Linux X86 Documentation

The kernel development community

Jul 14, 2020

CONTENTS

CHAPTER

ONE

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	(Kernel 3.14) Support 32- and 64-bit flags being set in xloadflags to support

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

Continued on next page

Table 1 – continued from previous page

Offset/Size	Proto	Name	Meaning
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_swtch	Boot loader hook (see below)
020C/2	2.00+	start_sys_seg	The load-low segment (0x1000) (obsolete)
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 kerne
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 struc
0258/8	2.10+	pref_address	Preferred loading address
0260/4	2.10+	init_size	Linear memory required during initializatio
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_swtch
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 <http://sebastian-plotz.blogspot.de>
12	OVMF UEFI virtualization stack

Please contact <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 nonstandard 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 (type_of_loader & 0x0f) + (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 (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 (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 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), and LZ4 (magic number 02 21). 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_data);
```

(continues on next page)

(continued from previous page)

```
  __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:	pref_address
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:
```

(continues on next page)

(continued from previous page)

```
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 Configuartion

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;
```

(continues on next page)

(continued from previous page)

```
                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 (setup_sects+1)*512 in the kernel file (again, if 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 >= 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_swtch: 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 that described in zero-page.txt.

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 zero-page.txt.

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

CHAPTER

TWO

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> and Borislav Petkov <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

2.1 Package

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

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.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.

- 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.

2.2 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:

```
threads_per_package = cpuinfo_x86.x86_max_cores * smp_num_siblings
```

2.3 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."

2.4 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
```

(continues on next page)

(continued from previous page)

```
            -> [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
```

CHAPTER

THREE

KERNEL LEVEL EXCEPTION HANDLING

Commentary by Joerg Pommnitz <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 do_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.

do_page_fault first obtains the unaccessible 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__(
```

(continues on next page)

(continued from previous page)

```
              "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   $0xfffffff2,%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 $0xfffffff2,%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 do_page_fault
4. do page fault calls search_exception_table (regs->eip == c017e7a5);
5. search_exception_table 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.
6. do_page_fault modifies its own return address to point to the fault handle code and returns.
7. execution continues in the fault handling code.
8. 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!

CHAPTER

FOUR

KERNEL STACKS

4.1 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 adjusts 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.

4.2 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)[1], from stack top to stack bottom, and print out anything that 'looks like' a kernel text address.

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.

[1] 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.

CHAPTER

FIVE

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:

http://lkml.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 shootdown.
- 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 gsbase correctly. This triggers three main behavior changes:

- Interrupt entry will use the slower gsbase 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.

CHAPTER

SIX

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 and two USB cables, connected like this:

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

6.1 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.

6.2 Software requirements

a) On the host/target system:

   You need to enable the following kernel config option:

```
CONFIG_EARLY_PRINTK_DBGP=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.

6.3 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 mailing list or contact the x86 maintainers.

CHAPTER

SEVEN

ORC UNWINDER

7.1 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.

7.2 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[1] 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."

[1] https://lkml.kernel.org/r/20170602104048.jkkzssljsompjdwy@suse.de

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.

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.

7.3 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[2], 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.

7.4 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

[2] https://lkml.kernel.org/r/d2ca5435-6386-29b8-db87-7f227c2b713a@suse.cz

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 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.

7.5 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.

7.6 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."[3] Similarly, unlike the esoteric DWARF unwinder, the veracious 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.

[3] http://dustin.wikidot.com/half-orcs-and-orcs

CHAPTER

EIGHT

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
```

Off-set/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)
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
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)

CHAPTER

NINE

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/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[1], hugetlbfs always uses the full flushes. THP is treated exactly the same as normal memory.

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.

[1] 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."

CHAPTER

TEN

MTRR (MEMORY TYPE RANGE REGISTER) CONTROL

Authors

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

10.1 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.

10.2 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.

10.3 Creating overlapping MTRRs

```
%echo "base=0xfb000000 size=0x1000000 type=write-combining" >/proc/mtrr
%echo "base=0xfb000000 size=0x1000 type=uncachable" >/proc/mtrr
```

And the results:

```
% cat /proc/mtrr
reg00: base=0x00000000 (   0MB), size=  64MB: write-back, count=1
reg01: base=0xfb000000 (4016MB), size=  16MB: write-combining, count=1
reg02: base=0xfb000000 (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.

