llc_occupancy
14789000
# cat m12/mon_data/mon_L3_00/llc_occupancy
16789000
```

The parent ctrl\_mon group shows the aggregated data.

```
# cat /sys/fs/resctrl/p1/mon_data/mon_l3_00/llc_occupancy
31234000
```

### 23.6.3 Example 2 (Monitor a task from its creation)

On a two socket machine (one L3 cache per socket):

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir p0 p1
```

An RMID is allocated to the group once its created and hence the <cmd> below is monitored from its creation.

```
# echo $$ > /sys/fs/resctrl/p1/tasks
# <cmd>
```

Fetch the data:

```
# cat /sys/fs/resctrl/p1/mon_data/mon_l3_00/llc_occupancy
31789000
```

### 23.6.4 Example 3 (Monitor without CAT support or before creating CAT groups)

Assume a system like HSW has only CQM and no CAT support. In this case the resctrl will still mount but cannot create CTRL\_MON directories. But user can create different MON groups within the root group thereby able to monitor all tasks including kernel threads.

This can also be used to profile jobs cache size footprint before being able to allocate them to different allocation groups.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir mon_groups/m01
# mkdir mon_groups/m02

# echo 3478 > /sys/fs/resctrl/mon_groups/m01/tasks
# echo 2467 > /sys/fs/resctrl/mon_groups/m02/tasks
```

Monitor the groups separately and also get per domain data. From the below its apparent that the tasks are mostly doing work on domain(socket) 0.

```
# cat /sys/fs/resctrl/mon_groups/m01/mon_L3_00/llc_occupancy
31234000
# cat /sys/fs/resctrl/mon_groups/m01/mon_L3_01/llc_occupancy
34555
# cat /sys/fs/resctrl/mon_groups/m02/mon_L3_00/llc_occupancy
31234000
# cat /sys/fs/resctrl/mon_groups/m02/mon_L3_01/llc_occupancy
32789
```



### 23.6.5 Example 4 (Monitor real time tasks)

A single socket system which has real time tasks running on cores 4-7 and non real time tasks on other cpus. We want to monitor the cache occupancy of the real time threads on these cores.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir p1
```

Move the cpus 4-7 over to p1:

```
# echo f0 > p1/cpus
```

View the llc occupancy snapshot:

```
# cat /sys/fs/resctrl/p1/mon_data/mon_L3_00/llc_occupancy
11234000
```

## 23.7 Intel RDT Errata

### 23.7.1 Intel MBM Counters May Report System Memory Bandwidth Incorrectly

Errata SKX99 for Skylake server and BDF102 for Broadwell server.

**Problem:** Intel Memory Bandwidth Monitoring (MBM) counters track metrics according to the assigned Resource Monitor ID (RMID) for that logical core. The IA32\_QM\_CTR register (MSR 0xC8E), used to report these metrics, may report incorrect system bandwidth for certain RMID values.

**Implication:** Due to the errata, system memory bandwidth may not match what is reported.

**Workaround:** MBM total and local readings are corrected according to the following correction factor table:

core count	rmid count	rmid threshold	correction factor
1	8	0	1.000000
2	16	0	1.000000
3	24	15	0.969650
4	32	0	1.000000
6	48	31	0.969650
7	56	47	1.142857
8	64	0	1.000000
9	72	63	1.185115
10	80	63	1.066553
11	88	79	1.454545
12	96	0	1.000000
13	104	95	1.230769
14	112	95	1.142857
15	120	95	1.066667
16	128	0	1.000000
17	136	127	1.254863
18	144	127	1.185255
19	152	0	1.000000
20	160	127	1.066667
21	168	0	1.000000
22	176	159	1.454334
23	184	0	1.000000
24	192	127	0.969744
25	200	191	1.280246
26	208	191	1.230921
27	216	0	1.000000
28	224	191	1.143118

If  $\text{rmid} > \text{rmid threshold}$ , MBM total and local values should be multiplied by the correction factor.

See:

1. Erratum SKX99 in Intel Xeon Processor Scalable Family Specification Update: <http://web.archive.org/web/20200716124958/https://www.intel.com/content/www/us/en/processors/xeon/scalable/xeon-scalable-spec-update.html>
2. Erratum BDF102 in Intel Xeon E5-2600 v4 Processor Product Family Specification Update: <http://web.archive.org/web/20191125200531/https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v4-spec-update.pdf>
3. The errata in Intel Resource Director Technology (Intel RDT) on 2nd Generation Intel Xeon Scalable Processors Reference Manual: <https://software.intel.com/content/www/us/en/develop/articles/intel-resource-director-technology-rdt-reference-manual.html>

for further information.

## TSX ASYNC ABORT (TAA) MITIGATION

### 24.1 Overview

TSX Async Abort (TAA) is a side channel attack on internal buffers in some Intel processors similar to Microarchitectural Data Sampling (MDS). In this case certain loads may speculatively pass invalid data to dependent operations when an asynchronous abort condition is pending in a Transactional Synchronization Extensions (TSX) transaction. This includes loads with no fault or assist condition. Such loads may speculatively expose stale data from the same uarch data structures as in MDS, with same scope of exposure i.e. same-thread and cross-thread. This issue affects all current processors that support TSX.

### 24.2 Mitigation strategy

- a) TSX disable - one of the mitigations is to disable TSX. A new MSR IA32\_TSX\_CTRL will be available in future and current processors after microcode update which can be used to disable TSX. In addition, it controls the enumeration of the TSX feature bits (RTM and HLE) in CPUID.
- b) Clear CPU buffers - similar to MDS, clearing the CPU buffers mitigates this vulnerability. More details on this approach can be found in [Documentation/admin-guide/hw-vuln/mds.rst](#).

### 24.3 Kernel internal mitigation modes

off	Mitigation is disabled. Either the CPU is not affected or <code>tsx_async_abort=off</code> is supplied on the kernel command line.
tsx disabled	Mitigation is enabled. TSX feature is disabled by default at bootup on processors that support TSX control.
verw	Mitigation is enabled. CPU is affected and MD_CLEAR is advertised in CPUID.
ucode needed	Mitigation is enabled. CPU is affected and MD_CLEAR is not advertised in CPUID. That is mainly for virtualization scenarios where the host has the updated microcode but the hypervisor does not expose MD_CLEAR in CPUID. It's a best effort approach without guarantee.

If the CPU is affected and the “tsx\_async\_abort” kernel command line parameter is not provided then the kernel selects an appropriate mitigation depending on the status of RTM and MD\_CLEAR CPUID bits.

Below tables indicate the impact of tsx=on|off|auto cmdline options on state of TAA mitigation, VERW behavior and TSX feature for various combinations of MSR\_IA32\_ARCH\_CAPABILITIES bits.

1. “tsx=off”

MSR_IA32_ARCH_CAPABILITIES bits			Result with cmdline tsx=off			
TAA_NO	MDS_NO	TSX_CTRL_MSR	TSX state after bootup	VERW can clear CPU buffers	TAA mitigation tsx_async_abort=off	TAA mitigation tsx_async_abort=full
0	0	0	HW default	Yes	Same as MDS	Same as MDS
0	0	1	Invalid case	Invalid case	Invalid case	Invalid case
0	1	0	HW default	No	Need ucode update	Need ucode update
0	1	1	Disabled	Yes	TSX disabled	TSX disabled
1	X	1	Disabled	X	None needed	None needed

2. “tsx=on”

MSR_IA32_ARCH_CAPABILITIES bits			Result with cmdline tsx=on			
TAA_NO	MDS_NO	TSX_CTRL_MSR	TSX state after bootup	VERW can clear CPU buffers	TAA mitigation tsx_async_abort=off	TAA mitigation tsx_async_abort=full
0	0	0	HW default	Yes	Same as MDS	Same as MDS
0	0	1	Invalid case	Invalid case	Invalid case	Invalid case
0	1	0	HW default	No	Need ucode update	Need ucode update
0	1	1	Enabled	Yes	None	Same as MDS
1	X	1	Enabled	X	None needed	None needed

3. “tsx=auto”

MSR_IA32_ARCH_CAPABILITIES bits			Result with cmdline tsx=auto			
TAA_NO	MDS_NO	TSX_CTRL_MSR	TSX state after bootup	VERW can clear CPU buffers	TAA mitigation tsx_async_abort=off	TAA mitigation tsx_async_abort=full
0	0	0	HW default	Yes	Same as MDS	Same as MDS
0	0	1	Invalid case	Invalid case	Invalid case	Invalid case
0	1	0	HW default	No	Need ucode update	Need ucode update
0	1	1	Disabled	Yes	TSX disabled	TSX disabled
1	X	1	Enabled	X	None needed	None needed

In the tables, TSX\_CTRL\_MSR is a new bit in MSR\_IA32\_ARCH\_CAPABILITIES that indicates whether MSR\_IA32\_TSX\_CTRL is supported.

There are two control bits in IA32\_TSX\_CTRL MSR:

**Bit 0: When set it disables the Restricted Transactional Memory (RTM)** sub-feature of TSX (will force all transactions to abort on the XBEGIN instruction).

**Bit 1: When set it disables the enumeration of the RTM and HLE feature** (i.e. it will make CPUID(EAX=7).EBX{bit4} and CPUID(EAX=7).EBX{bit11} read as 0).



## BUS LOCK DETECTION AND HANDLING

**Copyright** © 2021 Intel Corporation

**Authors**

- Fenghua Yu <[fenghua.yu@intel.com](mailto:fenghua.yu@intel.com)>
- Tony Luck <[tony.luck@intel.com](mailto:tony.luck@intel.com)>

### 25.1 Problem

A split lock is any atomic operation whose operand crosses two cache lines. Since the operand spans two cache lines and the operation must be atomic, the system locks the bus while the CPU accesses the two cache lines.

A bus lock is acquired through either split locked access to writeback (WB) memory or any locked access to non-WB memory. This is typically thousands of cycles slower than an atomic operation within a cache line. It also disrupts performance on other cores and brings the whole system to its knees.

### 25.2 Detection

Intel processors may support either or both of the following hardware mechanisms to detect split locks and bus locks.

#### 25.2.1 #AC exception for split lock detection

Beginning with the Tremont Atom CPU split lock operations may raise an Alignment Check (#AC) exception when a split lock operation is attempted.

### 25.2.2 #DB exception for bus lock detection

Some CPUs have the ability to notify the kernel by an #DB trap after a user instruction acquires a bus lock and is executed. This allows the kernel to terminate the application or to enforce throttling.

## 25.3 Software handling

The kernel #AC and #DB handlers handle bus lock based on the kernel parameter “split\_lock\_detect”. Here is a summary of different options:

split_lock_detect=	#AC for split lock	#DB for bus lock
off	Do nothing	Do nothing
warn (default)	Kernel OOPs Warn once per task and disable future checking When both features are supported, warn in #AC	Warn once per task and and continues to run.
fatal	Kernel OOPs Send SIGBUS to user When both features are supported, fatal in #AC	Send SIGBUS to user.
ratelimit:N (0 < N <= 1000)	Do nothing	Limit bus lock rate to N bus locks per second system wide and warn on bus locks.

