



Paging



Table of contents

- Memory paging (Intel 80x86 architecture)
- Translation Lookaside Buffer (TLB)
- Paging in Linux: page directory and page tables
- Booting the kernel
- Process descriptor
- Process stack in kernel mode and `thread_info`
- Process address space – introduction
- Process address space – KASLR, KAISER (KPTI)



Memory paging (Intel 80x86)

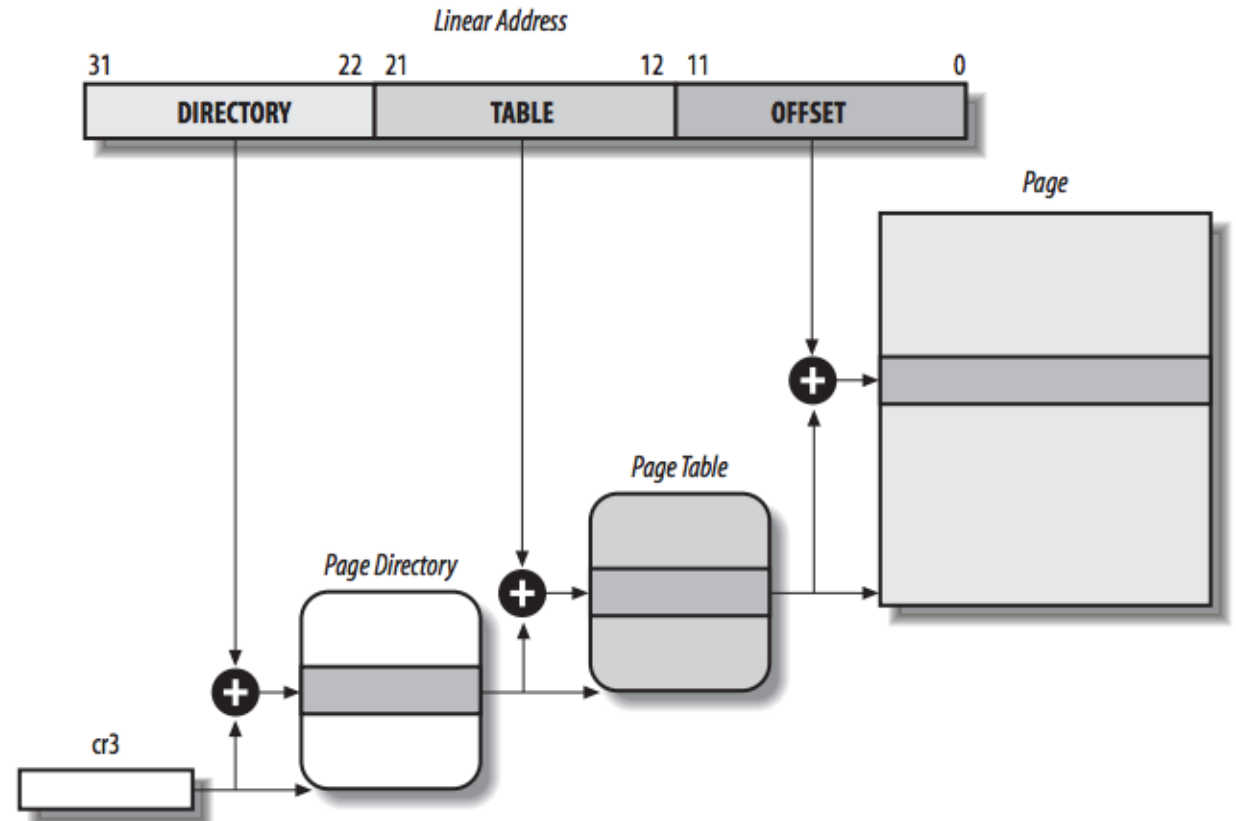
All 80x86 processors, starting from 80386, support **paging**. It is turned on by setting the **PG flag** of the **cr0** register – when the flag is zero, linear addresses are interpreted as physical addresses.

At the paging stage, the **linear address** is translated into **physical address**.

The **page directory address** is available in the **cr3** registry.

During the translation, access rights are checked and if they are not sufficient, a **page fault** is raised.

The standard page size for 32-bit architectures is 4 KB.



Bovet, Cesati, Understanding the Linux Kernel)



Memory paging (Intel 80x86)

The purpose of **multi-level paging** is to **reduce the amount of RAM** used on the process page tables.

In the case of **single-level paging**, the table would have a 2^{20} positions (if the process used a full 4 GB linear address space, even if it did not use all addresses in this range).

In the case of **multi-level paging**, the page tables are allocated only to those addresses that are used. Each active process needs a **directory of pages**, while **page tables** are allocated as needed.

The number of paging levels does not affect the **size** of the available address space, having **n bits** one can address 2^n memory cells, regardless of the number of paging levels used.

Starting with the Pentium model, 80x86 microprocessors support the **extended paging** mechanism, which allows **4 MB pages** (instead of 4 KB) – it is enabled by setting the **Page Size flag** in **page directory entry**.

Extended paging coexists with **regular paging**; current state is described by **PSE (Page Size Extension)** flag of the **cr4** register.

Intel's 80386 to Pentium processors used **32-bit physical addresses**. Theoretically, it allowed to install **4 GB of RAM**, in practice the **kernel** could not address directly more than **1 GB of RAM** (this will be explained in detail later).



Memory paging (Intel 80x86)

Starting with Pentium Pro, all Intel processors can address $2^{36}=64$ GB RAM. The width of the **address bus** increased to **36**. Additional memory requires special paging mechanism that translates **32-bit linear addresses** into **36-bit physical addresses**. The **PAE** (**Page Address Extension**) mechanism is used for this (**PAE** flag in **cr4**). The linear address still has only 32 bits, so system developers must use the same linear addresses to map different RAM areas.

Linux supports **64-bit architectures**, but they require **more paging levels** (3, 4, 5).

In **multiprocessor systems**, all **processors** usually **share memory**, i.e. they can reach the same blocks of RAM. Read and write operations on one RAM block must be executed sequentially, a hardware element called **memory arbiter**, located between the bus and the RAM block, takes care of this.

The arbiter of memory is also available in **single-processor systems**, because RAM is accessed not only by **the processor**, but also by the **DMA** (*Direct Memory Access*) **drivers**.



Translation Lookaside Buffer (TLB)

An important role during the translation of linear addresses is performed by **TLB associative registers** (**Translation Lookaside Buffer**) (well described in [Wikipedia](https://en.wikipedia.org/wiki/Translation_Lookaside_Buffer)).

In **multiprocessor systems**, each **processor has its own TLB**.

There may be separate TLBs for **instructions** and **data**.

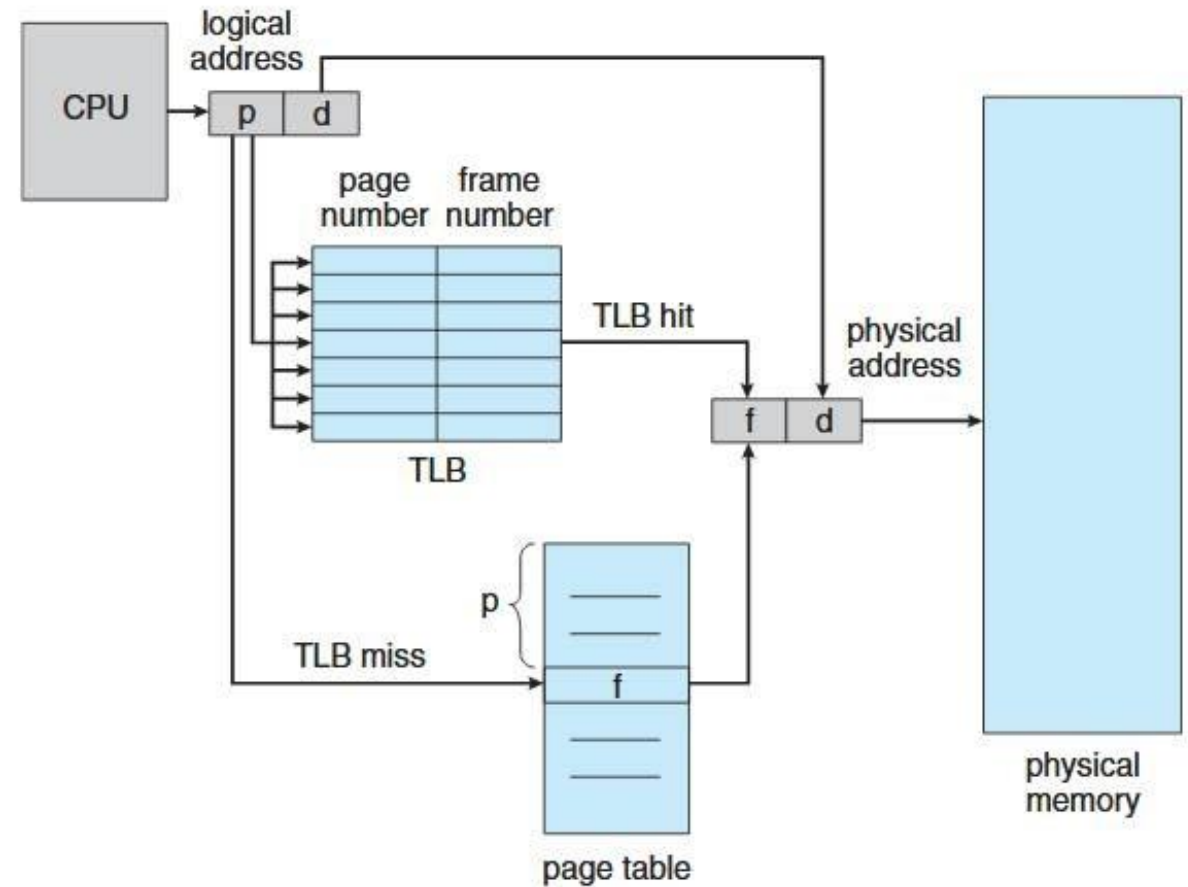
Small:

- 64 entries, 4-way (4KB pages), cover 256KB of data
- 32 entries, 4-way (2MB pages), cover 64MB of data

(page table has 1M entries)

Fast:

- PTE in the TLB: < **1 cycle** to translate
- PTE not in the TLB: **10-100 cycles** to load PTE from RAM
- Page not in RAM: **~80M cycles** to get it from disk





TLB miss

TLB makes virtual memory practical:

it is **small**, it can be built **directly** into the **CPU**, it runs at **full CPU speed**. As long as a translation can be found in the TLB, a virtual access executes just **as fast as a physical access**.

TLB miss, depending on the CPU architecture, is handled in one of two ways:

- **Hardware TLB miss handling**: CPU goes ahead and walks the page table to find the right **PTE**. If the PTE can be found and is marked present, the **CPU installs the new translation in the TLB**. Otherwise, the CPU raises a **page fault** and hands over control to the **operating system**.
- **Software TLB miss handling**: CPU raises a **TLB miss fault**. The fault is intercepted by the operating system, which invokes the **TLB miss handler**. The miss handler walks the page table in software and, if a matching PTE that is marked present is found, the new translation is inserted in the TLB. If the PTE is not found, control is handed over to the **page fault handler**.