10.4 Removing MTRRs from the C-shel

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

or using bash:

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

10.5 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.

```

(continues on next page)

(continued from previous page)

```
    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",
```

(continues on next page)

(continued from previous page)

```
      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  */
```

10.6 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.
*/
```

(continues on next page)

(continued from previous page)

```
/*
    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)
```

(continues on next page)

(continued from previous page)

```
    {
  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  */
```

CHAPTER

ELEVEN

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

11.1 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 resource_wc is IORESOURCE_PREFETCH	–	–	WC
pci proc !PCIIOC_WRITE_COMBINE	–	–	UC-
pci proc PCIIOC_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-

11.2 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.

11.3 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 uncacheable 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	
	PAT			Non-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.

11.4 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.

11.5 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

CHAPTER

TWELVE

LINUX IOMMU SUPPORT

The architecture spec can be obtained from the below location.

http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf

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

Some Keywords

- DMAR - DMA remapping
- DRHD - DMA Remapping Hardware Unit Definition
- RMRR - Reserved memory Region Reporting Structure
- ZLR - Zero length reads from PCI devices
- IOVA - IO Virtual address.

12.1 Basic stuff

ACPI enumerates and lists the different DMA engines in the platform, and device scope relationships between PCI devices and which DMA engine controls them.

12.2 What is 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.

12.3 How is IOVA generated?

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

The Intel IOMMU driver allocates a virtual address per domain. Each PCIE device has its own domain (hence protection). Devices under p2p bridges share the virtual address with all devices under the p2p bridge due to transaction id aliasing for p2p bridges.

IOVA generation is pretty generic. We used the same technique as vmalloc() but these are not global address spaces, but separate for each domain. Different DMA engines may support different number of domains.

We also allocate guard pages with each mapping, so we can attempt to catch any overflow that might happen.

12.4 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.

12.5 Some exceptions to IOVA

Interrupt ranges are not address translated, (0xfee00000 - 0xfeeffffff). 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.

12.6 Fault reporting

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

See below for sample.

12.7 Boot Message Sample

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: 0x00000000000ed000 end: 0x00000000000effff
ACPI DMAR:RMRR base: 0x000000007f600000 end: 0x000000007fffffff
```

When DMAR is enabled for use, you will notice..

12.8 PCI-DMA: Using DMAR IOMMU

12.8.1 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
```

12.9 TBD

- For compatibility testing, could use unity map domain for all devices, just provide a 1-1 for all useful memory under a single domain for all devices.
- API for paravirt ops for abstracting functionality for VMM folks.

CHAPTER

THIRTEEN

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
- **IDF, Shanghai:** http://www.prcidf.com.cn/index_en.html
- **IDFs 2006, 2007** (I'm not sure if/where they are online)

13.1 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.

13.2 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).

13.3 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.

13.4 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
```

(continues on next page)

(continued from previous page)

```
            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.

CHAPTER

FOURTEEN

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
```

(continues on next page)

(continued from previous page)

```
                  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_K8_SYSCFG) can be used to determine if SME is enabled and/or to enable memory encryption:

```
0xc0010010:
        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_K8_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_K8_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.

CHAPTER

FIFTEEN

PAGE TABLE ISOLATION (PTI)

15.1 Overview

Page Table Isolation (pti, previously known as KAISER[1]) is a countermeasure against attacks on the shared user/kernel address space such as the "Meltdown" approach[2].

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).

15.2 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

[1] https://gruss.cc/files/kaiser.pdf
[2] https://meltdownattack.com/meltdown.pdf

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 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…

15.3 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 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.

15.4 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.

15.5 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.

15.6 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.

CHAPTER

SIXTEEN

MICROARCHITECTURAL DATA SAMPLING (MDS) MITIGATION

16.1 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.

16.2 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 extremly 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.

16.3 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.

16.4 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.

16.5 Mitigation points

16.5.1 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 migitation 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.)

16.5.2 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.

CHAPTER

SEVENTEEN

THE LINUX MICROCODE LOADER

Authors

- Fenghua Yu <fenghua.yu@intel.com>
- Borislav Petkov <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:

17.1 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.

17.2 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.

17.3 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.

CHAPTER

EIGHTEEN

USER INTERFACE FOR RESOURCE CONTROL FEATURE

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.

18.1 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.

If RDT monitoring is available there will be an “L3_MON” directory with the following files:

“num_rmids” : 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
```