## 25.4 Usages

Detecting and handling bus lock may find usages in various areas:

It is critical for real time system designers who build consolidated real time systems. These systems run hard real time code on some cores and run “untrusted” user processes on other cores. The hard real time cannot afford to have any bus lock from the untrusted processes to hurt real time performance. To date the designers have been unable to deploy these solutions as they have no way to prevent the “untrusted” user code from generating split lock and bus lock to block the hard real time code to access memory during bus locking.

It’s also useful for general computing to prevent guests or user applications from slowing down the overall system by executing instructions with bus lock.

## 25.5 Guidance

### 25.5.1 off

Disable checking for split lock and bus lock. This option can be useful if there are legacy applications that trigger these events at a low rate so that mitigation is not needed.



### 25.5.2 warn

A warning is emitted when a bus lock is detected which allows to identify the offending application. This is the default behavior.

### 25.5.3 fatal

In this case, the bus lock is not tolerated and the process is killed.

### 25.5.4 ratelimit

A system wide bus lock rate limit  $N$  is specified where  $0 < N \leq 1000$ . This allows a bus lock rate up to  $N$  bus locks per second. When the bus lock rate is exceeded then any task which is caught via the buslock `#DB` exception is throttled by enforced sleeps until the rate goes under the limit again.

This is an effective mitigation in cases where a minimal impact can be tolerated, but an eventual Denial of Service attack has to be prevented. It allows to identify the offending processes and analyze whether they are malicious or just badly written.

Selecting a rate limit of 1000 allows the bus to be locked for up to about seven million cycles each second (assuming 7000 cycles for each bus lock). On a 2 GHz processor that would be about 0.35% system slowdown.



## USB LEGACY SUPPORT

**Author** Vojtech Pavlik <[vojtech@suse.cz](mailto:vojtech@suse.cz)>, January 2004

Also known as “USB Keyboard” or “USB Mouse support” in the BIOS Setup is a feature that allows one to use the USB mouse and keyboard as if they were their classic PS/2 counterparts. This means one can use an USB keyboard to type in LILO for example.

It has several drawbacks, though:

- 1) On some machines, the emulated PS/2 mouse takes over even when no USB mouse is present and a real PS/2 mouse is present. In that case the extra features (wheel, extra buttons, touchpad mode) of the real PS/2 mouse may not be available.
- 2) If CONFIG\_HIGHMEM64G is enabled, the PS/2 mouse emulation can cause system crashes, because the SMM BIOS is not expecting to be in PAE mode. The Intel E7505 is a typical machine where this happens.
- 3) If AMD64 64-bit mode is enabled, again system crashes often happen, because the SMM BIOS isn't expecting the CPU to be in 64-bit mode. The BIOS manufacturers only test with Windows, and Windows doesn't do 64-bit yet.

Solutions:

**Problem 1)** can be solved by loading the USB drivers prior to loading the PS/2 mouse driver. Since the PS/2 mouse driver is in 2.6 compiled into the kernel unconditionally, this means the USB drivers need to be compiled-in, too.

**Problem 2)** can currently only be solved by either disabling HIGHMEM64G in the kernel config or USB Legacy support in the BIOS. A BIOS update could help, but so far no such update exists.

**Problem 3)** is usually fixed by a BIOS update. Check the board manufacturers web site. If an update is not available, disable USB Legacy support in the BIOS. If this alone doesn't help, try also adding `idle=poll` on the kernel command line. The BIOS may be entering the SMM on the HLT instruction as well.



## **I386 SUPPORT**

### **27.1 IO-APIC**

**Author** Ingo Molnar <[mingo@kernel.org](mailto:mingo@kernel.org)>

Most (all) Intel-MP compliant SMP boards have the so-called ‘IO-APIC’, which is an enhanced interrupt controller. It enables us to route hardware interrupts to multiple CPUs, or to CPU groups. Without an IO-APIC, interrupts from hardware will be delivered only to the CPU which boots the operating system (usually CPU#0).

Linux supports all variants of compliant SMP boards, including ones with multiple IO-APICs. Multiple IO-APICs are used in high-end servers to distribute IRQ load further.

There are (a few) known breakages in certain older boards, such bugs are usually worked around by the kernel. If your MP-compliant SMP board does not boot Linux, then consult the linux-smp mailing list archives first.

If your box boots fine with enabled IO-APIC IRQs, then your /proc/interrupts will look like this one:

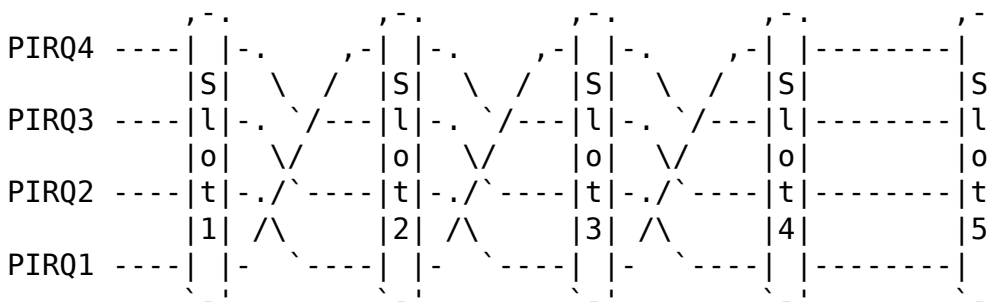
```
hell:~> cat /proc/interrupts
          CPU0
 0:   1360293   IO-APIC-edge   timer
 1:         4   IO-APIC-edge   keyboard
 2:         0           XT-PIC   cascade
13:         1           XT-PIC   fpu
14:       1448   IO-APIC-edge   ide0
16:       28232 IO-APIC-level   Intel EtherExpress Pro 10/100 Ethernet
17:       51304 IO-APIC-level   eth0
NMI:         0
ERR:         0
hell:~>
```

Some interrupts are still listed as ‘XT PIC’, but this is not a problem; none of those IRQ sources is performance-critical.

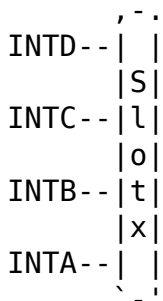
In the unlikely case that your board does not create a working mp-table, you can use the `pirq=` boot parameter to ‘hand-construct’ IRQ entries. This is non-trivial though and cannot be automated. One sample /etc/lilo.conf entry:

```
append="pirq=15,11,10"
```

The actual numbers depend on your system, on your PCI cards and on their PCI slot position. Usually PCI slots are 'daisy chained' before they are connected to the PCI chipset IRQ routing facility (the incoming PIRQ1-4 lines):



Every PCI card emits a PCI IRQ, which can be INTA, INTB, INTC or INTD:



These INTA-D PCI IRQs are always 'local to the card', their real meaning depends on which slot they are in. If you look at the daisy chaining diagram, a card in slot4, issuing INTA IRQ, it will end up as a signal on PIRQ4 of the PCI chipset. Most cards issue INTA, this creates optimal distribution between the PIRQ lines. (distributing IRQ sources properly is not a necessity, PCI IRQs can be shared at will, but it's a good for performance to have non shared interrupts). Slot5 should be used for videocards, they do not use interrupts normally, thus they are not daisy chained either.

so if you have your SCSI card (IRQ11) in Slot1, Tulip card (IRQ9) in Slot2, then you'll have to specify this `pirq=` line:

```
append="pirq=11,9"
```

the following script tries to figure out such a default `pirq=` line from your PCI configuration:

```
echo -n pirq=; echo `scanpci | grep T_L | cut -c56-` | sed 's/ /,/g'
```

note that this script won't work if you have skipped a few slots or if your board does not do default daisy-chaining. (or the IO-APIC has the PIRQ pins connected in some strange way). E.g. if in the above case you have your SCSI card (IRQ11) in Slot3, and have Slot1 empty:

```
append="pirq=0,9,11"
```

[value '0' is a generic 'placeholder', reserved for empty (or non-IRO emitting) slots.]

Generally, it's always possible to find out the correct `pirq=` settings, just permute all IRQ numbers properly ... it will take some time though. An 'incorrect' `pirq` line will cause the booting process to hang, or a device won't function properly (e.g. if it's inserted as a module).

If you have 2 PCI buses, then you can use up to 8 pirq values, although such boards tend to have a good configuration.

Be prepared that it might happen that you need some strange pirq line:

```
append="pirq=0,0,0,0,0,0,9,11"
```

Use smart trial-and-error techniques to find out the correct pirq line ...

Good luck and mail to [linux-smp@vger.kernel.org](mailto:linux-smp@vger.kernel.org) or [linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org) if you have any problems that are not covered by this document.





## X86\_64 SUPPORT

### 28.1 AMD64 Specific Boot Options

There are many others (usually documented in driver documentation), but only the AMD64 specific ones are listed here.

#### 28.1.1 Machine check

Please see *Configurable sysfs parameters for the x86-64 machine check code* for sysfs runtime tunables.

**mce=off** Disable machine check

**mce=no\_cmci** Disable CMCI(Corrected Machine Check Interrupt) that Intel processor supports. Usually this disablement is not recommended, but it might be handy if your hardware is misbehaving. Note that you'll get more problems without CMCI than with due to the shared banks, i.e. you might get duplicated error logs.

**mce=dont\_log\_ce** Don't make logs for corrected errors. All events reported as corrected are silently cleared by OS. This option will be useful if you have no interest in any of corrected errors.

**mce=ignore\_ce** Disable features for corrected errors, e.g. polling timer and CMCI. All events reported as corrected are not cleared by OS and remained in its error banks. Usually this disablement is not recommended, however if there is an agent checking/clearing corrected errors (e.g. BIOS or hardware monitoring applications), conflicting with OS's error handling, and you cannot deactivate the agent, then this option will be a help.

**mce=no\_lmce** Do not opt-in to Local MCE delivery. Use legacy method to broadcast MCEs.

**mce=bootlog** Enable logging of machine checks left over from booting. Disabled by default on AMD Fam10h and older because some BIOS leave bogus ones. If your BIOS doesn't do that it's a good idea to enable though to make sure you log even machine check events that result in a reboot. On Intel systems it is enabled by default.

**mce=nobootlog** Disable boot machine check logging.

**mce=monarchtimeout (number)** monarchtimeout: Sets the time in us to wait for other CPUs on machine checks. 0 to disable.

**mce=bios\_cmci\_threshold** Don't overwrite the bios-set CMCI threshold. This boot option prevents Linux from overwriting the CMCI threshold set by the bios. Without this option, Linux always sets the CMCI threshold to 1. Enabling this may make memory predictive failure analysis less effective if the bios sets thresholds for memory errors since we will not see details for all errors.

**mce=recovery** Force-enable recoverable machine check code paths

**nomce (for compatibility with i386)** same as mce=off

Everything else is in sysfs now.

### 28.1.2 APICs

**apic** Use IO-APIC. Default

**noapic** Don't use the IO-APIC.

**disableapic** Don't use the local APIC

**noapic** Don't use the local APIC (alias for i386 compatibility)

**pirq=...** See *IO-APIC*

**noapictimer** Don't set up the APIC timer

**no\_timer\_check** Don't check the IO-APIC timer. This can work around problems with incorrect timer initialization on some boards.

**apicpmtimer** Do APIC timer calibration using the pmtimer. Implies apicmaintimer. Useful when your PIT timer is totally broken.