Most CISC architectures (such as IA-32) perform TLB miss handling in **hardware**, and most RISC architectures (such as Alpha) use a **software** approach.

A hardware solution is often **faster**, but is **less flexible**.

IA-64 provides a **hybrid** solution that retains much of the flexibility of the software approach without sacrificing the speed of the hardware approach.



TLB flush

On an **address space switch**, as occurs on a process switch (but not on a thread switch), some TLB entries can become **invalid**, since the **virtual-to-physical mapping is different**.

Options:

1. **Completely flush the TLB** – after a switch, the TLB is **empty** and any memory reference will be a miss.
2. **Avoid flushing** – some CPUs have a process ID register, and the hardware uses TLB entries only if they match the current process ID.
 1. Intel Pentium Pro: the **Page Global Enable (PGE) flag** in the register **CR4** and the **global (G) flag** of a **page directory** or **page table entry** can be used to prevent frequently used pages from being automatically invalidated.
 2. Since 2010 **Intel 64 processors** support 12-bit **PCIDs** (*process-context identifiers*), which allow retaining TLB entries for multiple linear-address spaces, with only those that match the current PCID being used for address translation.
3. **Selective flushing** – is an option in software managed TLBs, but in some hardware TLBs (e.g. the TLB in the Intel 80386) the only option is the complete flushing. **Other hardware** TLBs (TLB in the Intel 80486 and later x86 processors, TLB in ARM processors) allow the flushing of **individual entries** from the TLB indexed by virtual address.

[Wikipedia - Address space switch](#)



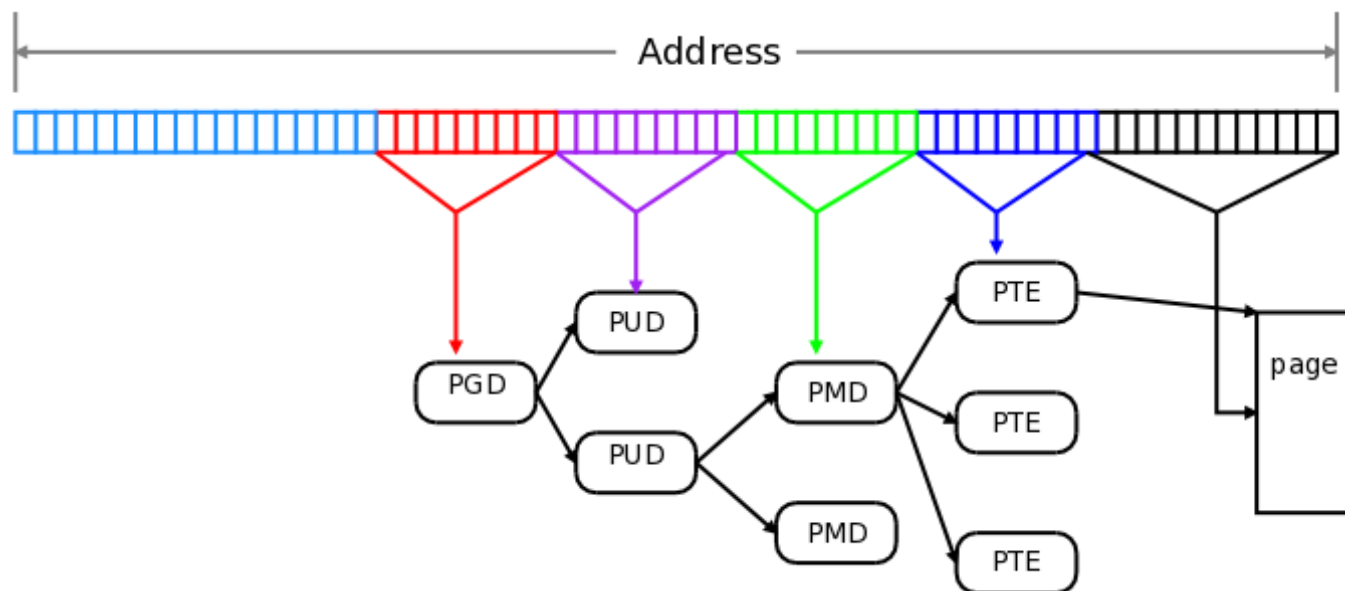
Paging in Linux

Page directories and page tables



Different architectures require a different number of paging levels. Beginning with the **2.6.11** kernel version, Linux supports **4 levels of paging**.

[Four-level page tables merged](#) (Jonathan Corbet, January 2005).

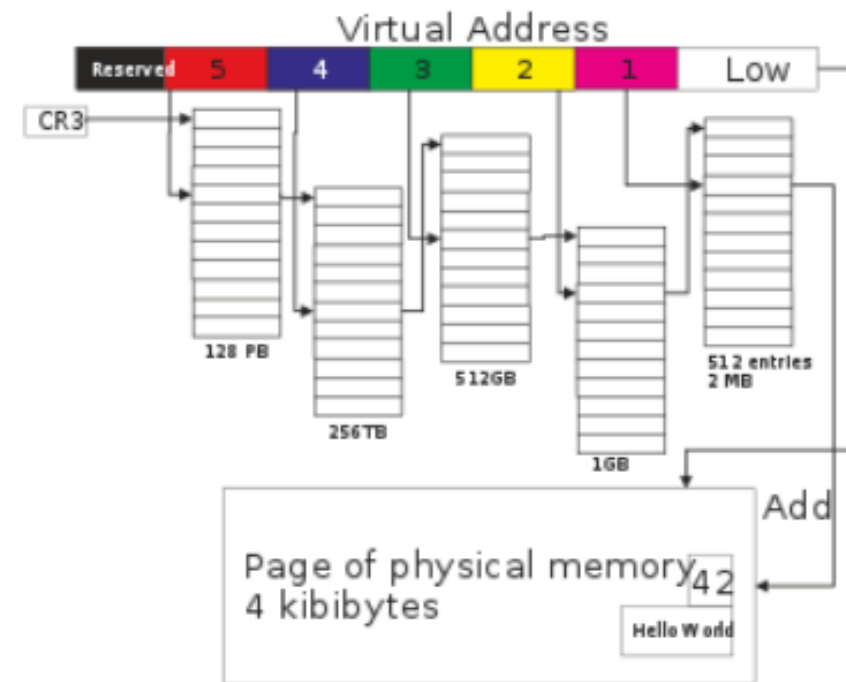


[Lwn.net](#) (uppermost 16 bits out of 64 are discarded)
2 MB huge pages

Beginning with **4.12** Linux supports **5 levels of paging**.

[Five-level page tables](#) (Jonathan Corbet, March 2017) – a new level P4D is inserted between PGD and PUD, 7 bits out of 64 are discarded

[Intel 5-level paging](#)





Paging in Linux

Page directories and page tables

For 32-bit architectures without PAE, two paging levels are sufficient. In that case Linux does not use **Page Upper Directory** and **Page Middle Directory** by setting for them the number of bits to 0 and the number of positions to 1.

Each process has its own **page directory** and its own set of **page tables**. During context switch, the kernel saves the contents of the **cr3 register** in the **task struct** of the previously running process and loads to cr3 the address of the page directory stored in the task struct of the process to which control will be handed over. When the process **resumes execution**, it has access to its **own set of pages**.

The entries in page directories and page tables have the same structure. Each of them contains the **address of the frame with another table or page** (if they are resident in memory). Because the frame address is a multiple of 4 KB, the youngest **12 bits are always zero**. Linux uses them to store **additional information about the page**.

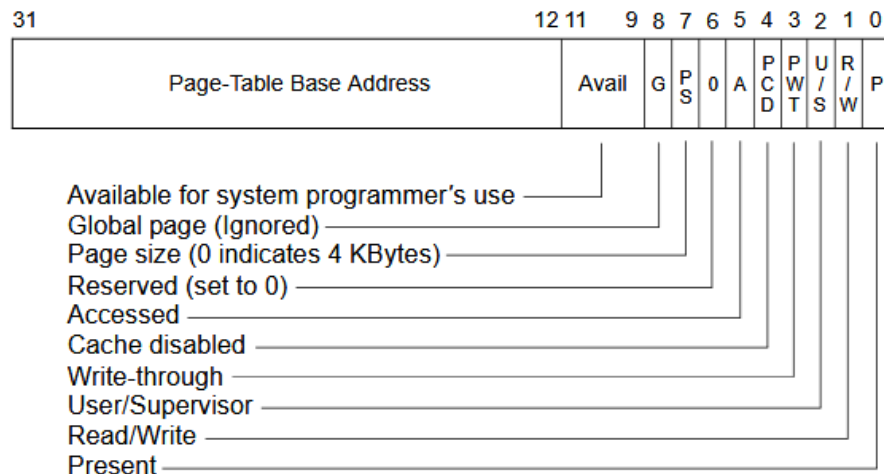
The bits of the page's information are marked in large letters. They have exactly the same names and layout as the hardware-supported bits in CPUs from the 80x86 family (there may be more flags).



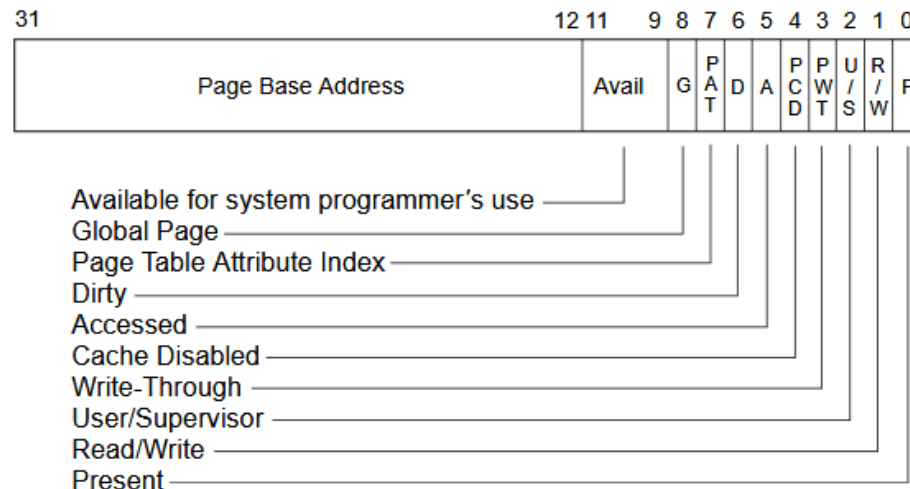
Paging in Linux

Page directories and page tables

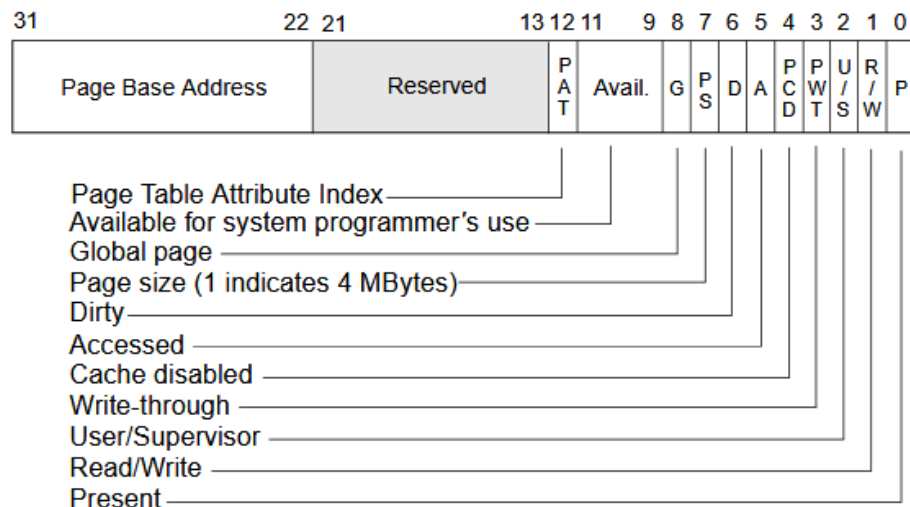
Page-Directory Entry (4-KByte Page Table)