18.2 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.

18.2.1 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.

18.2.2 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.

18.3 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.

18.3.1 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.

18.3.2 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.

18.3.3 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

18.3.4 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.

18.4 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: min_bw + N * 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 will result in both threads being throttled to use the low bandwidth. The fact that Memory bandwidth allocation(MBA) is 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.

18.4.1 L3 schemata file details (code and data prioritization disabled)

With CDP disabled the L3 schemata format is:

```
L3:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...
```

18.4.2 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>;...
```

18.4.3 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>;…
```

18.4.4 Memory bandwidth Allocation (default mode)

Memory b/w domain is L3 cache.

```
MB:<cache_id0>=bandwidth0;<cache_id1>=bandwidth1;...
```

18.4.5 Memory bandwidth Allocation specified in MBps

Memory bandwidth domain is L3 cache.

```
MB:<cache_id0>=bw_MBps0;<cache_id1>=bw_MBps1;...
```

18.4.6 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
```

18.5 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 INVD, 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.

18.5.1 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.

18.5.2 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
#
```

(continues on next page)

(continued from previous page)

```
#                              _-----=> 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=fffff\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=fffff" > 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=fffff" > 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=fffff\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=fffff\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
```

(continues on next page)

(continued from previous page)

```
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);
```

(continues on next page)

(continued from previous page)

```
  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);
}
```

18.5.3 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);
```

(continues on next page)

(continued from previous page)

```
  }
}

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);
}
```

18.6 Examples for RDT Monitoring along with allocation usage

18.6.1 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.

18.6.2 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
```

18.6.3 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
```

18.6.4 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
```

18.6.5 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
```

CHAPTER

NINETEEN

TSX ASYNC ABORT (TAA) MITIGATION

19.1 Overview

TSX Async Abort (TAA) is a side channel attack on internal buffers in some Intel processors similar to Microachitectural 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.

19.2 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.

19.3 Kernel internal mitigation modes

off	Mitigation is disabled. Either the CPU is not affected or tsx_async_abort=off 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	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	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).

CHAPTER

TWENTY

USB LEGACY SUPPORT

Author Vojtech Pavlik <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.

CHAPTER

TWENTYONE

I386 SUPPORT

21.1 IO-APIC

Author Ingo Molnar <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:

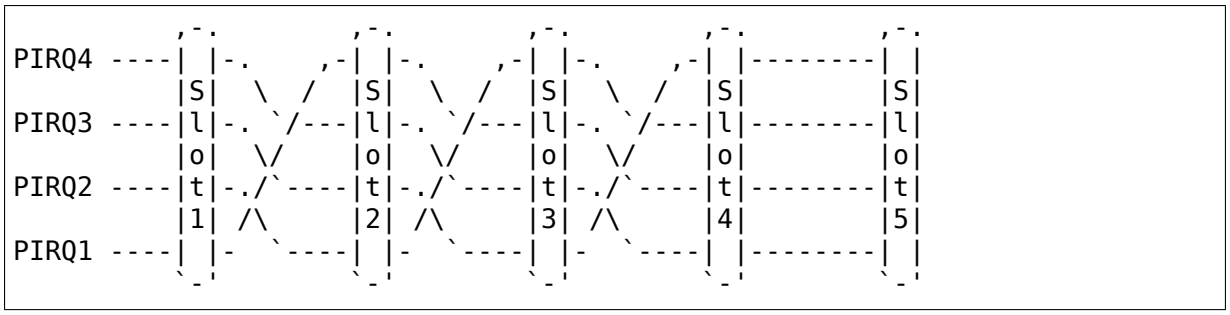
```
hell:~> cat /proc/interrupts
           CPU0
  0:    1360293    IO-APIC-edge  timer
  1:          4    IO-APIC-edge  keyboard
  2:          0          XT-PIC  cascade
 13:          1          XT-PIC  fpu
 14:       1448    IO-APIC-edge  ide0
 16:      28232   IO-APIC-level  Intel EtherExpress Pro 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.