### 28.1.3 Timing

**notsc** Deprecated, use tsc=unstable instead.

**nohpet** Don't use the HPET timer.

### 28.1.4 Idle loop

**idle=poll** Don't do power saving in the idle loop using HLT, but poll for rescheduling event. This will make the CPUs eat a lot more power, but may be useful to get slightly better performance in multiprocessor benchmarks. It also makes some profiling using performance counters more accurate. Please note that on systems with MONITOR/MWAIT support (like Intel EM64T CPUs) this option has no performance advantage over the normal idle loop. It may also interact badly with hyperthreading.

### 28.1.5 Rebooting

**reboot=b[ios] | t[riple] | k[bd] | a[cpil] | e[fi] | p[ci] [, [w]arm | [c]old]**

**bios** Use the CPU reboot vector for warm reset

**warm** Don't set the cold reboot flag

**cold** Set the cold reboot flag

**triple** Force a triple fault (init)

**kbd** Use the keyboard controller. cold reset (default)

**acpi** Use the ACPI RESET\_REG in the FADT. If ACPI is not configured or the ACPI reset does not work, the reboot path attempts the reset using the keyboard controller.

**efi** Use efi reset\_system runtime service. If EFI is not configured or the EFI reset does not work, the reboot path attempts the reset using the keyboard controller.

**pci** Use a write to the PCI config space register 0xcf9 to trigger reboot.

Using warm reset will be much faster especially on big memory systems because the BIOS will not go through the memory check. Disadvantage is that not all hardware will be completely reinitialized on reboot so there may be boot problems on some systems.

**reboot=force** Don't stop other CPUs on reboot. This can make reboot more reliable in some cases.

**reboot=default** There are some built-in platform specific "quirks" - you may see: "reboot: <name> series board detected. Selecting <type> for reboots." In the case where you think the quirk is in error (e.g. you have newer BIOS, or newer board) using this option will ignore the built-in quirk table, and use the generic default reboot actions.

### 28.1.6 NUMA

**numa=off** Only set up a single NUMA node spanning all memory.

**numa=noacpi** Don't parse the SRAT table for NUMA setup

**numa=nohmat** Don't parse the HMAT table for NUMA setup, or soft-reserved memory partitioning.

**numa=fake=<size>[MG]** If given as a memory unit, fills all system RAM with nodes of size interleaved over physical nodes.

**numa=fake=<N>** If given as an integer, fills all system RAM with N fake nodes interleaved over physical nodes.

**numa=fake=<N>U** If given as an integer followed by 'U', it will divide each physical node into N emulated nodes.

### 28.1.7 ACPI

**acpi=off** Don't enable ACPI

**acpi=ht** Use ACPI boot table parsing, but don't enable ACPI interpreter

**acpi=force** Force ACPI on (currently not needed)

**acpi=strict** Disable out of spec ACPI workarounds.

**acpi\_sci={edge,level,high,low}** Set up ACPI SCI interrupt.

**acpi=noirq** Don't route interrupts

**acpi=nocmccff** Disable firmware first mode for corrected errors. This disables parsing the HEST CMC error source to check if firmware has set the FF flag. This may result in duplicate corrected error reports.

### 28.1.8 PCI

**pci=off** Don't use PCI

**pci=conf1** Use conf1 access.

**pci=conf2** Use conf2 access.

**pci=rom** Assign ROMs.

**pci=assign-busses** Assign busses

**pci=irqmask=MASK** Set PCI interrupt mask to MASK

**pci=lastbus=NUMBER** Scan up to NUMBER busses, no matter what the mptable says.

**pci=noacpi** Don't use ACPI to set up PCI interrupt routing.

### 28.1.9 IOMMU (input/output memory management unit)

Multiple x86-64 PCI-DMA mapping implementations exist, for example:

1. <kernel/dma/direct.c>: use no hardware/software IOMMU at all (e.g. because you have < 3 GB memory). Kernel boot message: "PCI-DMA: Disabling IOMMU"
2. <arch/x86/kernel/amd\_gart\_64.c>: AMD GART based hardware IOMMU. Kernel boot message: "PCI-DMA: using GART IOMMU"
3. <arch/x86\_64/kernel/pci-swiotlb.c> : Software IOMMU implementation. Used e.g. if there is no hardware IOMMU in the system and it is need because you have >3GB memory or told the kernel to us it (iommu=soft)) Kernel boot message: "PCI-DMA: Using software bounce buffering for IO (SWIOTLB)"

```
iommu=[<size>][,noagp][,off][,force][,noforce]
[,memaper[=<order>]][,merge][,fullflush][,nomerge]
[,noaperture]
```

General iommu options:

**off** Don't initialize and use any kind of IOMMU.

**noforce** Don't force hardware IOMMU usage when it is not needed. (default).

**force** Force the use of the hardware IOMMU even when it is not actually needed (e.g. because < 3 GB memory).

**soft** Use software bounce buffering (SWIOTLB) (default for Intel machines). This can be used to prevent the usage of an available hardware IOMMU.

iommu options only relevant to the AMD GART hardware IOMMU:

**<size>** Set the size of the remapping area in bytes.

**allowed** Overwrite iommu off workarounds for specific chipsets.

**fullflush** Flush IOMMU on each allocation (default).

**nofullflush** Don't use IOMMU fullflush.

**memaper[=<order>]** Allocate an own aperture over RAM with size 32MB<order. (default: order=1, i.e. 64MB)

**merge** Do scatter-gather (SG) merging. Implies "force" (experimental).

**nomerge** Don't do scatter-gather (SG) merging.

**noaperture** Ask the IOMMU not to touch the aperture for AGP.

**noagp** Don't initialize the AGP driver and use full aperture.

**panic** Always panic when IOMMU overflows.

iommu options only relevant to the software bounce buffering (SWIOTLB) IOMMU implementation:

**swiotlb=<pages>[,force]**

**<pages>** Prereserve that many 128K pages for the software IO bounce buffering.

**force** Force all IO through the software TLB.

### 28.1.10 Miscellaneous

**nogbpages** Do not use GB pages for kernel direct mappings.

**gbpages** Use GB pages for kernel direct mappings.

### 28.1.11 AMD SEV (Secure Encrypted Virtualization)

Options relating to AMD SEV, specified via the following format:

```
sev=option1[,option2]
```

The available options are:

**debug** Enable debug messages.

## 28.2 General note on [U]EFI x86\_64 support

The nomenclature EFI and UEFI are used interchangeably in this document.

Although the tools below are not needed for building the kernel, the needed bootloader support and associated tools for x86\_64 platforms with EFI firmware and specifications are listed below.

1. UEFI specification: <http://www.uefi.org>
2. Booting Linux kernel on UEFI x86\_64 platform requires bootloader support. Elilo with x86\_64 support can be used.
3. x86\_64 platform with EFI/UEFI firmware.

### 28.2.1 Mechanics

- Build the kernel with the following configuration:

```
CONFIG_FB_EFI=y
CONFIG_FRAMEBUFFER_CONSOLE=y
```

If EFI runtime services are expected, the following configuration should be selected:

```
CONFIG_EFI=y
CONFIG_EFI_VARS=y or m          # optional
```

- Create a VFAT partition on the disk
- Copy the following to the VFAT partition:
  - elilo bootloader with x86\_64 support, elilo configuration file, kernel image built in first step and corresponding initrd. Instructions on building elilo and its dependencies can be found in the elilo sourceforge project.
- Boot to EFI shell and invoke elilo choosing the kernel image built in first step.
- If some or all EFI runtime services don't work, you can try following kernel command line parameters to turn off some or all EFI runtime services.
  - noefi** turn off all EFI runtime services
  - reboot\_type=k** turn off EFI reboot runtime service
- If the EFI memory map has additional entries not in the E820 map, you can include those entries in the kernels memory map of available physical RAM by using the following kernel command line parameter.
  - add\_efi\_memmap** include EFI memory map of available physical RAM

## 28.3 Memory Management

### 28.3.1 Complete virtual memory map with 4-level page tables

**Note:**

- Negative addresses such as “-23 TB” are absolute addresses in bytes, counted down from the top of the 64-bit address space. It’s easier to understand the layout when seen both in absolute addresses and in distance-from-top notation.  
For example 0xffffe90000000000 == -23 TB, it’s 23 TB lower than the top of the 64-bit address space (ffffffffffffffff).  
Note that as we get closer to the top of the address space, the notation changes from TB to GB and then MB/KB.
- “16M TB” might look weird at first sight, but it’s an easier way to visualize size notation than “16 EB”, which few will recognize at first sight as 16 exabytes. It also shows it nicely how incredibly large 64-bit address space is.

Start addr ↪description	Offset	End addr	Size	VM area
0000000000000000 ↪virtual memory, different per mm	0	00007fffffffffffff	128 TB	user-space
↪				
0000800000000000 ↪64 bits wide hole of non-canonical ↪memory addresses up to the -128 TB ↪offset of kernel mappings.	+128 TB	ffff7fffffffffffff	~16M TB	... huge, almost virtual starting
↪				
↪virtual memory, shared between all processes:				Kernel-space
↪				
ffff800000000000 ↪also reserved for hypervisor	-128 TB	ffff87ffffffffffff	8 TB	... guard hole,
ffff880000000000	-120 TB	ffff887ffffffffffff	0.5 TB	LDT remap for PTI
ffff888000000000 ↪of all physical memory (page_offset_base)	-119.5 TB	ffffc87ffffffffffff	64 TB	direct mapping
ffffc88000000000	-55.5 TB	ffffc87ffffffffffff	0.5 TB	... unused hole
ffffc90000000000 ↪space (vmalloc_base)	-55 TB	ffffe87ffffffffffff	32 TB	vmalloc/ioremap

fffffe90000000000		-23	TB		fffffe9fffffffffff		1 TB		... unused hole	
fffffea0000000000		-22	TB		fffffeafffffffffffff		1 TB		virtual memory	
→map (vmemmap_base)										
fffffeb0000000000		-21	TB		fffffebfffffffffff		1 TB		... unused hole	
fffffec0000000000		-20	TB		ffffffbfffffffffff		16 TB		KASAN shadow	
→memory										
									Identical layout	
→										
→to the 56-bit one from here on:										
→										
ffffffc0000000000		-4	TB		ffffffdfffffffffff		2 TB		... unused hole	
→KASLR										
ffffffe0000000000		-2	TB		ffffffe7fffffffffff		0.5 TB		cpu_entry_area	
→mapping										
ffffffe8000000000		-1.5	TB		ffffffefffffffffffff		0.5 TB		... unused hole	
fffffff0000000000		-1	TB		fffffff7fffffffffff		0.5 TB		%esp fixup stacks	
fffffff8000000000		-512	GB		fffffffeefffffffffff		444 GB		... unused hole	
fffffffef00000000		-68	GB		ffffffefffffffffffff		64 GB		EFI region	
→mapping space										
fffffffff00000000		-4	GB		fffffffff7fffffffff		2 GB		... unused hole	
fffffffff80000000		-2	GB		fffffffff9fffffffff		512 MB		kernel text	
→mapping, mapped to physical address 0										
fffffffff80000000		-2048	MB							
ffffffffffa0000000		-1536	MB		ffffffffffefffffff		1520 MB		module mapping	
→space										
ffffffffffff000000		-16	MB							
FIXADDR_START		~-11	MB		ffffffffffff5fffff		~0.5 MB		kernel-internal	
→fixmap range, variable size and offset										
ffffffffffff600000		-10	MB		ffffffffffff600fff		4 kB		legacy vsyscall	
→ABI										
ffffffffffffe00000		-2	MB		ffffffffffffffffff		2 MB		... unused hole	
→										

### 28.3.2 Complete virtual memory map with 5-level page tables

#### Note:

- With 56-bit addresses, user-space memory gets expanded by a factor of 512x, from 0.125 PB to 64 PB. All kernel mappings shift down to the -64 PB starting offset and many of the regions expand to support the much larger physical memory supported.