Page-Table Entry (4-KByte Page)



Page-Directory Entry (4-MByte Page)



Protected-mode memory management in x86
(source: [Intel](https://www.intel.com/content/www/us/en/processors/x86/x86-64/architecture-manuals/protected-mode-memory-management-in-x86-64-architecture-manuals.html))



Paging in Linux

Page directories and page tables

PRESENT If this bit is **1** on a memory reference, the page is **in memory**. The **address** field contains the correct frame address in physical memory. If this bit is zero, the **linear address** is inserted into the **cr2 register** and a **page fault** error is raised. All other bits have a different meaning (they determine the position of the page on the swap device).

RW Bit set to 1 means **permission to write** on a given page. This bit is used to organize page sharing. Set to 1, allows to write directly to the page, and when is zero, then a page fault error is raised when writing.

USER/SUPERVISOR Bit setting the **security level** when accessing items in a page table or page. If set to **zero**, only the **kernel** can reach the page.

PCD (Page Cache Disabled) i **PWT (Page Write-Through)** Flags controlling the handling of a page or page table by the **hardware cache** of the processor (all of the cache controls can only be used to **reduce caching**). See [Write-back vs Write-through](#).

ACCESSED Linux assumes that this bit is **set by hardware to 1** when page is **referenced** and uses this fact to **age** the pages (the operating system changes the value of the flag to 0, it is not done by hardware).

PAGE SIZE Applies only to entries in the **page directory**. If set, it means that the page has a size of **4 MB** (page directory entry points directly to a 4 MB page), otherwise the size is **4 KB**.

DIRTY Applies only to items in the **page table**. If set, indicates that page has been written to. It is set by hardware (CPU) to 1, but has to be cleared by the operating system. Used in **page replacement** algorithms.

G (Global) Applies only to items in the **page table**. The Global, or 'G' above, flag, if set, prevents the TLB from updating the address in its cache if CR3 is reset. Note, that the **page global enable bit in CR4** must be set to enable this feature.



Paging in Linux

Page directories and page tables

The **entire entry of the page table can be set to 0**. Linux assigns a physical frame to the process only when it is **referenced**. Zero is the initial value to which entries in the page directory and page tables are usually set.

By allocating a new memory for the **process**, the kernel does so only **logically** – through entries in the page table and segment table.

For the **kernel**, memory is allocated **immediately** – a frame is found in physical memory.

The system does not believe the user and tries to optimize his possible excessive requests.

Linux has functions to **read** and **write** each **flag** in **PTE**.



Paging in Linux

Page directories and page tables

Useful macros:

- **PAGE_SHIFT** – number of offset bits (12 for a 4 KB page)
- **PAGE_SIZE** – page size, calculated using PAGE_SHIFT
- **PAGE_MASK** – mask for zeroing all bits of the offset field (0xfffff000 for a 4 KB page)

```
#define PAGE_SIZE (1 << PAGE_SHIFT)
#define PAGE_MASK (~(PAGE_SIZE-1))
```

Types **pte_t**, **pmd_t**, **pud_t**, **pgd_t** – define the format of the entry in, respectively, **Page Table**, **Page Middle Directory**, **Page Upper Directory**, **Page Global Directory**.

Each process has its own page tables. Also, the kernel has page tables describing the **kernel address space**.

The virtual page is a **memory protection unit**, since all its bytes share the U/S and R/W flags. The same physical memory can be mapped by different page table entries, with different protection flags.

Nowhere in the PTE there are bits saying about the right to perform (**execute**). That's why classic paging in x86 allows you to **execute code located on the stack**, making it easier for hackers to use buffer overflow attacks.



Do kernel pages get swapped out?

Do kernel pages get swapped out?

Under normal circumstances **kernel pages** are **not swappable**, in fact, once detected (see the page fault handler source code), the kernel will explicitly crash itself.

In particular, page tables aren't swapped out.

That said, kernel does swap out kernel structures/memory/tasklists etc. during **software suspend** and **hibernation** operation.


And during the **resume phase** it will **restore back** the **kernel memory** from swap file.



Booting the kernel



BIOS, CMOS, UEFI



The diagram shows a motherboard with various components labeled. A callout line points from the text 'The BIOS is stored in the BIOS chip.' to a small chip on the motherboard. Another callout line points from the text 'The BIOS settings are retained in the CMOS chip.' to a small chip on the motherboard. A third callout line points from the text 'On modern motherboards, the CMOS chip is integrated with the RTC on the south bridge chipset.' to a small chip on the motherboard.

The **BIOS** is the firmware or program that comes with your motherboard.

The BIOS is stored in the **BIOS chip**.

The BIOS settings are retained in the **CMOS chip**.

On modern motherboards, the CMOS chip is integrated with the RTC on the south bridge chipset.

https://www.youtube.com/watch?v=LGz0lo_dh_I



Booting the kernel

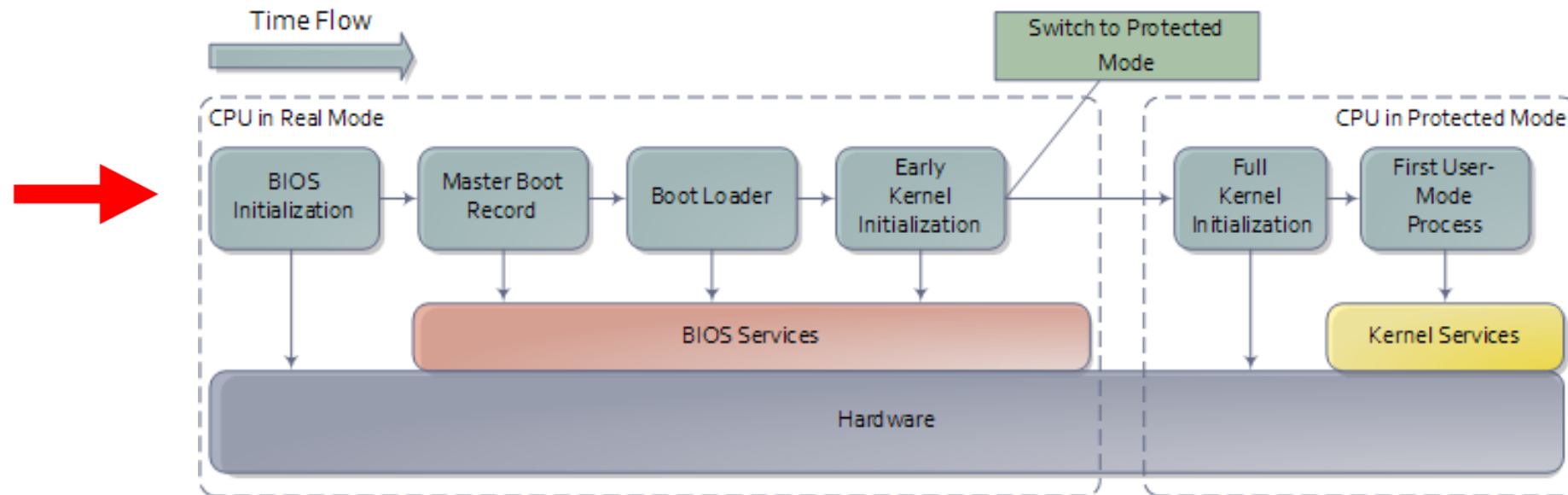
bootstrap – *pull oneself up by one's bootstraps*

Booting with **BIOS** is shown here, difference between **BIOS** and **UEFI** will be explained later.

You **press the power button** on the computer. Once the **motherboard** is powered up it **initializes** its own **firmware**. The **CPU starts running**.

In a multi-processor or multi-core system one CPU is dynamically chosen to be the **bootstrap processor (BSP)**, that runs all of the BIOS and kernel initialization code. The remaining processors (**AP, application processors**) remain halted until later on when they are explicitly activated by the kernel.

The processor is in **real mode** with **memory paging disabled** and there is **no protection**.





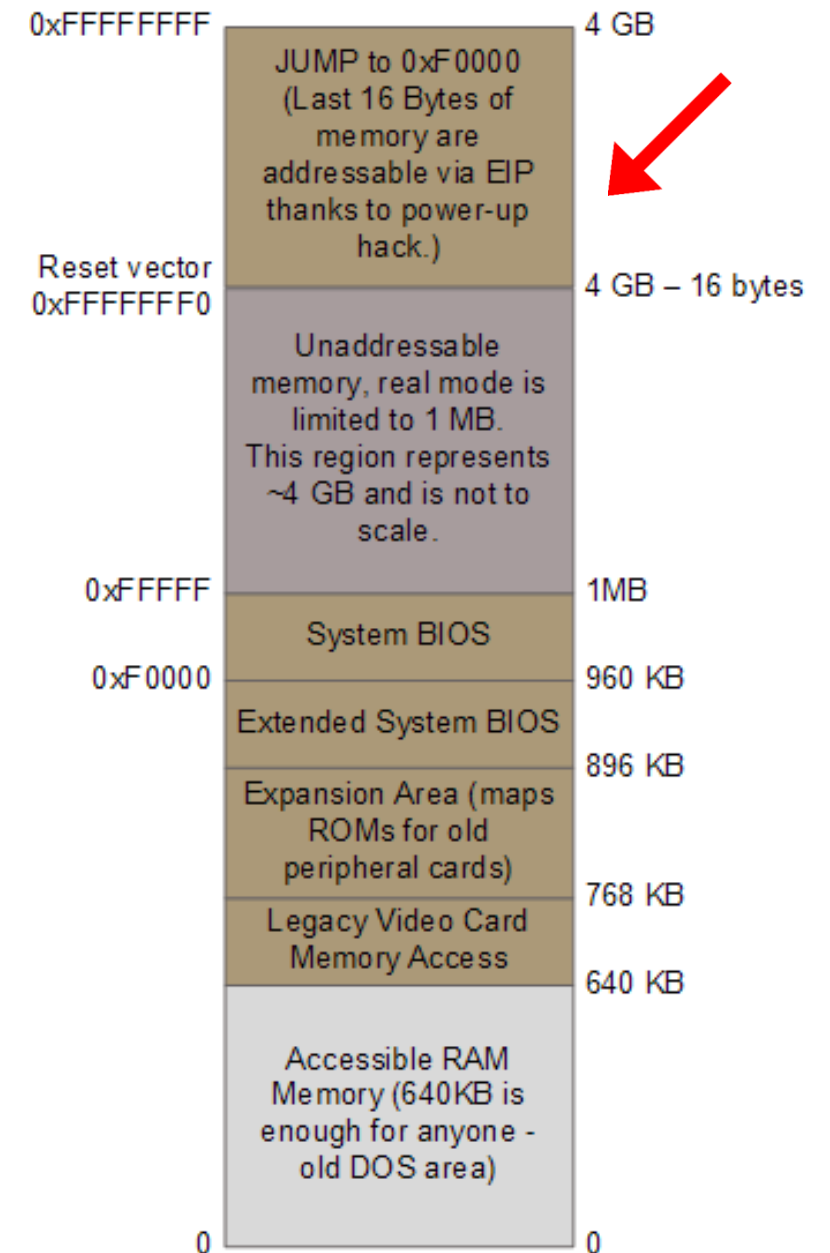
Booting the kernel

Most registers in the CPU have well-defined **values** after power up, including the **instruction pointer** (EIP).