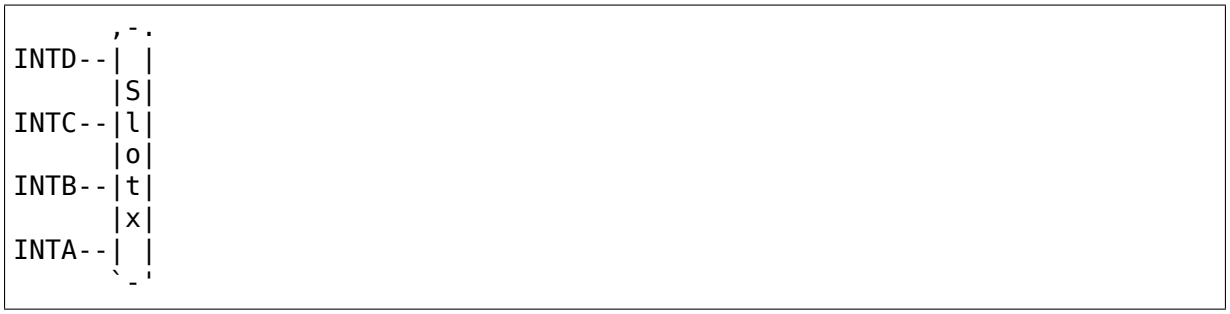
In the unlikely case that your board does not create a working mp-table, you can use the pirq= boot parameter to 'hand-construct' IRQ entries. This is non-trivial though and cannot be automated. One sample /etc/lilo.conf entry:

```
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-IRQ 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 or linux-kernel@vger.kernel.org if you have any problems that are not covered by this document.

CHAPTER

TWENTYTWO

X86_64 SUPPORT

22.1 AMD64 Specific Boot Options

There are many others (usually documented in driver documentation), but only the AMD64 specific ones are listed here.

22.1.1 Machine check

Please see Documentation/x86/x86_64/machinecheck.rst 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=tolerancelevel[,monarchtimeout] (number,number)
tolerance levels: 0: always panic on uncorrected errors, log corrected errors 1: panic or SIGBUS on uncorrected errors, log corrected errors 2: SIGBUS or log uncorrected errors, log corrected errors 3: never panic or SIGBUS, log all errors (for testing only) Default is 1 Can be also set using sysfs which is preferable. 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.

22.1.2 APICs

apic Use IO-APIC. Default

noapic Don't use the IO-APIC.

disableapic Don't use the local APIC

nolapic Don't use the local APIC (alias for i386 compatibility)

pirq=… See Documentation/x86/i386/IO-APIC.rst

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.

22.1.3 Timing

notsc Deprecated, use tsc=unstable instead.

nohpet Don't use the HPET timer.

22.1.4 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.

22.1.5 Rebooting

reboot=b[ios] | t[riple] | k[bd] | a[cpi] | e[fi] [, [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.

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.

22.1.6 Non Executable Mappings

noexec=on|off

on Enable(default)

off Disable

22.1.7 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=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.

22.1.8 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=nocmcff 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.

22.1.9 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.

22.1.10 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)"
4. <arch/x86_64/pci-calgary.c> : IBM Calgary hardware IOMMU. Used in IBM pSeries and xSeries servers. This hardware IOMMU supports DMA address mapping with memory protection, etc. Kernel boot message: "PCI-DMA: Using Calgary IOMMU"