Start addr →description	Offset	End addr	Size	VM area
0000000000000000 →virtual memory, different per mm	0	00ffffffffffffff	64 PB	user-space
0100000000000000 →almost 64 bits wide hole of non-canonical →memory addresses up to the -64 PB →offset of kernel mappings.	+64 PB	feffffffffffffff	~16K PB	... huge, still virtual starting
→				Kernel-space
→virtual memory, shared between all processes:				
ff00000000000000 →also reserved for hypervisor	-64 PB	ff0fffffffffffff	4 PB	... guard hole,
ff10000000000000	-60 PB	ff10ffffffffffff	0.25 PB	LDT remap for PTI
ff11000000000000 →of all physical memory (page_offset_base)	-59.75 PB	ff90ffffffffffff	32 PB	direct mapping
ff91000000000000	-27.75 PB	ff9fffffffffffff	3.75 PB	... unused hole
ffa0000000000000 →space (vmalloc_base)	-24 PB	ffd1ffffffffffff	12.5 PB	vmalloc/ioremap
ffd2000000000000	-11.5 PB	ffd3ffffffffffff	0.5 PB	... unused hole
ffd4000000000000 →map (vmemmap_base)	-11 PB	ffd5ffffffffffff	0.5 PB	virtual memory
ffd6000000000000	-10.5 PB	ffdeffffffffffff	2.25 PB	... unused hole
ffdf000000000000 →memory	-8.25 PB	fffffbffffffffff	~8 PB	KASAN shadow
→				Identical layout
→to the 47-bit one from here on:				
fffffc0000000000 →KASLR	-4 TB	fffffdffffffffff	2 TB	... unused hole vaddr_end for
fffffe0000000000 →mapping	-2 TB	fffffe7fffffffff	0.5 TB	cpu_entry_area

fffffe8000000000		-1.5	TB		fffffeffffffffffff		0.5	TB		... unused hole
ffffff0000000000		-1	TB		ffffff7fffffffffff		0.5	TB		%esp fixup stacks
ffffff8000000000		-512	GB		ffffffeefeffffff		444	GB		... unused hole
ffffffef00000000		-68	GB		fffffffeffffffff		64	GB		EFI region
→ mapping space										
ffffffff00000000		-4	GB		fffffff7fffffff		2	GB		... unused hole
fffffff800000000		-2	GB		fffffff9fffffff		512	MB		kernel text
→ mapping, mapped to physical address 0										
fffffff800000000		-2048	MB							
fffffffa00000000		-1536	MB		fffffffeffff		1520	MB		module mapping
→ space										
fffffffffff000000		-16	MB							
FIXADDR_START		~-11	MB		fffffffff5ffff		~0.5	MB		kernel-internal
→ fixmap range, variable size and offset										
fffffffffff600000		-10	MB		fffffffff600fff		4	kB		legacy vsyscall
→ ABI										
fffffffffffef00000		-2	MB		ffffffffffffffff		2	MB		... unused hole

Architecture defines a 64-bit virtual address. Implementations can support less. Currently supported are 48- and 57-bit virtual addresses. Bits 63 through to the most-significant implemented bit are sign extended. This causes hole between user space and kernel addresses if you interpret them as unsigned.

The direct mapping covers all memory in the system up to the highest memory address (this means in some cases it can also include PCI memory holes).

We map EFI runtime services in the 'efi\_pgd' PGD in a 64Gb large virtual memory window (this size is arbitrary, it can be raised later if needed). The mappings are not part of any other kernel PGD and are only available during EFI runtime calls.

Note that if CONFIG\_RANDOMIZE\_MEMORY is enabled, the direct mapping of all physical memory, vmalloc/ioremap space and virtual memory map are randomized. Their order is preserved but their base will be offset early at boot time.

Be very careful vs. KASLR when changing anything here. The KASLR address range must not overlap with anything except the KASAN shadow area, which is correct as KASAN disables KASLR.

For both 4- and 5-level layouts, the STACKLEAK\_POISON value in the last 2MB hole: ffffffff4111

## 28.4 5-level paging

### 28.4.1 Overview

Original x86-64 was limited by 4-level paging to 256 TiB of virtual address space and 64 TiB of physical address space. We are already bumping into this limit: some vendors offer servers with 64 TiB of memory today.

To overcome the limitation upcoming hardware will introduce support for 5-level paging. It

is a straight-forward extension of the current page table structure adding one more layer of translation.

It bumps the limits to 128 PiB of virtual address space and 4 PiB of physical address space. This “ought to be enough for anybody” ©.

QEMU 2.9 and later support 5-level paging.

Virtual memory layout for 5-level paging is described in [Memory Management](#)

### 28.4.2 Enabling 5-level paging

`CONFIG_X86_5LEVEL=y` enables the feature.

Kernel with `CONFIG_X86_5LEVEL=y` still able to boot on 4-level hardware. In this case additional page table level - p4d - will be folded at runtime.

### 28.4.3 User-space and large virtual address space

On x86, 5-level paging enables 56-bit userspace virtual address space. Not all user space is ready to handle wide addresses. It's known that at least some JIT compilers use higher bits in pointers to encode their information. It collides with valid pointers with 5-level paging and leads to crashes.

To mitigate this, we are not going to allocate virtual address space above 47-bit by default.

But userspace can ask for allocation from full address space by specifying hint address (with or without `MAP_FIXED`) above 47-bits.

If hint address set above 47-bit, but `MAP_FIXED` is not specified, we try to look for unmapped area by specified address. If it's already occupied, we look for unmapped area in *full* address space, rather than from 47-bit window.

A high hint address would only affect the allocation in question, but not any future `mmap()`s.

Specifying high hint address on older kernel or on machine without 5-level paging support is safe. The hint will be ignored and kernel will fall back to allocation from 47-bit address space.

This approach helps to easily make application's memory allocator aware about large address space without manually tracking allocated virtual address space.

One important case we need to handle here is interaction with MPX. MPX (without MAWA extension) cannot handle addresses above 47-bit, so we need to make sure that MPX cannot be enabled we already have VMA above the boundary and forbid creating such VMAs once MPX is enabled.

## 28.5 Fake NUMA For CPUSets

**Author** David Rientjes <[rientjes@cs.washington.edu](mailto:rientjes@cs.washington.edu)>

Using numa=fake and CPUSets for Resource Management

This document describes how the numa=fake x86\_64 command-line option can be used in conjunction with cpusets for coarse memory management. Using this feature, you can create fake NUMA nodes that represent contiguous chunks of memory and assign them to cpusets and their attached tasks. This is a way of limiting the amount of system memory that are available to a certain class of tasks.

For more information on the features of cpusets, see [Documentation/admin-guide/cgroup-v1/cpusets.rst](#). There are a number of different configurations you can use for your needs. For more information on the numa=fake command line option and its various ways of configuring fake nodes, see [AMD64 Specific Boot Options](#).

For the purposes of this introduction, we'll assume a very primitive NUMA emulation setup of "numa=fake=4\*512,". This will split our system memory into four equal chunks of 512M each that we can now use to assign to cpusets. As you become more familiar with using this combination for resource control, you'll determine a better setup to minimize the number of nodes you have to deal with.

A machine may be split as follows with "numa=fake=4\*512," as reported by dmesg:

```
Faking node 0 at 0000000000000000-0000000020000000 (512MB)
Faking node 1 at 0000000020000000-0000000040000000 (512MB)
Faking node 2 at 0000000040000000-0000000060000000 (512MB)
Faking node 3 at 0000000060000000-0000000080000000 (512MB)
...
On node 0 totalpages: 130975
On node 1 totalpages: 131072
On node 2 totalpages: 131072
On node 3 totalpages: 131072
```

Now following the instructions for mounting the cpusets filesystem from [Documentation/admin-guide/cgroup-v1/cpusets.rst](#), you can assign fake nodes (i.e. contiguous memory address spaces) to individual cpusets:

```
[root@xroads /]# mkdir exampleset
[root@xroads /]# mount -t cuset none exampleset
[root@xroads /]# mkdir exampleset/ddset
[root@xroads /]# cd exampleset/ddset
[root@xroads /exampleset/ddset]# echo 0-1 > cpus
[root@xroads /exampleset/ddset]# echo 0-1 > mems
```

Now this cuset, 'ddset', will only allowed access to fake nodes 0 and 1 for memory allocations (1G).

You can now assign tasks to these cpusets to limit the memory resources available to them according to the fake nodes assigned as mems:

```
[root@xroads /exampleset/ddset]# echo $$ > tasks
[root@xroads /exampleset/ddset]# dd if=/dev/zero of=tmp bs=1024 count=1G
```

[1] 13425

Notice the difference between the system memory usage as reported by `/proc/meminfo` between the restricted cpuset case above and the unrestricted case (i.e. running the same ‘dd’ command without assigning it to a fake NUMA cpuset):

Name	Unrestricted	Restricted
MemTotal	3091900 kB	3091900 kB
MemFree	42113 kB	1513236 kB

This allows for coarse memory management for the tasks you assign to particular cpusets. Since cpusets can form a hierarchy, you can create some pretty interesting combinations of use-cases for various classes of tasks for your memory management needs.

## 28.6 Firmware support for CPU hotplug under Linux/x86-64

Linux/x86-64 supports CPU hotplug now. For various reasons Linux wants to know in advance of boot time the maximum number of CPUs that could be plugged into the system. ACPI 3.0 currently has no official way to supply this information from the firmware to the operating system.

In ACPI each CPU needs an LAPIC object in the MADT table (5.2.11.5 in the ACPI 3.0 specification). ACPI already has the concept of disabled LAPIC objects by setting the Enabled bit in the LAPIC object to zero.

For CPU hotplug Linux/x86-64 expects now that any possible future hotpluggable CPU is already available in the MADT. If the CPU is not available yet it should have its LAPIC Enabled bit set to 0. Linux will use the number of disabled LAPICs to compute the maximum number of future CPUs.

In the worst case the user can overwrite this choice using a command line option (`additional_cpus=...`), but it is recommended to supply the correct number (or a reasonable approximation of it, with erring towards more not less) in the MADT to avoid manual configuration.

## 28.7 Configurable sysfs parameters for the x86-64 machine check code

Machine checks report internal hardware error conditions detected by the CPU. Uncorrected errors typically cause a machine check (often with panic), corrected ones cause a machine check log entry.