Intel CPUs use a hack whereby even though only **1 MB** of memory can be addressed in **real mode**, a **hidden base address** (an offset) is applied to EIP so that the first instruction executed is at address 0xFFFFFFF0 (16 bytes short of the **end of 4 GB** of memory and well above one megabyte).

This magical address which is called the **reset vector**, contains a jump instruction to the **BIOS entry point** and is standard for modern Intel CPUs.

This jump implicitly **clears** the **hidden base** address.





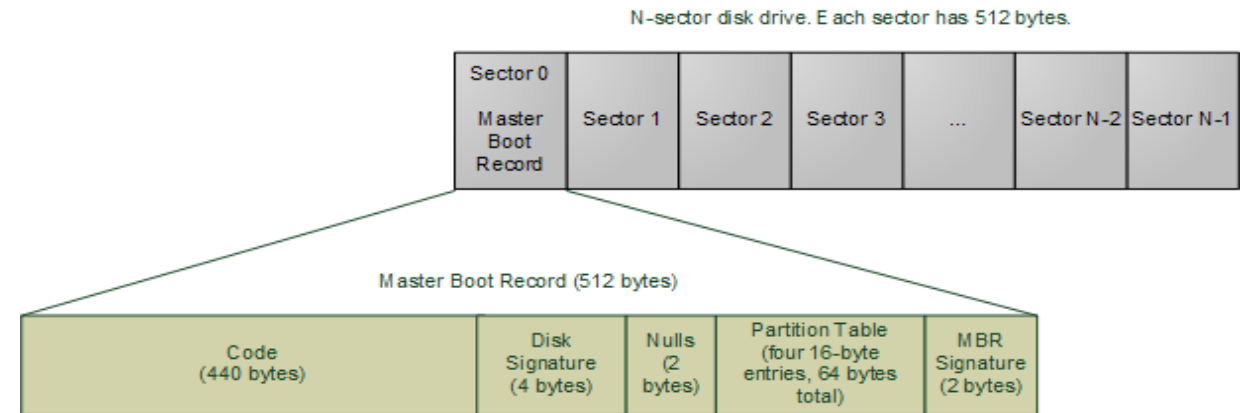
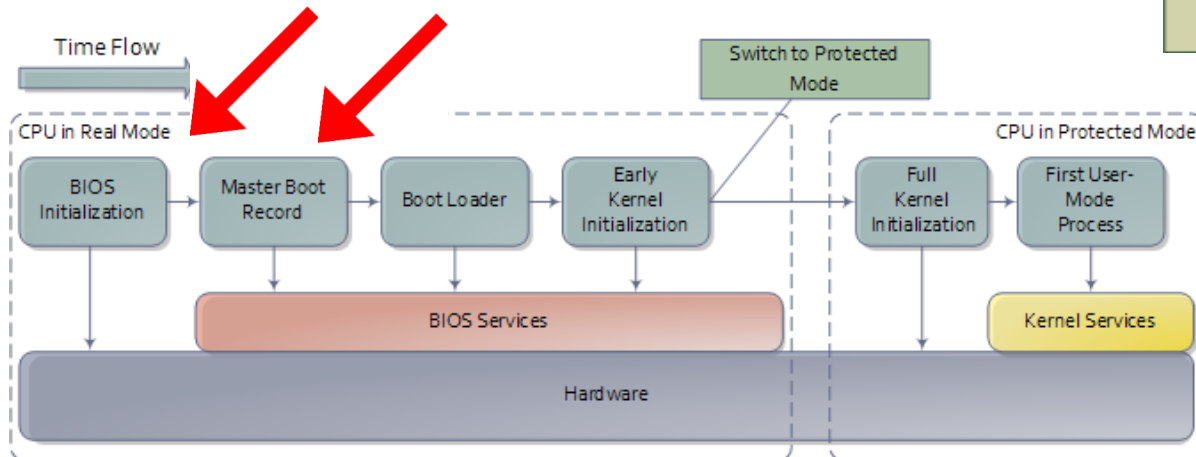
Booting the kernel

The CPU starts executing **BIOS code**, which initializes some of the hardware in the machine.

Afterwards the BIOS kicks off the **POST** (Power-on Self Test), which tests various components in the computer.

After the POST the BIOS wants to boot up an operating system – it reads the first 512-byte sector of the hard disk, (**Master Boot Record, MBR**), which normally contains a tiny OS-specific **bootstrapping program** at the start of the MBR followed by a **partition table** for the disk.

BIOS loads the contents of the MBR into memory location 0x7c00 and **jumps** to that location to **start executing** whatever code is in the **MBR**.



Master Boot Record (MBR) (source: Duarte, [Software Illustrated](#))



Booting the kernel

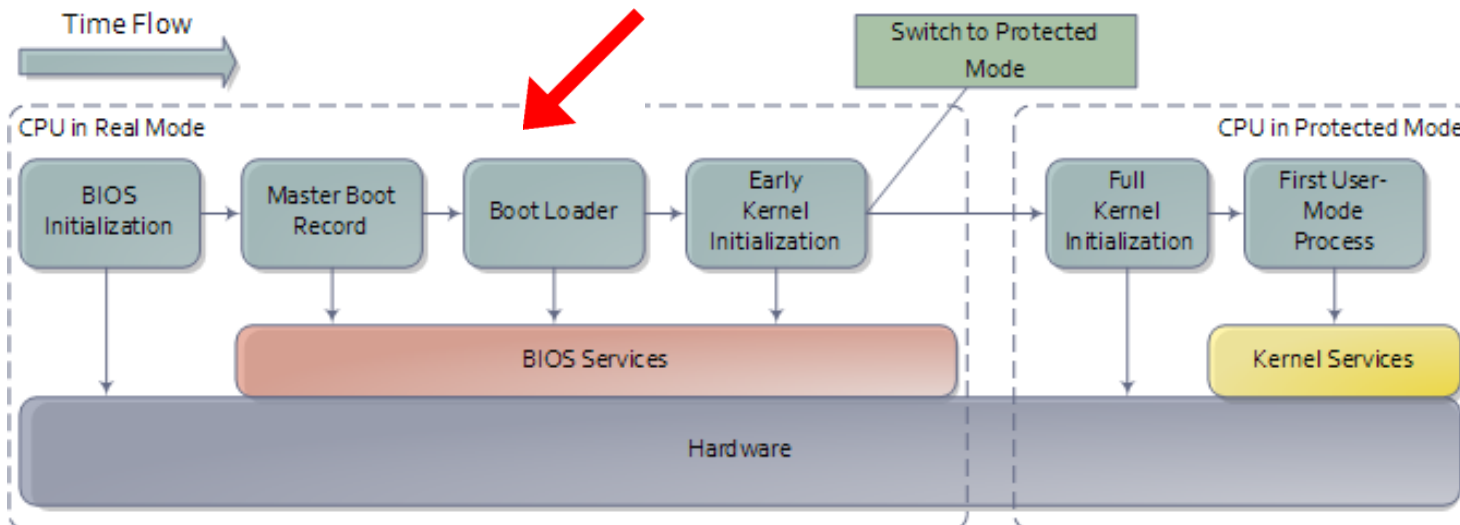
The specific code in the MBR could be **LILO** (old) or **GRUB/GRUB2** – Linux **loaders**.

This program will **load** the **kernel image** (e.g. vmlinuz-2.6.22-14-server) from the appropriate partition.

This involves some complications, as the kernel image, even if compressed, **will not fit in 640 KB RAM** available in **real mode**. Yet the **boot loader must run in real mode** in order to call the **BIOS routines** for reading from the disk, since the kernel is not available at that point.

The solution is the [unreal mode](#). This is not a true processor mode, but rather a technique where a program **switches** back and forth between **real** mode and **protected** mode in order to **access memory above 1 MB** while still using the **BIOS**.

In the GRUB source code you can see the instructions for it **real_to_prot** and **prot_to_real**.



When finished, the **kernel is loaded into memory** and the **processor is in real mode**.

In the diagram this corresponds to the situation just before moving from „Boot Loader” to „Early Kernel Initialization”.