```
iommu=[<size>][,noagp][,off][,force][,noforce]
[,memaper[=<order>]][,merge][,fullflush][,nomerge]
[,noaperture][,calgary]
```

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.

calgary Use the Calgary IOMMU if it is available

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.

Settings for the IBM Calgary hardware IOMMU currently found in IBM pSeries and xSeries machines

calgary=[64k,128k,256k,512k,1M,2M,4M,8M] Set the size of each PCI slot's translation table when using the Calgary IOMMU. This is the size of the translation table itself in main memory. The smallest table, 64k, covers an IO space of 32MB; the largest, 8MB table, can cover an IO space of 4GB. Normally the kernel will make the right choice by itself.

calgary=[translate_empty_slots] Enable translation even on slots that have no devices attached to them, in case a device will be hot-plugged in the future.

calgary=[disable=<PCI bus number>] Disable translation on a given PHB. For example, the built-in graphics adapter resides on the first bridge (PCI bus number 0); if translation (isolation) is enabled on this bridge, X servers that access the hardware directly from user space might stop working. Use this option if you have devices that are accessed from userspace directly on some PCI host bridge.

panic Always panic when IOMMU overflows

22.1.11 Miscellaneous

nogbpages Do not use GB pages for kernel direct mappings.

gbpages Use GB pages for kernel direct mappings.

22.2 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.

22.2.1 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

22.3 Memory Management

22.3.1 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 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    |   Offset   |     End addr     |  Size   | VM area␣
↪description
================================================================================
                  |            |                  |         |
 0000000000000000 |    0       | 00007fffffffffff |  128 TB | user-space␣
↪virtual memory, different per mm
__________________|____________|__________________|_________|_________________
↪_____________________________________________
                  |            |                  |         |
 0000800000000000 | +128    TB | ffff7fffffffffff | ~16M TB | ... huge,␣
↪almost 64 bits wide hole of non-canonical
                  |            |                  |         |     virtual␣
↪memory addresses up to the -128 TB
                  |            |                  |         |     starting␣
↪offset of kernel mappings.
__________________|____________|__________________|_________|_________________
↪_____________________________________________
                                                             |
                                                             | Kernel-space␣
↪virtual memory, shared between all processes:
_____________________________________________________________|_________________
↪_____________________________________________
                  |            |                  |         |
 ffff800000000000 | -128    TB | ffff87ffffffffff |    8 TB | ... guard␣
↪hole, also reserved for hypervisor
 ffff880000000000 | -120    TB | ffff887fffffffff |  0.5 TB | LDT remap␣
↪for PTI
 ffff888000000000 | -119.5  TB | ffffc87fffffffff |   64 TB | direct␣
↪mapping of all physical memory (page_offset_base)
 ffffc88000000000 |  -55.5  TB | ffffc8ffffffffff |  0.5 TB | ... unused␣
↪hole
 ffffc90000000000 |  -55    TB | ffffe8ffffffffff |   32 TB | vmalloc/
↪ioremap space (vmalloc_base)
 ffffe90000000000 |  -23    TB | ffffe9ffffffffff |    1 TB | ... unused␣
↪hole
 ffffea0000000000 |  -22    TB | ffffeaffffffffff |    1 TB | virtual␣
↪memory map (vmemmap_base)
 ffffeb0000000000 |  -21    TB | ffffebffffffffff |    1 TB | ... unused␣
↪hole
 ffffec0000000000 |  -20    TB | fffffbffffffffff |   16 TB | KASAN shadow␣
↪memory
__________________|____________|__________________|_________|_________________
↪_____________________________________________
                                                             |
                                                             | Identical␣
↪layout to the 56-bit one from here on:
_____________________________________________________________|_________________
↪_____________________________________________
                  |            |                  |         |
 fffffc0000000000 |   -4    TB | fffffdffffffffff |    2 TB | ... unused␣
↪hole
```

(continues on next page)