Machine checks are organized in banks (normally associated with a hardware subsystem) and subevents in a bank. The exact meaning of the banks and subevent is CPU specific.

mcelog knows how to decode them.

When you see the “Machine check errors logged” message in the system log then mcelog should run to collect and decode machine check entries from `/dev/mcelog`. Normally mcelog should be run regularly from a cronjob.

Each CPU has a directory in `/sys/devices/system/machinecheck/machinecheckN` ( $N$  = CPU number).

The directory contains some configurable entries. See `Documentation/ABI/testing/sysfs-mce` for more details.

TBD document entries for AMD threshold interrupt configuration

For more details about the x86 machine check architecture see the Intel and AMD architecture manuals from their developer websites.

For more details about the architecture see <http://one.firstfloor.org/~andi/mce.pdf>

## 28.8 Using FS and GS segments in user space applications

The x86 architecture supports segmentation. Instructions which access memory can use segment register based addressing mode. The following notation is used to address a byte within a segment:

Segment-register:Byte-address

The segment base address is added to the Byte-address to compute the resulting virtual address which is accessed. This allows to access multiple instances of data with the identical Byte-address, i.e. the same code. The selection of a particular instance is purely based on the base-address in the segment register.

In 32-bit mode the CPU provides 6 segments, which also support segment limits. The limits can be used to enforce address space protections.

In 64-bit mode the CS/SS/DS/ES segments are ignored and the base address is always 0 to provide a full 64bit address space. The FS and GS segments are still functional in 64-bit mode.

### 28.8.1 Common FS and GS usage

The FS segment is commonly used to address Thread Local Storage (TLS). FS is usually managed by runtime code or a threading library. Variables declared with the `'__thread'` storage class specifier are instantiated per thread and the compiler emits the FS: address prefix for accesses to these variables. Each thread has its own FS base address so common code can be used without complex address offset calculations to access the per thread instances. Applications should not use FS for other purposes when they use runtimes or threading libraries which manage the per thread FS.

The GS segment has no common use and can be used freely by applications. GCC and Clang support GS based addressing via address space identifiers.

## 28.8.2 Reading and writing the FS/GS base address

There exist two mechanisms to read and write the FS/GS base address:

- the `arch_prctl()` system call
- the FSGSBASE instruction family

## 28.8.3 Accessing FS/GS base with `arch_prctl()`

The `arch_prctl(2)` based mechanism is available on all 64-bit CPUs and all kernel versions.

Reading the base:

```
arch_prctl(ARCH_GET_FS, &fsbase); arch_prctl(ARCH_GET_GS, &gsbase);
```

Writing the base:

```
arch_prctl(ARCH_SET_FS, fsbase); arch_prctl(ARCH_SET_GS, gsbase);
```

The `ARCH_SET_GS prctl` may be disabled depending on kernel configuration and security settings.

## 28.8.4 Accessing FS/GS base with the FSGSBASE instructions

With the Ivy Bridge CPU generation Intel introduced a new set of instructions to access the FS and GS base registers directly from user space. These instructions are also supported on AMD Family 17H CPUs. The following instructions are available:

RDFSBASE %reg	Read the FS base register
RDGSBASE %reg	Read the GS base register
WRFSBASE %reg	Write the FS base register
WRGSBASE %reg	Write the GS base register

The instructions avoid the overhead of the `arch_prctl()` syscall and allow more flexible usage of the FS/GS addressing modes in user space applications. This does not prevent conflicts between threading libraries and runtimes which utilize FS and applications which want to use it for their own purpose.

## FSGSBASE instructions enablement

The instructions are enumerated in CPUID leaf 7, bit 0 of EBX. If available `/proc/cpuinfo` shows 'fsgsbase' in the flag entry of the CPUs.

The availability of the instructions does not enable them automatically. The kernel has to enable them explicitly in CR4. The reason for this is that older kernels make assumptions about the values in the GS register and enforce them when GS base is set via `arch_prctl()`. Allowing user space to write arbitrary values to GS base would violate these assumptions and cause malfunction.

On kernels which do not enable FSGSBASE the execution of the FSGSBASE instructions will fault with a `#UD` exception.

The kernel provides reliable information about the enabled state in the ELF AUX vector. If the `HWCAP2_FSGSBASE` bit is set in the AUX vector, the kernel has FS-GSBASE instructions enabled and applications can use them. The following code example shows how this detection works:

```
#include <sys/auxv.h>
#include <elf.h>

/* Will be eventually in asm/hwcap.h */
#ifndef HWCAP2_FSGSBASE
#define HWCAP2_FSGSBASE      (1 << 1)
#endif

....

unsigned val = getauxval(AT_HWCAP2);

if (val & HWCAP2_FSGSBASE)
    printf("FSGSBASE enabled\n");
```

### FSGSBASE instructions compiler support

GCC version 4.6.4 and newer provide intrinsics for the FSGSBASE instructions. Clang 5 supports them as well.

<code>_readfsbase_u64()</code>	Read the FS base register
<code>_readgsbase_u64()</code>	Read the GS base register
<code>_writefsbase_u64()</code>	Write the FS base register
<code>_writegsbase_u64()</code>	Write the GS base register

To utilize these intrinsics `<immintrin.h>` must be included in the source code and the compiler option `-mfsgsbase` has to be added.

### 28.8.5 Compiler support for FS/GS based addressing

GCC version 6 and newer provide support for FS/GS based addressing via Named Address Spaces. GCC implements the following address space identifiers for x86:

<code>__seg_fs</code>	Variable is addressed relative to FS
<code>__seg_gs</code>	Variable is addressed relative to GS

The preprocessor symbols `__SEG_FS` and `__SEG_GS` are defined when these address spaces are supported. Code which implements fallback modes should check whether these symbols are defined. Usage example:

```
#ifdef __SEG_GS

long data0 = 0;
long data1 = 1;
```



```

long __seg_gs *ptr;

/* Check whether FSGSBASE is enabled by the kernel (HWCAP2_FSGSBASE) */
....

/* Set GS base to point to data0 */
_writegsbase_u64(&data0);

/* Access offset 0 of GS */
ptr = 0;
printf("data0 = %ld\n", *ptr);

/* Set GS base to point to data1 */
_writegsbase_u64(&data1);
/* ptr still addresses offset 0! */
printf("data1 = %ld\n", *ptr);

```

Clang does not provide the GCC address space identifiers, but it provides address spaces via an attribute based mechanism in Clang 2.6 and newer versions:

<code>__attribute__((address_space(256)))</code>	Variable is addressed relative to GS
<code>__attribute__((address_space(257)))</code>	Variable is addressed relative to FS

### 28.8.6 FS/GS based addressing with inline assembly

In case the compiler does not support address spaces, inline assembly can be used for FS/GS based addressing mode:

```

mov %fs:offset, %reg
mov %gs:offset, %reg

mov %reg, %fs:offset
mov %reg, %gs:offset

```

### In-Field Scan



## IN-FIELD SCAN

### 29.1 Introduction

In Field Scan (IFS) is a hardware feature to run circuit level tests on a CPU core to detect problems that are not caught by parity or ECC checks. Future CPUs will support more than one type of test which will show up with a new platform-device instance-id, for now only .0 is exposed.

### 29.2 IFS Image

Intel provides a firmware file containing the scan tests via [github](https://github.com/intel/TBD)<sup>1</sup>. Similar to microcode there is a separate file for each family-model-stepping.

### 29.3 IFS Image Loading

The driver loads the tests into memory reserved BIOS local to each CPU socket in a two step process using writes to MSRs to first load the SHA hashes for the test. Then the tests themselves. Status MSRs provide feedback on the success/failure of these steps. When a new test file is installed it can be loaded by writing to the driver reload file:

```
# echo 1 > /sys/devices/virtual/misc/intel_ifs_0/reload
```

Similar to microcode, the current version of the scan tests is stored in a fixed location: `/lib/firmware/intel/ifs.0/family-model-stepping.scan`

### 29.4 Running tests

Tests are run by the driver synchronizing execution of all threads on a core and then writing to the `ACTIVATE_SCAN` MSR on all threads. Instruction execution continues when:

- 1) All tests have completed.
- 2) Execution was interrupted.
- 3) A test detected a problem.

---

<sup>1</sup> <https://github.com/intel/TBD>

Note that ALL THREADS ON THE CORE ARE EFFECTIVELY OFFLINE FOR THE DURATION OF THE TEST. This can be up to 200 milliseconds. If the system is running latency sensitive applications that cannot tolerate an interruption of this magnitude, the system administrator must arrange to migrate those applications to other cores before running a core test. It may also be necessary to redirect interrupts to other CPUs.

In all cases reading the SCAN\_STATUS MSR provides details on what happened. The driver makes the value of this MSR visible to applications via the “details” file (see below). Interrupted tests may be restarted.

The IFS driver provides sysfs interfaces via `/sys/devices/virtual/misc/intel_ifs_0/` to control execution:

Test a specific core:

```
# echo <cpu#> > /sys/devices/virtual/misc/intel_ifs_0/run_test
```

when HT is enabled any of the sibling `cpu#` can be specified to test its corresponding physical core. Since the tests are per physical core, the result of testing any thread is same. All siblings must be online to run a core test. It is only necessary to test one thread.

For e.g. to test core corresponding to `cpu5`

```
# echo 5 > /sys/devices/virtual/misc/intel_ifs_0/run_test
```

Results of the last test is provided in `/sys`:

```
$ cat /sys/devices/virtual/misc/intel_ifs_0/status
pass
```

Status can be one of pass, fail, untested

Additional details of the last test is provided by the details file:

```
$ cat /sys/devices/virtual/misc/intel_ifs_0/details
0x8081
```

The details file reports the hex value of the SCAN\_STATUS MSR. Hardware defined error codes are documented in volume 4 of the Intel Software Developer’s Manual but the `error_code` field may contain one of the following driver defined software codes:

0xFD	Software timeout
0xFE	Partial completion

## 29.5 Driver design choices

1) The ACTIVATE\_SCAN MSR allows for running any consecutive subrange of available tests. But the driver always tries to run all tests and only uses the subrange feature to restart an interrupted test.

2) Hardware allows for some number of cores to be tested in parallel. The driver does not make use of this, it only tests one core at a time.

struct **ifs\_data**

attributes related to intel IFS driver

## Definition

```
struct ifs_data {
    int integrity_cap_bit;
    int loaded_version;
    bool loaded;
    bool loading_error;
    int valid_chunks;
    int status;
    u64 scan_details;
};
```

## Members

**integrity\_cap\_bit** MSR\_INTEGRITY\_CAPS bit enumerating this test

**loaded\_version** stores the currently loaded ifs image version.

**loaded** If a valid test binary has been loaded into the memory

**loading\_error** Error occurred on another CPU while loading image

**valid\_chunks** number of chunks which could be validated.

**status** it holds simple status pass/fail/untested

**scan\_details** opaque scan status code from h/w



## **SHARED VIRTUAL ADDRESSING (SVA) WITH ENQCMD**

### **30.1 Background**

Shared Virtual Addressing (SVA) allows the processor and device to use the same virtual addresses avoiding the need for software to translate virtual addresses to physical addresses. SVA is what PCIe calls Shared Virtual Memory (SVM).