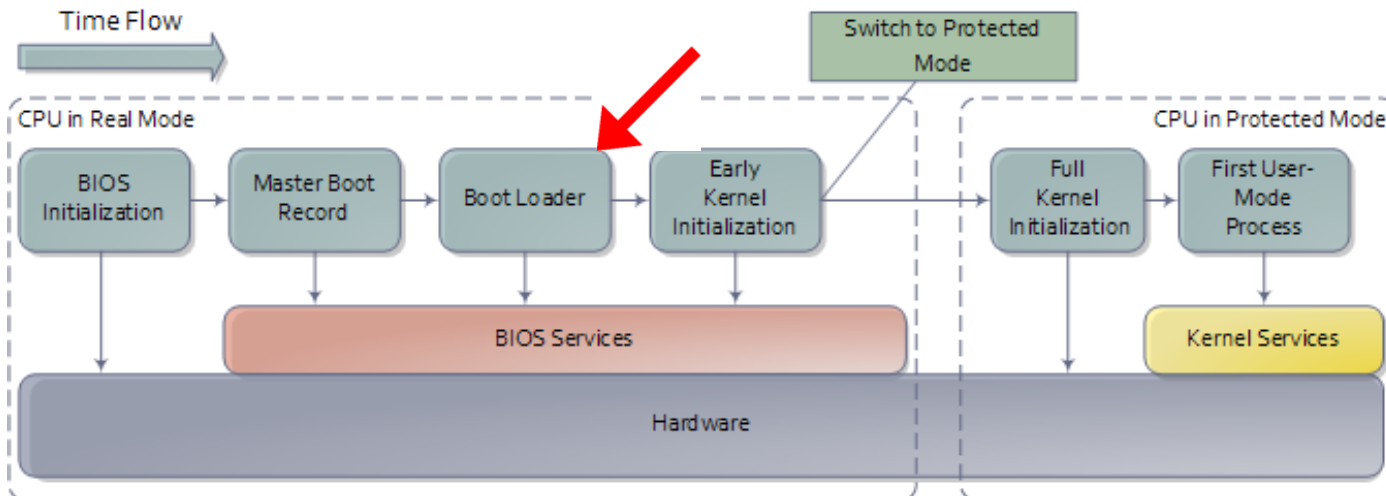
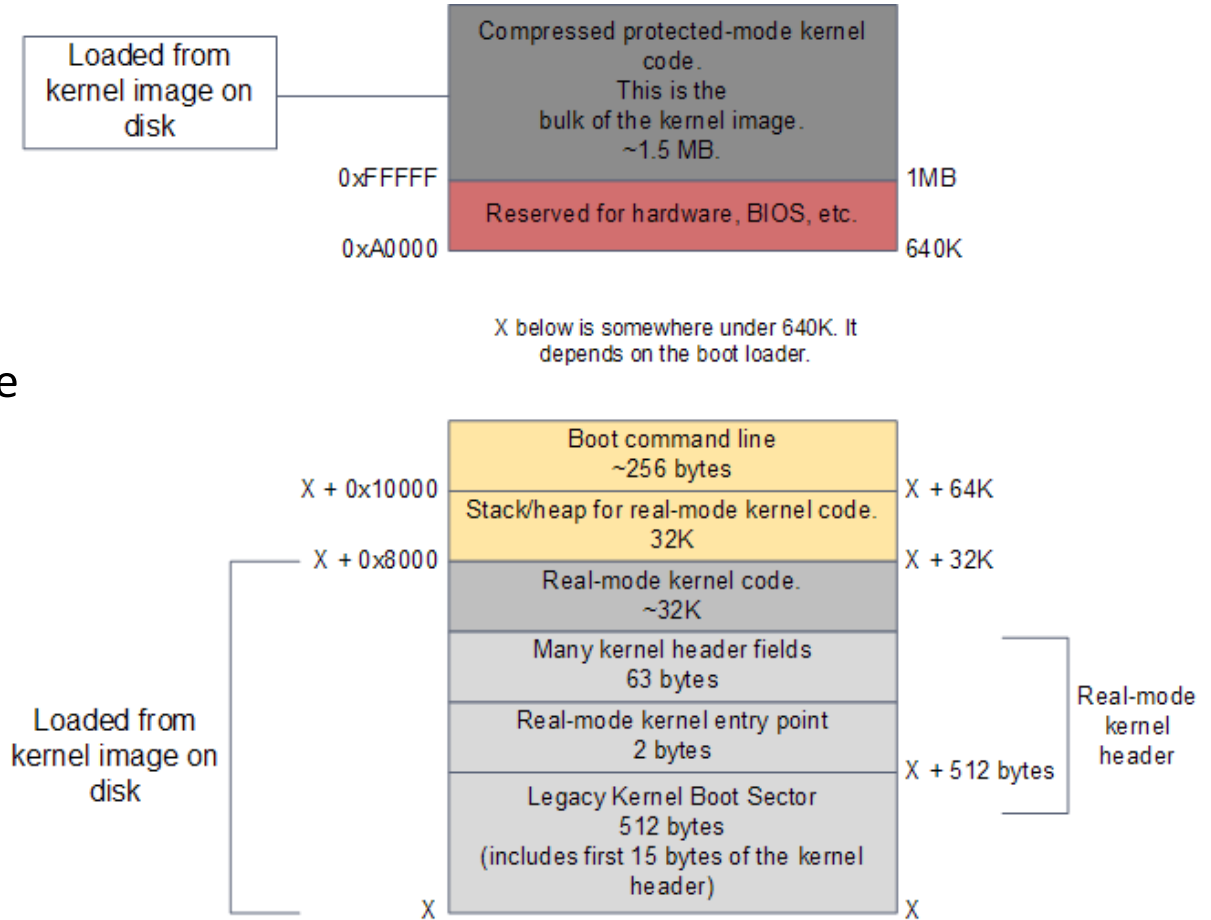
Booting the kernel

The processor is working in **real mode**, it can address 1 MB of memory, and RAM looks like in the picture.

In a moment, there will be a jump to the entry point of the kernel code.

The **kernel image** is split into two pieces:

- a small part containing the **real-mode kernel code** is loaded **below the 640 K** barrier;
- the bulk of the kernel, which runs in **protected mode**, is loaded **after the first megabyte of memory**.



RAM contents after boot loader is done
(source: Duarte, [Software Illustrated](#))



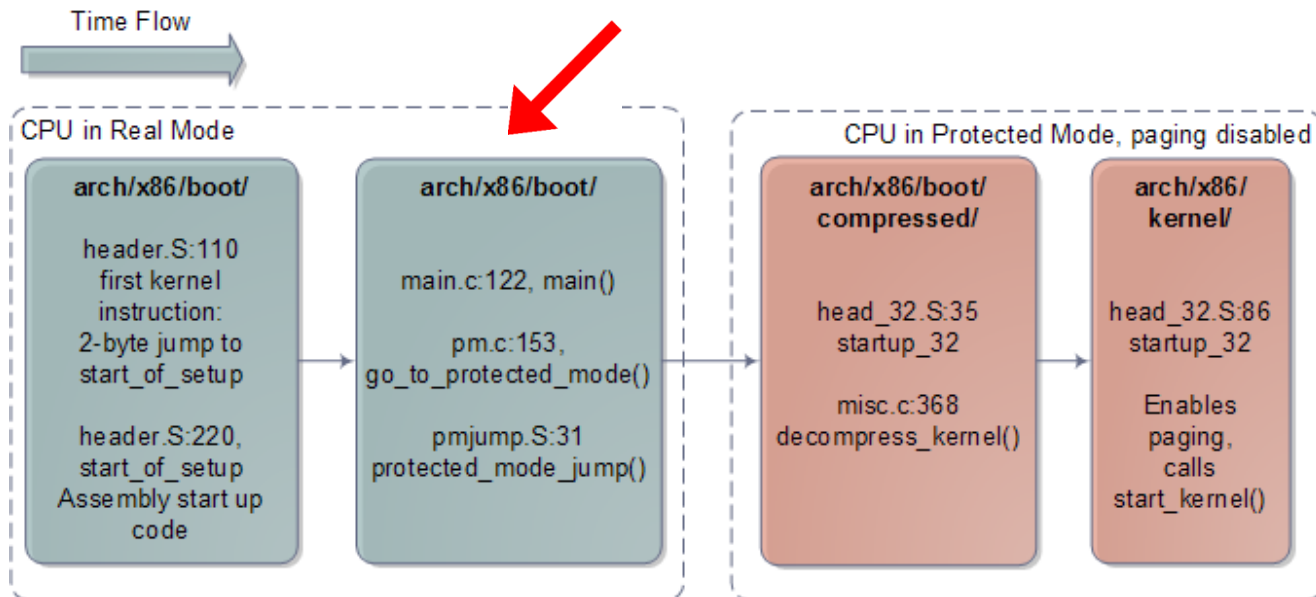
Booting the kernel

The execution starts in the part running in **real mode**.

Before the CPU can be set to **protected mode** the location of the **interrupt vector table** is stored in a CPU register **IDTR** (Interrupt Descriptor Table register), and the address of the **Global Descriptor Table** in a CPU register **GDTR** (routine **go_to_protected_mode** calls **setup_idt()** and **setup_gdt()**).

Jump into **protected mode** is done by the routine **protected_mode_jump()**, which enables protected mode by setting the **PE** (ang. **Protection Enabled**) bit in the CR0 (**control register**).

Paging is disabled. A processor can now **address up to 4 GB** of RAM.



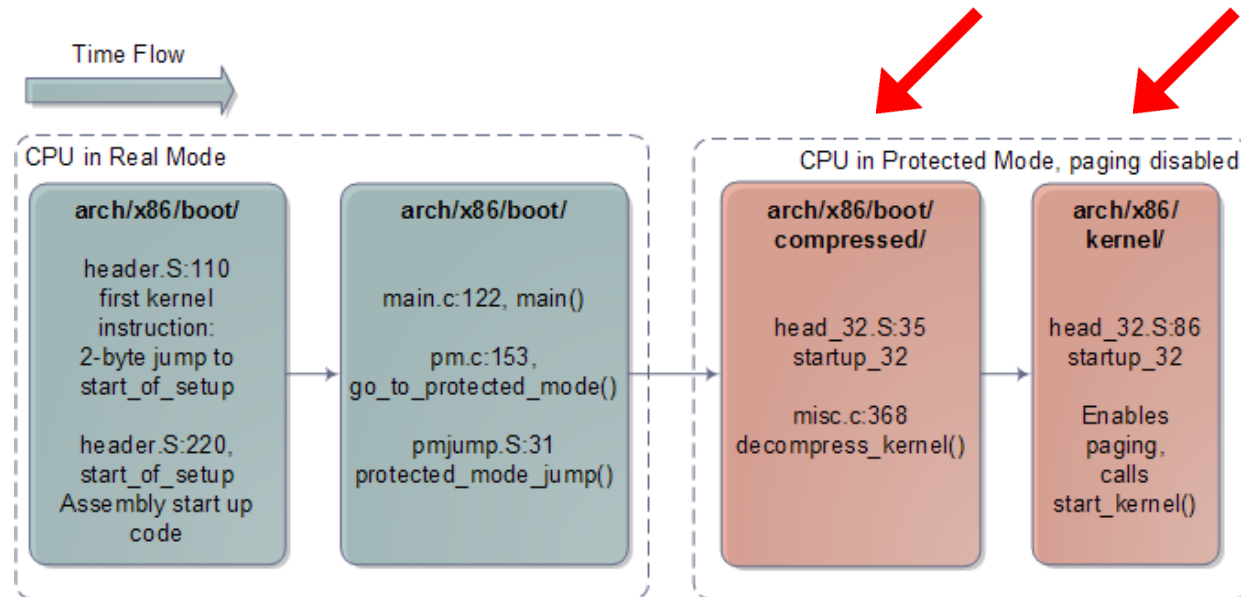


Booting the kernel

The routine then calls **startup_32**, which does some basic register initializations and calls **decompress_kernel()** (prints the familiar „Decompressing Linux...” message, then „done”, then „Booting the kernel.”).

A jump occurs at the kernel **entry point** in **protected mode**, at the start of the second megabyte of RAM (0x100000). This location contains (another) routine **startup_32**.

It clears the bss segment for the protected-mode kernel, sets up the final global descriptor table for memory, **builds page tables** so that paging can be turned on, **enables paging**, initializes a **stack**, creates the final interrupt descriptor table, and finally jumps to the **architecture-independent** kernel start-up **start_kernel()**.