(continued from previous page)

```
                  |            |                  |           | vaddr_end␣
↪for KASLR
 ffffe0000000000 |   -2    TB | fffffe7fffffffff |  0.5 TB | cpu_entry_
↪area mapping
 ffffe8000000000 |   -1.5  TB | ffffffffffffffff |  0.5 TB | ... unused␣
↪hole
 ffffff0000000000 |   -1    TB | ffffff7fffffffff |  0.5 TB | %esp fixup␣
↪stacks
 ffffff8000000000 | -512    GB | ffffffeeffffffff |  444 GB | ... unused␣
↪hole
 ffffffef00000000 |  -68    GB | fffffffeffffffff |   64 GB | EFI region␣
↪mapping space
 fffffffff0000000 |   -4    GB | fffffffff7ffffff |    2 GB | ... unused␣
↪hole
 ffffffff80000000 |   -2    GB | ffffffff9fffffff |  512 MB | kernel text␣
↪mapping, mapped to physical address 0
 ffffffff80000000 |-2048    MB |                  |         |
 ffffffffa0000000 |-1536    MB | fffffffffeffffff | 1520 MB | module␣
↪mapping space
 ffffffffff000000 |  -16    MB |                  |         |
    FIXADDR_START | ~-11    MB | ffffffffff5fffff | ~0.5 MB | kernel-
↪internal fixmap range, variable size and offset
 ffffffffff600000 |  -10    MB | ffffffffff600fff |    4 kB | legacy␣
↪vsyscall ABI
 ffffffffffe00000 |   -2    MB | ffffffffffffffff |    2 MB | ... unused␣
↪hole
__________________|____________|__________________|_________|_______________
↪_______________________________________________
```

22.3.2 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    |   Offset   |     End addr     |  Size   | VM area␣
↪description
=============================================================================================
                  |            |                  |         |
 0000000000000000 |    0       | 00ffffffffffffff |   64 PB | user-space␣
↪virtual memory, different per mm
__________________|____________|__________________|_________|_______________
↪_______________________________________________
                  |            |                  |         |
 0100000000000000 |  +64    PB | feffffffffffffff | ~16K PB | ... huge,␣
↪still almost 64 bits wide hole of non-canonical
                  |            |                  |         |     virtual␣
↪memory addresses up to the -64 PB
```

(continues on next page)

(continued from previous page)

```
                  |            |                  |          |     starting␣
↪offset of kernel mappings.
__________________|____________|__________________|_________|_____________
↪_____________________________________________
                                                              |
                                                              | Kernel-space␣
↪virtual memory, shared between all processes:
______________________________________________________________|_____________
↪_____________________________________________
                  |            |                  |         |
 ff00000000000000 |  -64    PB | ff0fffffffffffff |    4 PB | ... guard␣
↪hole, also reserved for hypervisor
 ff10000000000000 |  -60    PB | ff10ffffffffffff | 0.25 PB | LDT remap␣
↪for PTI
 ff11000000000000 |  -59.75 PB | ff90ffffffffffff |   32 PB | direct␣
↪mapping of all physical memory (page_offset_base)
 ff91000000000000 |  -27.75 PB | ff9fffffffffffff | 3.75 PB | ... unused␣
↪hole
 ffa0000000000000 |  -24    PB | ffd1ffffffffffff | 12.5 PB | vmalloc/
↪ioremap space (vmalloc_base)
 ffd2000000000000 |  -11.5  PB | ffd3ffffffffffff |  0.5 PB | ... unused␣
↪hole
 ffd4000000000000 |  -11    PB | ffd5ffffffffffff |  0.5 PB | virtual␣
↪memory map (vmemmap_base)
 ffd6000000000000 |  -10.5  PB | ffdeffffffffffff | 2.25 PB | ... unused␣
↪hole
 ffdf000000000000 |   -8.25 PB | fffffbffffffffff |   ~8 PB | KASAN shadow␣
↪memory
__________________|____________|__________________|_________|_____________
↪_____________________________________________
                                                              |
                                                              | Identical␣
↪layout to the 47-bit one from here on:
______________________________________________________________|_____________
↪_____________________________________________
                  |            |                  |         |
 fffffc0000000000 |   -4    TB | fffffdffffffffff |    2 TB | ... unused␣
↪hole
                  |            |                  |         | vaddr_end␣
↪for KASLR
 fffffe0000000000 |   -2    TB | fffffe7fffffffff |  0.5 TB | cpu_entry_
↪area mapping
 fffffe8000000000 |   -1.5  TB | fffffeffffffffff |  0.5 TB | ... unused␣
↪hole
 ffffff0000000000 |   -1    TB | ffffff7fffffffff |  0.5 TB | %esp fixup␣
↪stacks
 ffffff8000000000 | -512    GB | ffffffeeffffffff |  444 GB | ... unused␣
↪hole
 ffffffef00000000 |  -68    GB | fffffffeffffffff |   64 GB | EFI region␣
↪mapping space
 ffffffff00000000 |   -4    GB | ffffffff7fffffff |    2 GB | ... unused␣
↪hole
 ffffffff80000000 |   -2    GB | ffffffff9fffffff |  512 MB | kernel text␣
↪mapping, mapped to physical address 0
 ffffffff80000000 |-2048    MB |                  |         |
 ffffffffa0000000 |-1536    MB | fffffffffeffffff | 1520 MB | module␣
↪mapping space
```

(continues on next page)

(continued from previous page)