In addition to the convenience of using application virtual addresses by the device, it also doesn't require pinning pages for DMA. PCIe Address Translation Services (ATS) along with Page Request Interface (PRI) allow devices to function much the same way as the CPU handling application page-faults. For more information please refer to the PCIe specification Chapter 10: ATS Specification.

Use of SVA requires IOMMU support in the platform. IOMMU is also required to support the PCIe features ATS and PRI. ATS allows devices to cache translations for virtual addresses. The IOMMU driver uses the `mmu_notifier()` support to keep the device TLB cache and the CPU cache in sync. When an ATS lookup fails for a virtual address, the device should use the PRI in order to request the virtual address to be paged into the CPU page tables. The device must use ATS again in order to fetch the translation before use.

### **30.2 Shared Hardware Workqueues**

Unlike Single Root I/O Virtualization (SR-IOV), Scalable IOV (SIOV) permits the use of Shared Work Queues (SWQ) by both applications and Virtual Machines (VM's). This allows better hardware utilization vs. hard partitioning resources that could result in under utilization. In order to allow the hardware to distinguish the context for which work is being executed in the hardware by SWQ interface, SIOV uses Process Address Space ID (PASID), which is a 20-bit number defined by the PCIe SIG.

PASID value is encoded in all transactions from the device. This allows the IOMMU to track I/O on a per-PASID granularity in addition to using the PCIe Resource Identifier (RID) which is the Bus/Device/Function.

## 30.3 ENQCMD

ENQCMD is a new instruction on Intel platforms that atomically submits a work descriptor to a device. The descriptor includes the operation to be performed, virtual addresses of all parameters, virtual address of a completion record, and the PASID (process address space ID) of the current process.

ENQCMD works with non-posted semantics and carries a status back if the command was accepted by hardware. This allows the submitter to know if the submission needs to be retried or other device specific mechanisms to implement fairness or ensure forward progress should be provided.

ENQCMD is the glue that ensures applications can directly submit commands to the hardware and also permits hardware to be aware of application context to perform I/O operations via use of PASID.

## 30.4 Process Address Space Tagging

A new thread-scoped MSR (IA32\_PASID) provides the connection between user processes and the rest of the hardware. When an application first accesses an SVA-capable device, this MSR is initialized with a newly allocated PASID. The driver for the device calls an IOMMU-specific API that sets up the routing for DMA and page-requests.

For example, the Intel Data Streaming Accelerator (DSA) uses `iommu_sva_bind_device()`, which will do the following:

- Allocate the PASID, and program the process page-table (`%cr3` register) in the PASID context entries.
- Register for `mmu_notifier()` to track any page-table invalidations to keep the device TLB in sync. For example, when a page-table entry is invalidated, the IOMMU propagates the invalidation to the device TLB. This will force any future access by the device to this virtual address to participate in ATS. If the IOMMU responds with proper response that a page is not present, the device would request the page to be paged in via the PCIe PRI protocol before performing I/O.

This MSR is managed with the XSAVE feature set as “supervisor state” to ensure the MSR is updated during context switch.

## 30.5 PASID Management

The kernel must allocate a PASID on behalf of each process which will use ENQCMD and program it into the new MSR to communicate the process identity to platform hardware. ENQCMD uses the PASID stored in this MSR to tag requests from this process. When a user submits a work descriptor to a device using the ENQCMD instruction, the PASID field in the descriptor is auto-filled with the value from `MSR_IA32_PASID`. Requests for DMA from the device are also tagged with the same PASID. The platform IOMMU uses the PASID in the transaction to perform address translation. The IOMMU APIs setup the corresponding PASID entry in IOMMU with the process address used by the CPU (e.g. `%cr3` register in x86).



The MSR must be configured on each logical CPU before any application thread can interact with a device. Threads that belong to the same process share the same page tables, thus the same MSR value.

## 30.6 PASID Life Cycle Management

PASID is initialized as `INVALID_IOASID` (-1) when a process is created.

Only processes that access SVA-capable devices need to have a PASID allocated. This allocation happens when a process opens/binds an SVA-capable device but finds no PASID for this process. Subsequent binds of the same, or other devices will share the same PASID.

Although the PASID is allocated to the process by opening a device, it is not active in any of the threads of that process. It's loaded to the `IA32_PASID` MSR lazily when a thread tries to submit a work descriptor to a device using the `ENQCMD`.

That first access will trigger a `#GP` fault because the `IA32_PASID` MSR has not been initialized with the PASID value assigned to the process when the device was opened. The Linux `#GP` handler notes that a PASID has been allocated for the process, and so initializes the `IA32_PASID` MSR and returns so that the `ENQCMD` instruction is re-executed.

On `fork(2)` or `exec(2)` the PASID is removed from the process as it no longer has the same address space that it had when the device was opened.

On `clone(2)` the new task shares the same address space, so will be able to use the PASID allocated to the process. The `IA32_PASID` is not preemptively initialized as the PASID value might not be allocated yet or the kernel does not know whether this thread is going to access the device and the cleared `IA32_PASID` MSR reduces context switch overhead by xstate init optimization. Since `#GP` faults have to be handled on any threads that were created before the PASID was assigned to the mm of the process, newly created threads might as well be treated in a consistent way.

Due to complexity of freeing the PASID and clearing all `IA32_PASID` MSRs in all threads in `unbind`, free the PASID lazily only on mm exit.

If a process does a `close(2)` of the device file descriptor and `munmap(2)` of the device MMIO portal, then the driver will unbind the device. The PASID is still marked `VALID` in the `PASID_MSR` for any threads in the process that accessed the device. But this is harmless as without the MMIO portal they cannot submit new work to the device.

## 30.7 Relationships

- Each process has many threads, but only one PASID.
- Devices have a limited number (~10's to 1000's) of hardware workqueues. The device driver manages allocating hardware workqueues.
- A single `mmap()` maps a single hardware workqueue as a "portal" and each portal maps down to a single workqueue.
- For each device with which a process interacts, there must be one or more `mmap()`'d portals.
- Many threads within a process can share a single portal to access a single device.

- Multiple processes can separately `mmap()` the same portal, in which case they still share one device hardware workqueue.
- The single process-wide PASID is used by all threads to interact with all devices. There is not, for instance, a PASID for each thread or each thread $\leftrightarrow$ device pair.

## 30.8 FAQ

- What is SVA/SVM?

Shared Virtual Addressing (SVA) permits I/O hardware and the processor to work in the same address space, i.e., to share it. Some call it Shared Virtual Memory (SVM), but Linux community wanted to avoid confusing it with POSIX Shared Memory and Secure Virtual Machines which were terms already in circulation.

- What is a PASID?

A Process Address Space ID (PASID) is a PCIe-defined Transaction Layer Packet (TLP) prefix. A PASID is a 20-bit number allocated and managed by the OS. PASID is included in all transactions between the platform and the device.

- How are shared workqueues different?

Traditionally, in order for userspace applications to interact with hardware, there is a separate hardware instance required per process. For example, consider doorbells as a mechanism of informing hardware about work to process. Each doorbell is required to be spaced 4k (or page-size) apart for process isolation. This requires hardware to provision that space and reserve it in MMIO. This doesn't scale as the number of threads becomes quite large. The hardware also manages the queue depth for Shared Work Queues (SWQ), and consumers don't need to track queue depth. If there is no space to accept a command, the device will return an error indicating retry.

A user should check Deferrable Memory Write (DMWr) capability on the device and only submits ENQCMD when the device supports it. In the new DMWr PCIe terminology, devices need to support DMWr completer capability. In addition, it requires all switch ports to support DMWr routing and must be enabled by the PCIe subsystem, much like how PCIe atomic operations are managed for instance.

SWQ allows hardware to provision just a single address in the device. When used with ENQCMD to submit work, the device can distinguish the process submitting the work since it will include the PASID assigned to that process. This helps the device scale to a large number of processes.

- Is this the same as a user space device driver?

Communicating with the device via the shared workqueue is much simpler than a full blown user space driver. The kernel driver does all the initialization of the hardware. User space only needs to worry about submitting work and processing completions.

- Is this the same as SR-IOV?

Single Root I/O Virtualization (SR-IOV) focuses on providing independent hardware interfaces for virtualizing hardware. Hence, it's required to be almost fully functional interface to software supporting the traditional BARs, space for interrupts via MSI-X, its own register layout. Virtual Functions (VFs) are assisted by the Physical Function (PF) driver.

Scalable I/O Virtualization builds on the PASID concept to create device instances for virtualization. SIOV requires host software to assist in creating virtual devices; each virtual device is represented by a PASID along with the bus/device/function of the device. This allows device hardware to optimize device resource creation and can grow dynamically on demand. SR-IOV creation and management is very static in nature. Consult references below for more details.

- Why not just create a virtual function for each app?

Creating PCIe SR-IOV type Virtual Functions (VF) is expensive. VFs require duplicated hardware for PCI config space and interrupts such as MSI-X. Resources such as interrupts have to be hard partitioned between VFs at creation time, and cannot scale dynamically on demand. The VFs are not completely independent from the Physical Function (PF). Most VFs require some communication and assistance from the PF driver. SIOV, in contrast, creates a software-defined device where all the configuration and control aspects are mediated via the slow path. The work submission and completion happen without any mediation.

- Does this support virtualization?

ENQCMD can be used from within a guest VM. In these cases, the VMM helps with setting up a translation table to translate from Guest PASID to Host PASID. Please consult the ENQCMD instruction set reference for more details.

- Does memory need to be pinned?

When devices support SVA along with platform hardware such as IOMMU supporting such devices, there is no need to pin memory for DMA purposes. Devices that support SVA also support other PCIe features that remove the pinning requirement for memory.

Device TLB support - Device requests the IOMMU to lookup an address before use via Address Translation Service (ATS) requests. If the mapping exists but there is no page allocated by the OS, IOMMU hardware returns that no mapping exists.

Device requests the virtual address to be mapped via Page Request Interface (PRI). Once the OS has successfully completed the mapping, it returns the response back to the device. The device requests again for a translation and continues.

IOMMU works with the OS in managing consistency of page-tables with the device. When removing pages, it interacts with the device to remove any device TLB entry that might have been cached before removing the mappings from the OS.

## 30.9 References

VT-D: <https://01.org/blogs/ashokraj/2018/recent-enhancements-intel-virtualization-technology-director-o-intel-vt-d>

SIOV: <https://01.org/blogs/2019/assignable-interfaces-intel-scalable-i/o-virtualization-linux>

ENQCMD in ISE: <https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf>

DSA spec: <https://software.intel.com/sites/default/files/341204-intel-data-streaming-accelerator-spec.pdf>



## SOFTWARE GUARD EXTENSIONS (SGX)

### 31.1 Overview

Software Guard eXtensions (SGX) hardware enables for user space applications to set aside private memory regions of code and data:

- Privileged (ring-0) ENCLS functions orchestrate the construction of the regions.
- Unprivileged (ring-3) ENCLU functions allow an application to enter and execute inside the regions.

These memory regions are called enclaves. An enclave can be only entered at a fixed set of entry points. Each entry point can hold a single hardware thread at a time. While the enclave is loaded from a regular binary file by using ENCLS functions, only the threads inside the enclave can access its memory. The region is denied from outside access by the CPU, and encrypted before it leaves from LLC.