Booting the kernel

Routine **start_kernel()** is a long list of calls to initializations of the various kernel subsystems and data structures. These include the **scheduler**, **memory zones**, **time keeping**, and so on.

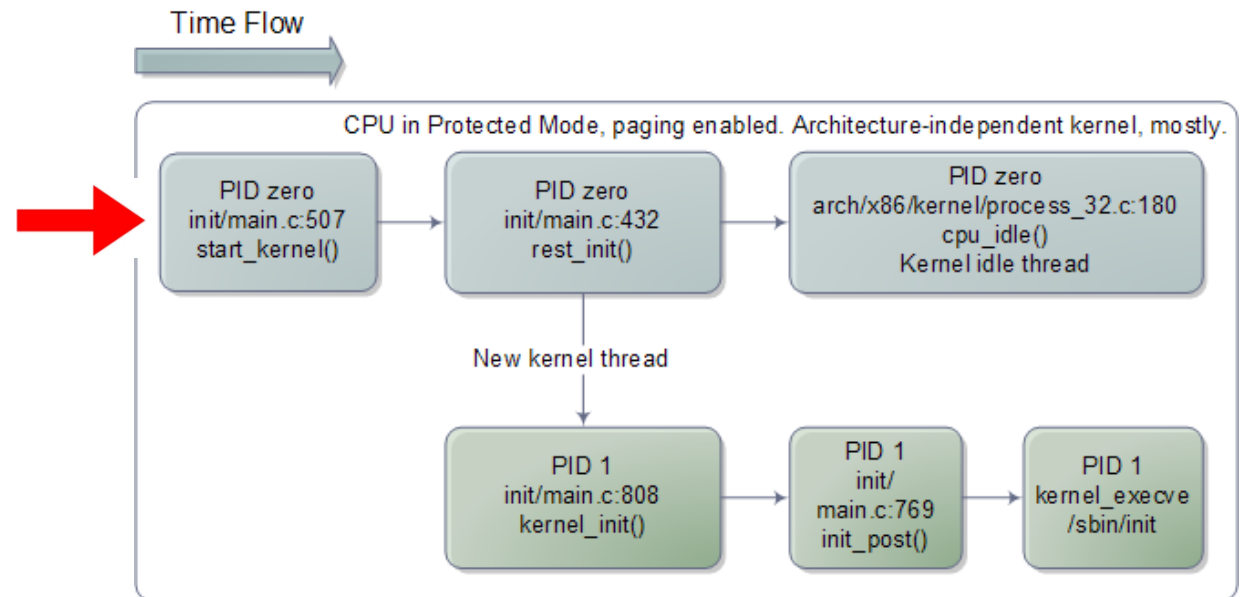
Then it calls **rest_init()**, which creates a kernel thread passing another routine **kernel_init()** as the entry point, then calls **schedule()** and goes to sleep by calling **cpu_idle()**.

Now **thread 1** starts, i.e. **kernel_init()**, which initiates the rest of the CPUs.

Finally **init_post()** is called, which tries to execute a **user-mode process** in the following order: `/sbin/init`, `/etc/init`, `/bin/init` i `/bin/sh`.

Thread 1 checks its configuration file to figure out which processes to launch, which might include X11 Windows, programs for logging in on the console, network daemons, and so on.


Thus ends the boot process.





BIOS vs UEFI

BIOS (OLD)




Phoenix - AwardBios CMOS Setup Utility

- Standard CMOS Features
- Advanced BIOS Features
- Advanced Chipset Features
- Integrated Peripherals
- Power Management Setup
- PnP/PCI Configurations
- PC Health Status
- Frequency/Voltage Control
- Load Fail - Safe Defaults
- Load Optimized Defaults
- Set Supervisor Password
- Set User Password
- Save & Exit Setup
- Exit Without Saving

Esc: Quit
F10: Save & Exit Setup

Time, Date, Hard Disk Type . . .

UEFI (NEW)



UEFI BIOS UTILITY : FULL MODE

12:35
TUESDAY 3/11/15

INTEL CORE i7 SPEED: 4.0 GZ
TOTAL MEMORY: 8 GB BUILD DATE: 7-2-15
BIOS VERSION: 235

POWERCERT

TEMPERATURE VOLTAGE FAN SPEED PERFORMANCE

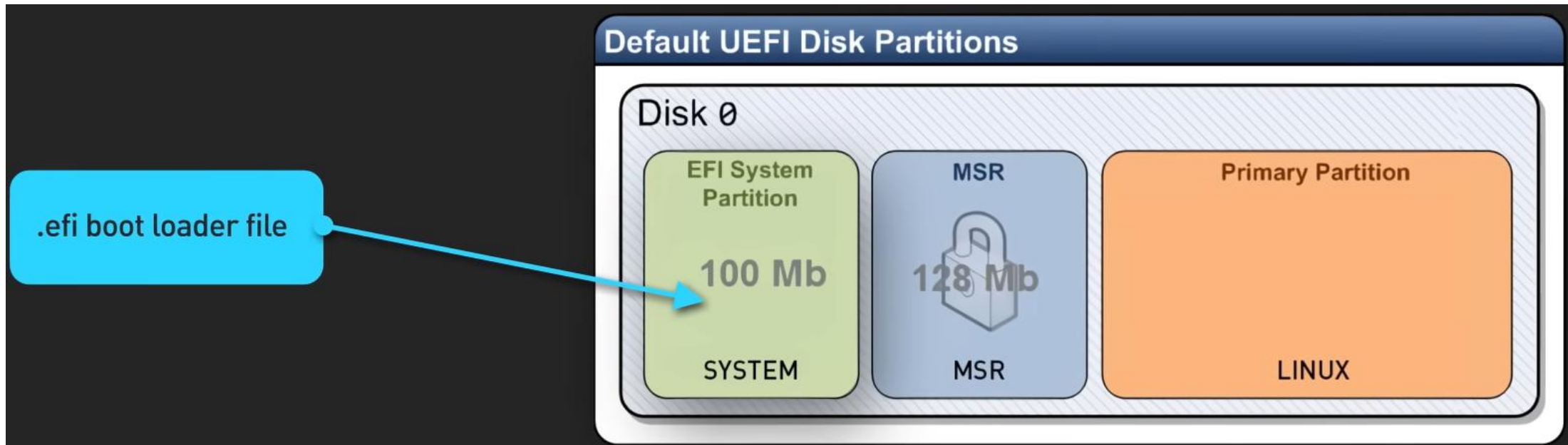
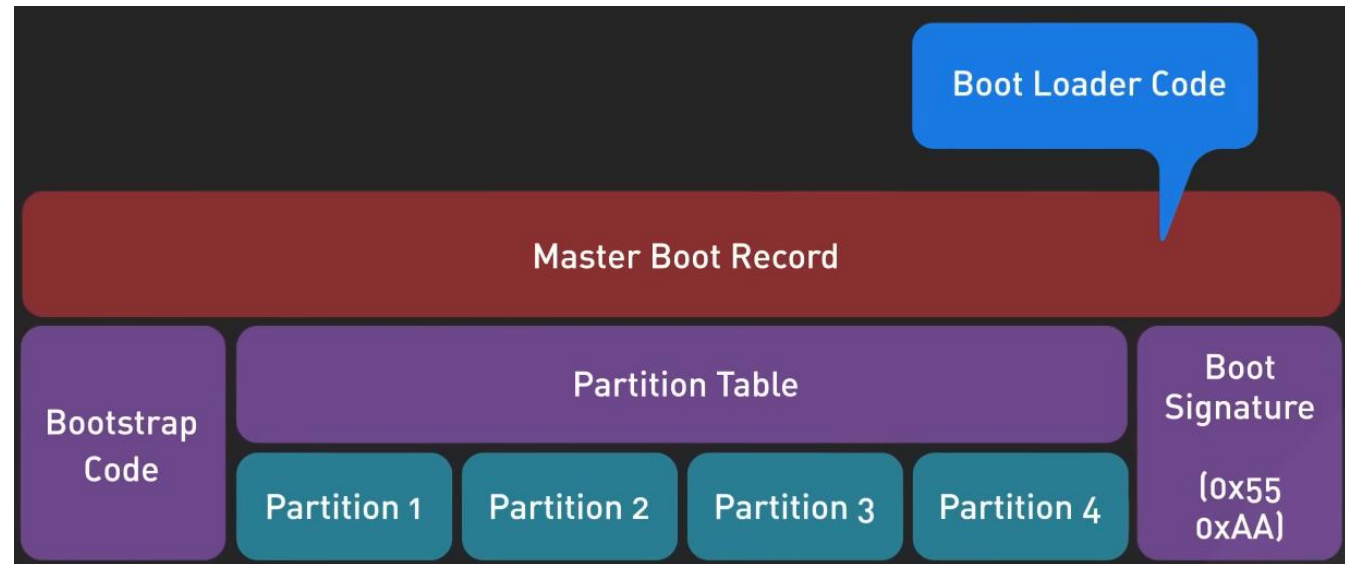
BOOT PRIORITY

HDD CD/DVD USB NETWORK

- Secure Boot – built in feature in UEFI that stops any digitally unsigned drivers from loading and helps to stop malicious software, such as rootkits.



BIOS vs UEFI





BIOS vs UEFI (Unified Extensible Firmware Interface)

UEFI stores all the information about **initialization** and **startup** in a .efi file, a file stored on a special partition called **EFI System Partition (ESP)**. The ESP partition will also contain the boot loader programs for the operating system installed on the computer.