```
 ffffffffff000000 |  -16    MB |                  |         |
    FIXADDR_START | ~-11    MB | ffffffffff5fffff | ~0.5 MB | kernel-
↪internal fixmap range, variable size and offset
 ffffffffff600000 |  -10    MB | ffffffffff600fff |    4 kB | legacy␣
↪vsyscall ABI
 ffffffffffe00000 |   -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).

vmalloc space is lazily synchronized into the different PML4/PML5 pages of the processes using the page fault handler, with init_top_pgt as reference.

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: ffffffffffff4111

22.4 5-level paging

22.4.1 Overview

Original x86-64 was limited by 4-level paing to 256 TiB of virtual address space and 64 TiB of physical address space. We are already bumping into this limit: some vendors offers 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 Documentation/x86/x86_64/mm.rst

22.4.2 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.

22.4.3 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.

22.5 Fake NUMA For CPUSets

Author David Rientjes <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 Documentation/x86/x86_64/boot-options.rst.

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 cpuset 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 cpuset, '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.

22.6 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.

22.7 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:

bankNctl (N bank number)

> 64bit Hex bitmask enabling/disabling specific subevents for bank N When a bit in the bitmask is zero then the respective subevent will not be reported. By default all events are enabled. Note that BIOS maintain another mask to disable specific events per bank. This is not visible here

The following entries appear for each CPU, but they are truly shared between all CPUs.

check_interval How often to poll for corrected machine check errors, in seconds (Note output is hexadecimal). Default 5 minutes. When the poller finds MCEs it triggers an exponential speedup (poll more often) on the polling interval. When the poller stops finding MCEs, it triggers an exponential backoff (poll less often) on the polling interval. The check_interval variable is both the initial and maximum polling interval. 0 means no polling for corrected machine check errors (but some corrected errors might be still reported in other ways)

tolerant Tolerance level. When a machine check exception occurs for a non corrected machine check the kernel can take different actions. Since machine check exceptions can happen any time it is sometimes risky for the kernel to kill a process because it defies normal kernel locking rules. The tolerance level configures how hard the kernel tries to recover even at some risk of deadlock. Higher tolerant values trade potentially better uptime with the risk of a crash or even corruption (for tolerant >= 3).

0: always panic on uncorrected errors, log corrected errors 1: panic or SIGBUS on uncorrected errors, log corrected errors 2: SIGBUS or log uncorrected errors, log corrected errors 3: never panic or SIGBUS, log all errors (for testing only)

Default: 1

Note this only makes a difference if the CPU allows recovery from a machine check exception. Current x86 CPUs generally do not.

trigger Program to run when a machine check event is detected. This is an alternative to running mcelog regularly from cron and allows to detect events faster.

monarch_timeout How long to wait for the other CPUs to machine check too on a exception. 0 to disable waiting for other CPUs. Unit: us

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 see http://one.firstfloor.org/~andi/mce.pdf