The support can be determined by

```
grep sgx /proc/cpuinfo
```

SGX must both be supported in the processor and enabled by the BIOS. If SGX appears to be unsupported on a system which has hardware support, ensure support is enabled in the BIOS. If a BIOS presents a choice between “Enabled” and “Software Enabled” modes for SGX, choose “Enabled”.

### 31.2 Enclave Page Cache

SGX utilizes an *Enclave Page Cache (EPC)* to store pages that are associated with an enclave. It is contained in a BIOS-reserved region of physical memory. Unlike pages used for regular memory, pages can only be accessed from outside of the enclave during enclave construction with special, limited SGX instructions.

Only a CPU executing inside an enclave can directly access enclave memory. However, a CPU executing inside an enclave may access normal memory outside the enclave.

The kernel manages enclave memory similar to how it treats device memory.

### 31.2.1 Enclave Page Types

**SGX Enclave Control Structure (SECS)** Enclave's address range, attributes and other global data are defined by this structure.

**Regular (REG)** Regular EPC pages contain the code and data of an enclave.

**Thread Control Structure (TCS)** Thread Control Structure pages define the entry points to an enclave and track the execution state of an enclave thread.

**Version Array (VA)** Version Array pages contain 512 slots, each of which can contain a version number for a page evicted from the EPC.

### 31.2.2 Enclave Page Cache Map

The processor tracks EPC pages in a hardware metadata structure called the *Enclave Page Cache Map (EPCM)*. The EPCM contains an entry for each EPC page which describes the owning enclave, access rights and page type among the other things.

EPCM permissions are separate from the normal page tables. This prevents the kernel from, for instance, allowing writes to data which an enclave wishes to remain read-only. EPCM permissions may only impose additional restrictions on top of normal x86 page permissions.

For all intents and purposes, the SGX architecture allows the processor to invalidate all EPCM entries at will. This requires that software be prepared to handle an EPCM fault at any time. In practice, this can happen on events like power transitions when the ephemeral key that encrypts enclave memory is lost.

## 31.3 Application interface

### 31.3.1 Enclave build functions

In addition to the traditional compiler and linker build process, SGX has a separate enclave "build" process. Enclaves must be built before they can be executed (entered). The first step in building an enclave is opening the **/dev/sgx\_enclave** device. Since enclave memory is protected from direct access, special privileged instructions are then used to copy data into enclave pages and establish enclave page permissions.

```
long sgx_ioc_enclave_create(struct sgx_encl *encl, void __user *arg)
    handler for SGX_IOC_ENCLAVE_CREATE
```

#### Parameters

**struct sgx\_encl \*encl** An enclave pointer.

**void \_\_user \*arg** The ioctl argument.

#### Description

Allocate kernel data structures for the enclave and invoke ECREATE.

#### Return

- 0: Success.
- -EIO: ECREATE failed.

- -errno: POSIX error.

long **sgx\_ioc\_enclave\_add\_pages**(struct sgx\_encl \*encl, void \_\_user \*arg)  
The handler for SGX\_IOC\_ENCLAVE\_ADD\_PAGES

### Parameters

**struct sgx\_encl \*encl** an enclave pointer

**void \_\_user \*arg** a user pointer to a struct sgx\_enclave\_add\_pages instance

### Description

Add one or more pages to an uninitialized enclave, and optionally extend the measurement with the contents of the page. The SECINFO and measurement mask are applied to all pages.

A SECINFO for a TCS is required to always contain zero permissions because CPU silently zeros them. Allowing anything else would cause a mismatch in the measurement.

mmap()'s protection bits are capped by the page permissions. For each page address, the maximum protection bits are computed with the following heuristics:

1. A regular page: PROT\_R, PROT\_W and PROT\_X match the SECINFO permissions.
2. A TCS page: PROT\_R | PROT\_W.

mmap() is not allowed to surpass the minimum of the maximum protection bits within the given address range.

The function deinitializes kernel data structures for enclave and returns -EIO in any of the following conditions:

- Enclave Page Cache (EPC), the physical memory holding enclaves, has been invalidated. This will cause EADD and EEXTEND to fail.
- If the source address is corrupted somehow when executing EADD.

### Return

- 0: Success.
- -EACCES: The source page is located in a noexec partition.
- -ENOMEM: Out of EPC pages.
- -EINTR: The call was interrupted before data was processed.
- **-EIO: Either EADD or EEXTEND failed because invalid source address** or power cycle.
- -errno: POSIX error.

long **sgx\_ioc\_enclave\_init**(struct sgx\_encl \*encl, void \_\_user \*arg)  
handler for SGX\_IOC\_ENCLAVE\_INIT

### Parameters

**struct sgx\_encl \*encl** an enclave pointer

**void \_\_user \*arg** userspace pointer to a struct sgx\_enclave\_init instance

### Description

Flush any outstanding enqueued EADD operations and perform EINIT. The Launch Enclave Public Key Hash MSRs are rewritten as necessary to match the enclave's MRSIGNER, which is caculated from the provided sigstruct.

**Return**

- 0: Success.
- -EPERM: Invalid SIGSTRUCT.
- -EIO: EINIT failed because of a power cycle.
- -errno: POSIX error.

long **sgx\_ioc\_enclave\_provision**(struct sgx\_encl \*encl, void \_\_user \*arg)  
handler for SGX\_IOC\_ENCLAVE\_PROVISION

**Parameters**

**struct sgx\_encl \*encl** an enclave pointer

**void \_\_user \*arg** userspace pointer to a struct sgx\_enclave\_provision instance

**Description**

Allow `ATTRIBUTE.PROVISION_KEY` for an enclave by providing a file handle to `/dev/sgx_provision`.

**Return**

- 0: Success.
- -errno: Otherwise.

### 31.3.2 Enclave vDSO

Entering an enclave can only be done through SGX-specific EENTER and ERESUME functions, and is a non-trivial process. Because of the complexity of transitioning to and from an enclave, enclaves typically utilize a library to handle the actual transitions. This is roughly analogous to how glibc implementations are used by most applications to wrap system calls.

Another crucial characteristic of enclaves is that they can generate exceptions as part of their normal operation that need to be handled in the enclave or are unique to SGX.

Instead of the traditional signal mechanism to handle these exceptions, SGX can leverage special exception fixup provided by the vDSO. The kernel-provided vDSO function wraps low-level transitions to/from the enclave like EENTER and ERESUME. The vDSO function intercepts exceptions that would otherwise generate a signal and return the fault information directly to its caller. This avoids the need to juggle signal handlers.

**vdso\_sgx\_enter\_enclave\_t**

**Typedef:** Prototype for `__vdso_sgx_enter_enclave()`, a vDSO function to enter an SGX enclave.

**Syntax**

```
int vdso_sgx_enter_enclave_t (unsigned long rdi, unsigned long rsi,  
unsigned long rdx, unsigned int function, unsigned long r8, unsigned  
long r9, struct sgx_enclave_run *run)
```

**Parameters**



**unsigned long rdi** Pass-through value for RDI

**unsigned long rsi** Pass-through value for RSI

**unsigned long rdx** Pass-through value for RDX

**unsigned int function** ENCLU function, must be EENTER or ERESUME

**unsigned long r8** Pass-through value for R8

**unsigned long r9** Pass-through value for R9

**struct sgx\_enclave\_run \*run** struct sgx\_enclave\_run, must be non-NULL

#### NOTE

`__vdso_sgx_enter_enclave()` does not ensure full compliance with the x86-64 ABI, e.g. doesn't handle XSAVE state. Except for non-volatile general purpose registers, EFLAGS.DF, and RSP alignment, preserving/setting state in accordance with the x86-64 ABI is the responsibility of the enclave and its runtime, i.e. `__vdso_sgx_enter_enclave()` cannot be called from C code without careful consideration by both the enclave and its runtime.

#### Description

All general purpose registers except RAX, RBX and RCX are passed as-is to the enclave. RAX, RBX and RCX are consumed by EENTER and ERESUME and are loaded with **function**, asynchronous exit pointer, and **run.tcs** respectively.

RBP and the stack are used to anchor `__vdso_sgx_enter_enclave()` to the pre-enclave state, e.g. to retrieve **run.exception** and **run.user\_handler** after an enclave exit. All other registers are available for use by the enclave and its runtime, e.g. an enclave can push additional data onto the stack (and modify RSP) to pass information to the optional user handler (see below).

Most exceptions reported on ENCLU, including those that occur within the enclave, are fixed up and reported synchronously instead of being delivered via a standard signal. Debug Exceptions (#DB) and Breakpoints (#BP) are never fixed up and are always delivered via standard signals. On synchronously reported exceptions, -EFAULT is returned and details about the exception are recorded in **run.exception**, the optional sgx\_enclave\_exception struct.

#### Return

- 0: ENCLU function was successfully executed.
- -EINVAL: Invalid ENCL number (neither EENTER nor ERESUME).

## 31.4 ksgxd

SGX support includes a kernel thread called *ksgxd*.

### 31.4.1 EPC sanitization

`ksgxd` is started when SGX initializes. Enclave memory is typically ready for use when the processor powers on or resets. However, if SGX has been in use since the reset, enclave pages may be in an inconsistent state. This might occur after a crash and `kexec()` cycle, for instance. At boot, `ksgxd` reinitializes all enclave pages so that they can be allocated and re-used.

The sanitization is done by going through EPC address space and applying the `EREMOVE` function to each physical page. Some enclave pages like SECS pages have hardware dependencies on other pages which prevents `EREMOVE` from functioning. Executing two `EREMOVE` passes removes the dependencies.

### 31.4.2 Page reclaimer

Similar to the core `kswapd`, `ksgxd`, is responsible for managing the overcommitment of enclave memory. If the system runs out of enclave memory, `ksgxd` “swaps” enclave memory to normal memory.

## 31.5 Launch Control

SGX provides a launch control mechanism. After all enclave pages have been copied, kernel executes `EINIT` function, which initializes the enclave. Only after this the CPU can execute inside the enclave.

`EINIT` function takes an RSA-3072 signature of the enclave measurement. The function checks that the measurement is correct and signature is signed with the key hashed to the four **IA32\_SGXLEPUBKEYHASH{0, 1, 2, 3}** MSRs representing the SHA256 of a public key.

Those MSRs can be configured by the BIOS to be either readable or writable. Linux supports only writable configuration in order to give full control to the kernel on launch control policy. Before calling `EINIT` function, the driver sets the MSRs to match the enclave’s signing key.

## 31.6 Encryption engines

In order to conceal the enclave data while it is out of the CPU package, the memory controller has an encryption engine to transparently encrypt and decrypt enclave memory.

In CPUs prior to Ice Lake, the Memory Encryption Engine (MEE) is used to encrypt pages leaving the CPU caches. MEE uses a n-ary Merkle tree with root in SRAM to maintain integrity of the encrypted data. This provides integrity and anti-replay protection but does not scale to large memory sizes because the time required to update the Merkle tree grows logarithmically in relation to the memory size.

CPUs starting from Icelake use Total Memory Encryption (TME) in the place of MEE. TME-based SGX implementations do not have an integrity Merkle tree, which means integrity and replay-attacks are not mitigated. B, it includes additional changes to prevent cipher text from being returned and SW memory aliases from being created.