It is because of this partition, UEFI can **directly boot** the operating system and save the BIOS self-test process, which is an important reason for UEFI faster booting.

| | BIOS | UEFI/EFI |
|--|--------------|----------------------------|
| Operating mode | 16 bit | 32/64 bit |
| Operating memory | 1 MB | Max available |
| Interface | Text | Graphical |
| Disk | MBR | GPT (GUID Partition Table) |
| Partition size | Up to 2,2 TB | Up to 10 billion TB |
| Number of partitions | Up to 4 | Up to 128 |
| Network access | No | Yes |
| Mode | Real | Protected |
| Secure Boot (DRM – Digital Rights Mngnt) | No | Yes |



Process data structures

- Process descriptor
- Process stack in kernel mode and thread_info
- Process address space – introduction
- Process address space – KASLR, KAISER (KPTI)



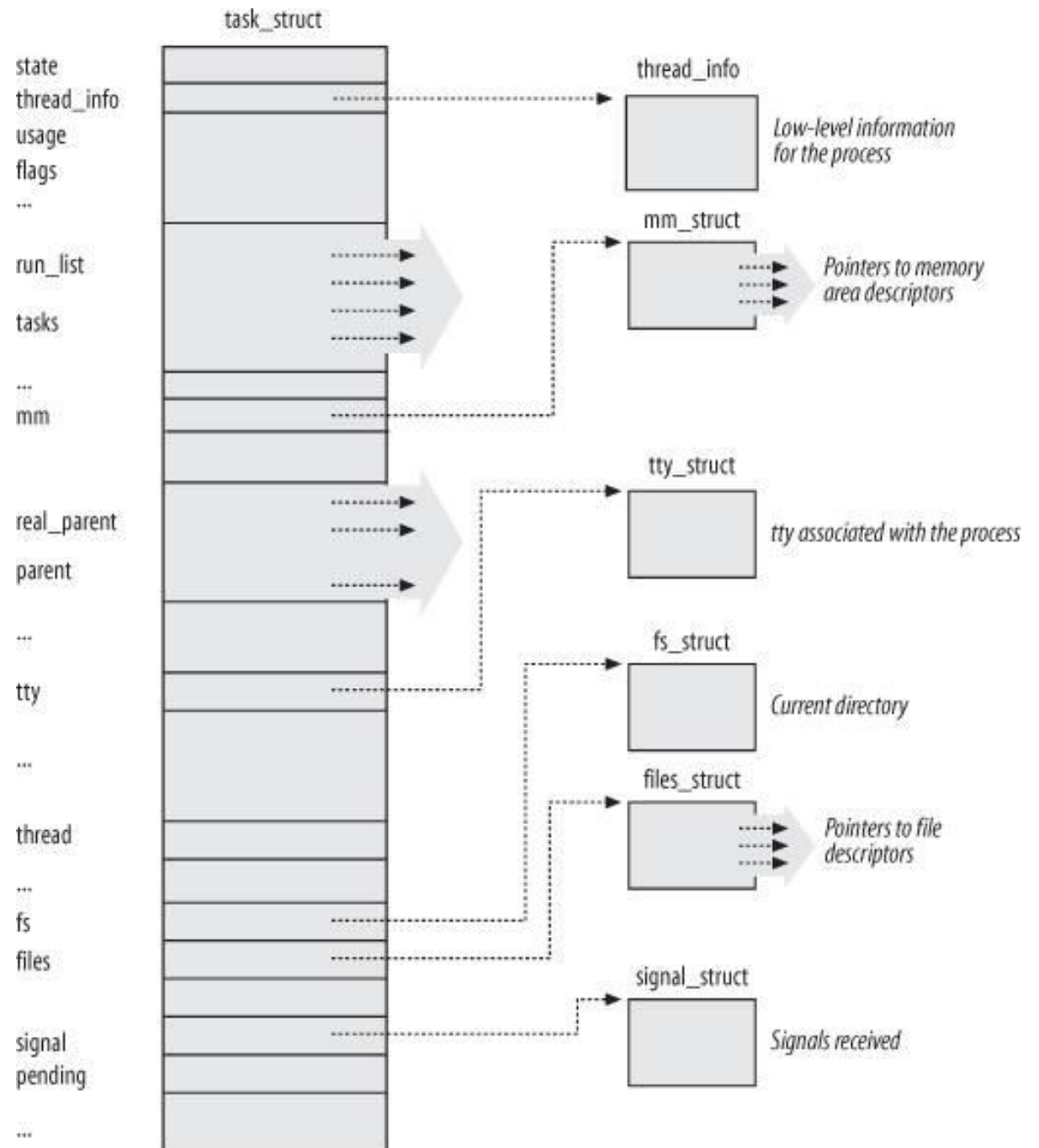
Process descriptor

In Linux, the `task_struct` type structure plays the role of the **process descriptor**.

It contains all information about the process (often in the form of pointers to other structures).

It is a structure associated with every existing process in the system.

The Linux **process descriptor** (source: Bovet, Cesati, Understanding the Linux Kernel)
– partly obsolete





Process descriptor

volatile long **state** – determines the state of the process.

struct nsproxy ***nsproxy** – pointer to an array with pointers to all per-process **name spaces** for different subsystems – fs (mount), uts, network, sysvipc, etc. The nsproxy is **shared** by tasks which share **all namespaces**. As soon as a single namespace is cloned or unshared, the nsproxy is **copied**.

void ***stack** – pointer to **thread_info** containing low-level information about the process, including the **kernel mode stack**.

The **parent process** and the **child process** after the **copy_process()** procedure inside the **do_fork()** procedure differ in the value of stack.

The **thread_info** structure stores low-level process information specific to the processor (and architecture-dependent).

struct thread_struct **thread** – process context, dependent on the CPU



Process descriptor

pid_t **pid** – unique process identifier (number).

pid_t **tgid** – the unique identifier of the **thread group** to which the process belongs.

The **group_leader** field indicates the process descriptor of the process that is the leader of this thread group.

struct task_struct ***real_parent**, **parent** – family connections of the process: original parent (during debugging), parent.

struct list_head **children**, **sibling** – family connections of the process

Two other identifiers are associated with the process:

- **process group ID** – the process can belong to a group of processes. The **setpgid(pid, pgid)** command sets the ID of the process group to pgid for the process specified by pid.
- **session ID** – processes can be associated with the same terminal session. To support such relationships, the **setsid()** command is used.

These identifiers are not kept directly in the process descriptor, but in the structure used to handle signals.

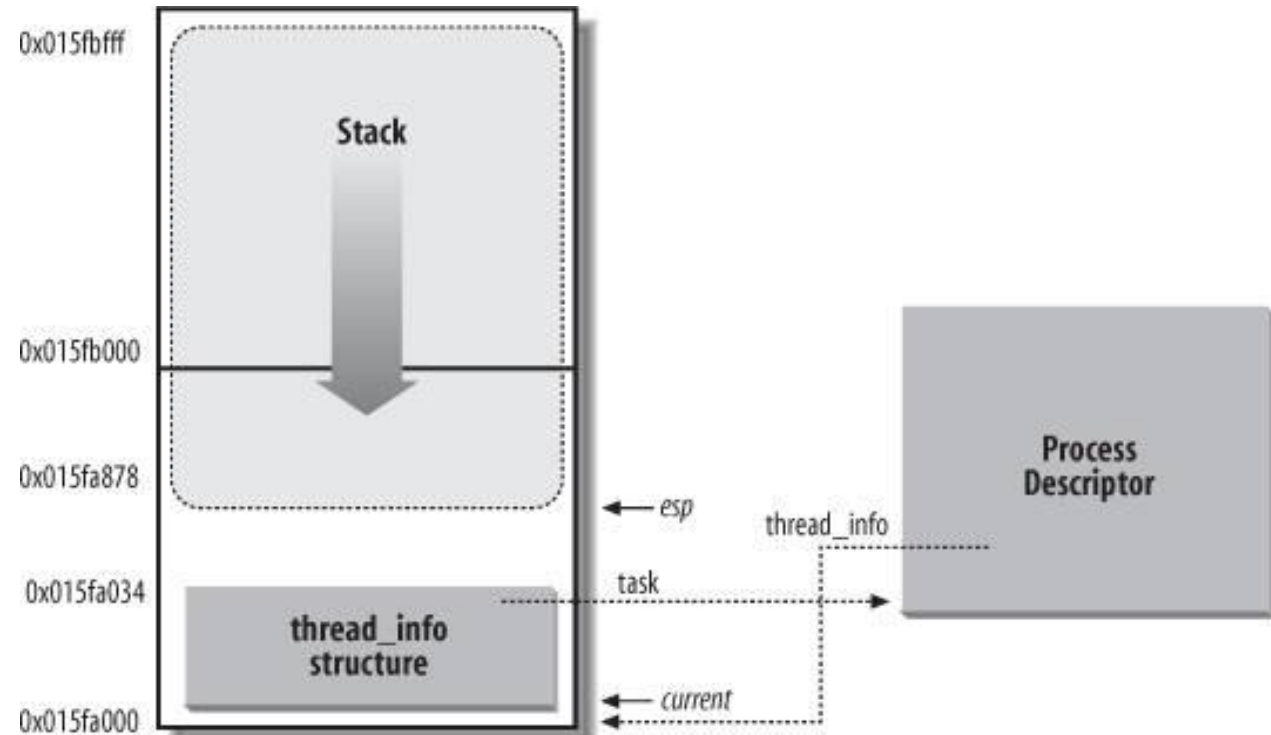


Process stack in kernel mode and thread_info

Until recently (and now it is also possible) the **process stack in kernel mode** was placed in one memory area together with the **thread_info** structure containing low-level process information. Both structures were kept in a single (contiguous), dynamically allocated memory area of a small size located in directly-mapped kernel memory.

The **thread_info** structure starts at the beginning of this memory area, while the **stack** at the end and grows "down".

The **thread_info structure** and the **process kernel stack**
(source: Bovet, Cesati, Understanding the Linux Kernel)





Process stack in kernel mode and thread_info

Stack size in 32-bit x86 processors – 8 KB (two page frames), 64-bit x86 processors – 16 KB (four page frames) or more.

```
#ifdef CONFIG_KASAN
#define KASAN_STACK_ORDER 1
#else
#define KASAN_STACK_ORDER 0
#endif

#ifdef CONFIG_64BIT
#define THREAD_SIZE_ORDER (2 + KASAN_STACK_ORDER)
#else
#define THREAD_SIZE_ORDER (1 + KASAN_STACK_ORDER)
#endif
#define THREAD_SIZE (PAGE_SIZE << THREAD_SIZE_ORDER)
```

KernelAddressSanitizer (KASAN) is a **dynamic memory error detector**. It provides a fast and comprehensive solution for finding **use-after-free** and **out-of-bounds** bugs. KASAN uses compile-time instrumentation for checking every memory access. Currently KASAN is supported only for the **64-bit x86** and **ARM** architectures.

[Sanitizing the Linux Kernel — On KASAN and other Dynamic Bug-finding Tools](#), Andrey Konoval



Process stack in kernel mode and thread_info

Such solution has its advantages and disadvantages (what?). For several years, kernel developers have been discussing other solutions, assuming a different **size** and **position** of the **stack** and **thread_info**:

- [4K stacks by default?](#) Jake Edge, 2008.
- [Expanding the kernel stack](#), Jonathan Corbet, 2014.
- [Virtually mapped kernel stacks \(!\)](#) (Jonathan Corbet, 2016) and [Virtually mapped stacks 2: thread_info strikes back](#) (Jonathan Corbet, 2016).
- [KernelNewbies: Linux 4.9](#) - this release adds support for ... virtually mapped kernel stacks that make the kernel more reliable and secure ...
- [Linus Torvalds' comments on Linux 4.9-rc1](#), October 2016.
- [Randomizing structure layout](#), Nur Hussein, 2017.



Process stack in kernel mode and thread_info

2008

The **memory savings can be significant**, especially in the embedded world. It would seem, however, premature to make **4 KB stacks the default**. Folks using **xfs** could run into problems.

2014

Some developers were trying to **shrink the stack to 4 KB**, but that effort eventually proved to be **unrealistic**. Modern kernels can end up creating **surprisingly deep call chains** that just do not fit into a 4KB stack. Those call chains don't even fit into an **8 KB** stack on **x86-64** systems.

2016

Each process has its own stack for use when it is running in the kernel; in current kernels, that stack is sized at either **8 KB** or (on 64-bit systems) **16 KB** of memory. The stack lives in **directly-mapped kernel** memory, so it must be **physically contiguous**.