DMA to enclave memory is blocked by range registers on both MEE and TME systems (SDM section 41.10).

## 31.7 Usage Models

### 31.7.1 Shared Library

Sensitive data and the code that acts on it is partitioned from the application into a separate library. The library is then linked as a DSO which can be loaded into an enclave. The application can then make individual function calls into the enclave through special SGX instructions. A run-time within the enclave is configured to marshal function parameters into and out of the enclave and to call the correct library function.

### 31.7.2 Application Container

An application may be loaded into a container enclave which is specially configured with a library OS and run-time which permits the application to run. The enclave run-time and library OS work together to execute the application when a thread enters the enclave.

## 31.8 Impact of Potential Kernel SGX Bugs

### 31.8.1 EPC leaks

When EPC page leaks happen, a WARNING like this is shown in dmesg:

“EREMOVE returned ... and an EPC page was leaked. SGX may become unusable...”

This is effectively a kernel use-after-free of an EPC page, and due to the way SGX works, the bug is detected at freeing. Rather than adding the page back to the pool of available EPC pages, the kernel intentionally leaks the page to avoid additional errors in the future.

When this happens, the kernel will likely soon leak more EPC pages, and SGX will likely become unusable because the memory available to SGX is limited. However, while this may be fatal to SGX, the rest of the kernel is unlikely to be impacted and should continue to work.

As a result, when this happens, user should stop running any new SGX workloads, (or just any new workloads), and migrate all valuable workloads. Although a machine reboot can recover all EPC memory, the bug should be reported to Linux developers.

## 31.9 Virtual EPC

The implementation has also a virtual EPC driver to support SGX enclaves in guests. Unlike the SGX driver, an EPC page allocated by the virtual EPC driver doesn't have a specific enclave associated with it. This is because KVM doesn't track how a guest uses EPC pages.

As a result, the SGX core page reclaimer doesn't support reclaiming EPC pages allocated to KVM guests through the virtual EPC driver. If the user wants to deploy SGX applications both on the host and in guests on the same machine, the user should reserve enough EPC (by taking out total virtual EPC size of all SGX VMs from the physical EPC size) for host SGX applications so they can run with acceptable performance.

Architectural behavior is to restore all EPC pages to an uninitialized state also after a guest reboot. Because this state can be reached only through the privileged ENCLS[EREMOVE] instruction, `/dev/sgx_vepc` provides the `SGX_IOC_VEPC_REMOVE_ALL` ioctl to execute the instruction on all pages in the virtual EPC.

EREMOVE can fail for three reasons. Userspace must pay attention to expected failures and handle them as follows:

1. Page removal will always fail when any thread is running in the enclave to which the page belongs. In this case the ioctl will return `EBUSY` independent of whether it has successfully removed some pages; userspace can avoid these failures by preventing execution of any vcpu which maps the virtual EPC.
2. Page removal will cause a general protection fault if two calls to `EREMOVE` happen concurrently for pages that refer to the same “SECS” metadata pages. This can happen if there are concurrent invocations to `SGX_IOC_VEPC_REMOVE_ALL`, or if a `/dev/sgx_vepc` file descriptor in the guest is closed at the same time as `SGX_IOC_VEPC_REMOVE_ALL`; it will also be reported as `EBUSY`. This can be avoided in userspace by serializing calls to the ioctl() and to `close()`, but in general it should not be a problem.
3. Finally, page removal will fail for SECS metadata pages which still have child pages. Child pages can be removed by executing `SGX_IOC_VEPC_REMOVE_ALL` on all `/dev/sgx_vepc` file descriptors mapped into the guest. This means that the ioctl() must be called twice: an initial set of calls to remove child pages and a subsequent set of calls to remove SECS pages. The second set of calls is only required for those mappings that returned a nonzero value from the first call. It indicates a bug in the kernel or the userspace client if any of the second round of `SGX_IOC_VEPC_REMOVE_ALL` calls has a return code other than 0.

## FEATURE STATUS ON X86 ARCHITECTURE

Subsystem	Feature	Kconfig	Status	Description
core	cBPF-JIT	HAVE_CBPF_JIT	TODO	architecture
core	eBPF-JIT	HAVE_EBPF_JIT	ok	architecture
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok	architecture
core	jump-labels	HAVE_ARCH_JUMP_LABEL	ok	architecture
core	thread-info-in-task	THREAD_INFO_IN_TASK	ok	architecture
core	tracehook	HAVE_ARCH_TRACEHOOK	ok	architecture
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	ok	architecture
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	ok	architecture
debug	KASAN	HAVE_ARCH_KASAN	ok	architecture
debug	kcov	ARCH_HAS_KCOV	ok	architecture
debug	kgdb	HAVE_ARCH_KGDB	ok	architecture
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	ok	architecture
debug	kprobes	HAVE_KPROBES	ok	architecture
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	ok	architecture
debug	kretprobes	HAVE_KRETPROBES	ok	architecture
debug	optprobes	HAVE_OPTPROBES	ok	architecture
debug	stackprotector	HAVE_STACKPROTECTOR	ok	architecture
debug	uprobes	ARCH_SUPPORTS_UPROBES	ok	architecture
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	ok	architecture
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	ok	architecture
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	ok	architecture
locking	lockdep	LOCKDEP_SUPPORT	ok	architecture
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	ok	architecture
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	ok	architecture
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	ok	architecture
perf	perf-regs	HAVE_PERF_REGS	ok	architecture
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	ok	architecture
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	ok	architecture
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	ok	architecture
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	ok	architecture
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	TODO	architecture
time	clockevents	!LEGACY_TIMER_TICK	ok	architecture
time	context-tracking	HAVE_CONTEXT_TRACKING	ok	architecture
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	ok	architecture
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	ok	architecture

Table 1 - continued from p

Subsystem	Feature	Kconfig	Status	Desc
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	ok	arch
vm	ELF-ASLR	ARCH_HAS_ELF_RANDOMIZE	ok	arch
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	ok	arch
vm	ioremap_prot	HAVE_IOREMAP_PROT	ok	arch
vm	PG_uncached	ARCH_USES_PG_UNCACHED	ok	arch
vm	pte_special	ARCH_HAS_PTE_SPECIAL	ok	arch
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	ok	arch

## X86-SPECIFIC ELF AUXILIARY VECTORS

This document describes the semantics of the x86 auxiliary vectors.

### 33.1 Introduction

ELF Auxiliary vectors enable the kernel to efficiently provide configuration-specific parameters to userspace. In this example, a program allocates an alternate stack based on the kernel-provided size:

```
#include <sys/auxv.h>
#include <elf.h>
#include <signal.h>
#include <stdlib.h>
#include <assert.h>
#include <err.h>

#ifdef AT_MINSIGSTKSZ
#define AT_MINSIGSTKSZ      51
#endif

....
stack_t ss;

ss.ss_sp = malloc(ss.ss_size);
assert(ss.ss_sp);

ss.ss_size = getauxval(AT_MINSIGSTKSZ) + SIGSTKSZ;
ss.ss_flags = 0;

if (sigaltstack(&ss, NULL))
    err(1, "sigaltstack");
```

## 33.2 The exposed auxiliary vectors

AT\_SYSINFO is used for locating the vsyscall entry point. It is not exported on 64-bit mode.

AT\_SYSINFO\_EHDR is the start address of the page containing the vDSO.

AT\_MINSIGSTKSZ denotes the minimum stack size required by the kernel to deliver a signal to user-space. AT\_MINSIGSTKSZ comprehends the space consumed by the kernel to accommodate the user context for the current hardware configuration. It does not comprehend subsequent user-space stack consumption, which must be added by the user. (e.g. Above, user-space adds SIGSTKSZ to AT\_MINSIGSTKSZ.)



## **USING XSTATE FEATURES IN USER SPACE APPLICATIONS**

The x86 architecture supports floating-point extensions which are enumerated via CPUID. Applications consult CPUID and use XGETBV to evaluate which features have been enabled by the kernel XCR0.

Up to AVX-512 and PKRU states, these features are automatically enabled by the kernel if available. Features like AMX TILE\_DATA (XSTATE component 18) are enabled by XCR0 as well, but the first use of related instruction is trapped by the kernel because by default the required large XSTATE buffers are not allocated automatically.

### **34.1 Using dynamically enabled XSTATE features in user space applications**

The kernel provides an `arch_prctl(2)` based mechanism for applications to request the usage of such features. The `arch_prctl(2)` options related to this are:

**-ARCH\_GET\_XCOMP\_SUPP** `arch_prctl(ARCH_GET_XCOMP_SUPP, &features);`

`ARCH_GET_XCOMP_SUPP` stores the supported features in userspace storage of type `uint64_t`. The second argument is a pointer to that storage.

**-ARCH\_GET\_XCOMP\_PERM** `arch_prctl(ARCH_GET_XCOMP_PERM, &features);`

`ARCH_GET_XCOMP_PERM` stores the features for which the userspace process has permission in userspace storage of type `uint64_t`. The second argument is a pointer to that storage.

**-ARCH\_REQ\_XCOMP\_PERM** `arch_prctl(ARCH_REQ_XCOMP_PERM, feature_nr);`

`ARCH_REQ_XCOMP_PERM` allows to request permission for a dynamically enabled feature or a feature set. A feature set can be mapped to a facility, e.g. AMX, and can require one or more XSTATE components to be enabled.

The feature argument is the number of the highest XSTATE component which is required for a facility to work.

When requesting permission for a feature, the kernel checks the availability. The kernel ensures that sigaltstacks in the process's tasks are large enough to accommodate the resulting large signal frame. It enforces this both during `ARCH_REQ_XCOMP_SUPP` and during any subsequent `sigaltstack(2)` calls. If an installed sigaltstack is smaller than the resulting sigframe

size, `ARCH_REQ_XCOMP_SUPP` results in `-ENOSUPP`. Also, `sigaltstack(2)` results in `-ENOMEM` if the requested altstack is too small for the permitted features.

Permission, when granted, is valid per process. Permissions are inherited on `fork(2)` and cleared on `exec(3)`.

The first use of an instruction related to a dynamically enabled feature is trapped by the kernel. The trap handler checks whether the process has permission to use the feature. If the process has no permission then the kernel sends `SIGILL` to the application. If the process has permission then the handler allocates a larger xstate buffer for the task so the large state can be context switched. In the unlikely cases that the allocation fails, the kernel sends `SIGSEGV`.

### 34.2 Dynamic features in signal frames

Dynamically enabled features are not written to the signal frame upon signal entry if the feature is in its initial configuration. This differs from non-dynamic features which are always written regardless of their configuration. Signal handlers can examine the `XSAVE` buffer's `XSTATE_BV` field to determine if a feature was written.

### I

`ifs_data` (*C struct*), [164](#)

### S

`sgx_ioc_enclave_add_pages` (*C function*),  
[175](#)

`sgx_ioc_enclave_create` (*C function*), [174](#)

`sgx_ioc_enclave_init` (*C function*), [175](#)

`sgx_ioc_enclave_provision` (*C function*),  
[176](#)

### V

`vdso_sgx_enter_enclave_t` (*C macro*), [176](#)