As memory gets fragmented, finding two or four physically contiguous pages can become difficult.

The use of directly mapped memory also rules out the use of **guard pages** — non-accessible pages that would trap an overflow of the stack — because adding a guard page would require wasting an actual page of memory.



Process stack in kernel mode and thread_info



2016 continued

Andy Lutomirski's patch **allocates kernel stacks** from the **vmalloc area**.

It seems like a significant improvement to the kernel. There are a few outstanding issues, though.

One of those is performance; allocating a stack from the vmalloc area, makes creating a process with **clone()** take about **1.5 μ s longer**.

The **directly mapped area** uses **huge-page mappings**, so the **entire kernel** (all of its code, data, and stacks) can fit in a **single TLB entry**. The **vmalloc area** uses **single-page mappings**. Since references to kernel stacks are common, the possibility of an **increase in TLB misses** is real if those stacks are reached via the vmalloc area.

Finally, kernels with this patch set applied will **detect an overflow of the kernel stack**, but there is still the problem of the **thread_info** structure living at the bottom of each stack. An overrun that overwrites only this structure, without overrunning the stack as a whole, will not be detected.

The proper solution here is to **move the thread_info structure away from the kernel stack** entirely. The current patch set does not do that, but Andy has said that he intends to tackle that problem once these patches are accepted.



Process stack in kernel mode and thread_info

2016 continued

In theory there should be no need for a process's kernel stack after that process has died, so one might think that the stack could be released immediately. The problem is that the **core information** the kernel maintains **about processes** lives in two different places:

- The massive **task_struct structure**, architecture-independent,
- The small **thread_info structure**, which is architecture-specific.

The removal of the **thread_info** structure makes it possible to **free the kernel stack** as soon as the owning process exits — no **RCU grace period** required. That, in turn, makes it sensible to add a **small per-CPU cache** holding **up to two free kernel stacks**.

With the cache the **1.5 μ s performance regression** becomes a **0.5–1 μ s performance gain**.

2017

The **task_struct** structure is a prime example of a structure that benefits from **field randomization**. Inside **task_struct** are **sensitive fields** such as process credentials, flags for enabling or disabling process auditing, and pointers to other **task_struct** structures. Those fields, among others, are juicy targets for potential attackers to overwrite.

However, we can't just randomize the entirety of **task_struct**, as some fields on the **very top** and **very bottom** of the structure need to be where they are.

Linus: Making "struct task_struct" be something that contains a fixed beginning and end, and just have an unnamed randomized part in the middle might be the way to go.



Process stack in kernel mode and thread_info

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
/*
 * For reasons of header soup (see current_thread_info()), this
 * must be the first element of task_struct.
 */
struct thread_info      thread_info;
#endif

.....

/*
 * This begins the randomizable portion of task_struct. Only
 * scheduling-critical items should be added above here.
 */
randomized_struct_fields_start
.....
randomized_struct_fields_end

/* CPU-specific state of this task: */
struct thread_struct    thread;
/*
 * WARNING: on x86, 'thread_struct'
 * contains a variable-sized structure.
 * It *MUST* be at the end of 'task_struct'.
 *
 * Do not put anything below here!
 */
};
```

The **stack** and **thread_info** can still occupy one area, defined by the union **thread_union**, or **thread_info** can be placed in the **process descriptor**:

The **thread_info** structure is successively 'slimming'.

```
union thread_union {
#ifdef CONFIG_THREAD_INFO_IN_TASK
    struct thread_info thread_info;
#endif

    unsigned long stack[THREAD_SIZE/sizeof(long)];
};

#ifdef CONFIG_THREAD_INFO_IN_TASK
static inline struct thread_info *task_thread_info(struct task_struct *task)
{
    return &task->thread_info;
}
#elif !defined(__HAVE_THREAD_FUNCTIONS)
#define task_thread_info(task) ((struct thread_info *) (task)->stack)
#endif
```



Process address space

Introduction

How the kernel manages the **virtual address space of user processes**? The task is not simple because:

- each process has its **own** address space,
- the process address space consists of a set of **disjoint areas** of **different sizes**,
- only part of this address space is directly related to **physical memory page frames**,
- the kernel **trusts itself**, but it should **not trust user processes**,
- the Unix model of creating processes can be very ineffective if it is not carefully implemented.

Process address space is a collection of linear addresses referenced by the process during execution.

Each process has its **own page directory** (the size of one frame – this is also the size of page tables).

The pointer to it is in the **mm_struct** structure under the name **pgd**. When changing the context, Linux makes sure that it is loaded into the appropriate processor register (cr3).



Process address space

Introduction

Memory allocation for the process is done by extending the segment (function **sys_brk** changes the location of the **program break**, which defines the end of the process's data segment, i.e. the **program break** is the **first location** after the **end of the uninitialized data segment**).

Obtaining **physical memory** is implemented in the **page frame management system**.

The **get_free_page()** function obtains a page, clears it and passes its linear address. Hence the new memory delivered to the process is zeroed.

In a **32-bit architecture**, the **user process** can access a contiguous address space **0-4 GB**. The **upper gigabyte (3-4 GB)** is visible only in **kernel mode**. In user mode the user's data is spread in the range of **0-3 GB**. This upper limit is defined by the **TASK_SIZE** parameter.

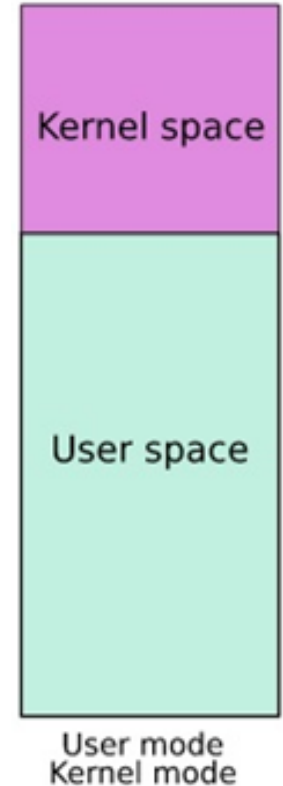
```
#ifdef CONFIG_X86_32
/*
 * User space process size: 3GB (default).
 */
#define TASK_SIZE      PAGE_OFFSET
#define TASK_SIZE_MAX  TASK_SIZE
#define STACK_TOP      TASK_SIZE
#define STACK_TOP_MAX  STACK_TOP
```



Process address space Introduction

Since the beginning, **Linux has mapped the kernel's memory into the address space of every running process**. There are **performance** reasons for doing this, and the processor's memory-management unit prevents user space from accessing that memory.

The **pgd_alloc()** function creates a new **page directory**, i.e. gets a page frame, fills it with zeros, and maps the address space of 3-4 GB to the kernel memory. **Zeros** in the page directory and in the page tables are treated as a frame **allocated** at the **first access**. The **upper gigabyte** is set to the **system**.



Keeping the kernel **permanently mapped eliminates the need to flush the TLB** when switching between **user** and **kernel space**, and it allows the **TLB entries for kernel space to never be flushed**.

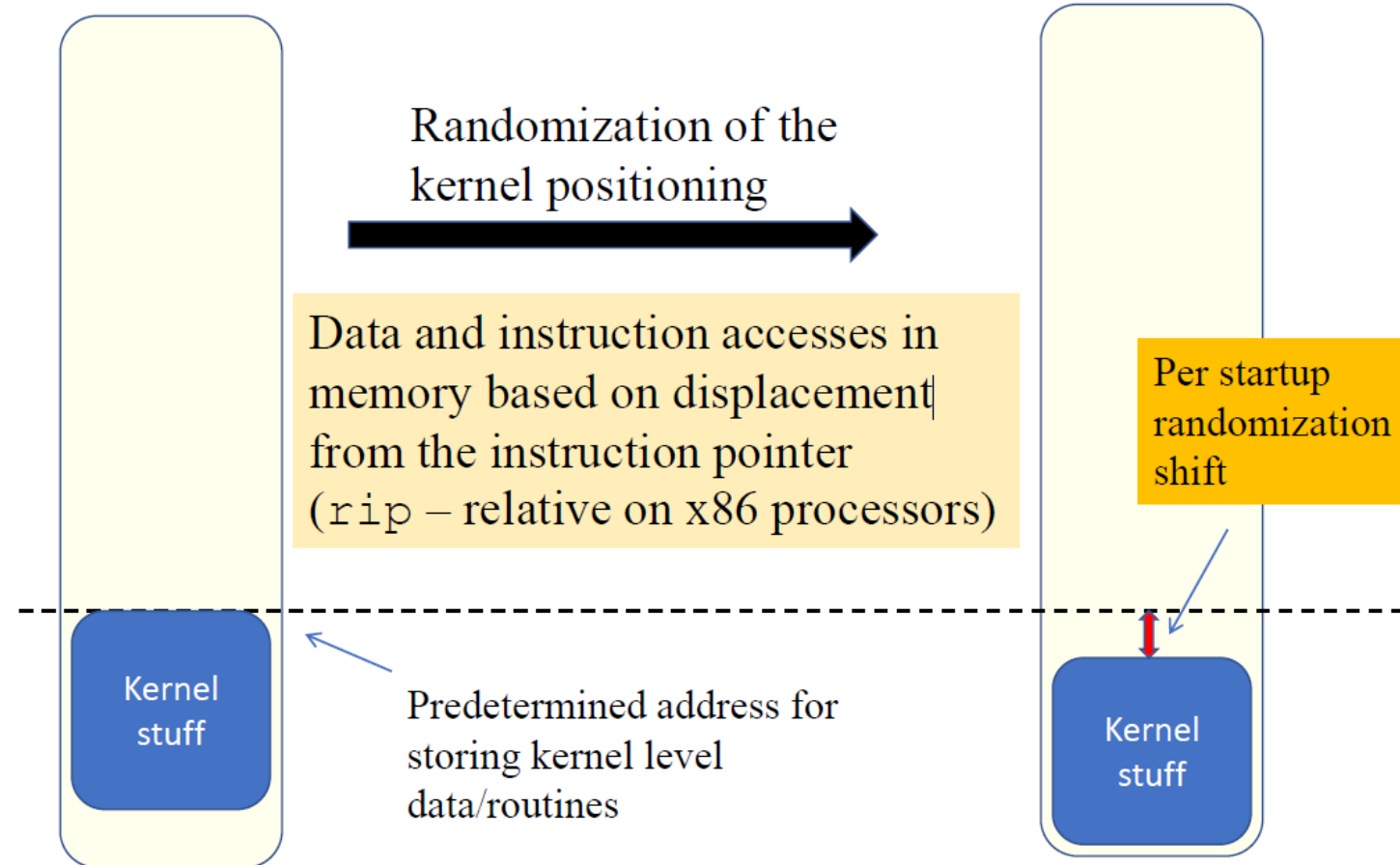
On contemporary **64-bit systems**, the shared address space does not constrain the amount of virtual memory that can be addressed as it used to, but there is another problem that is related to **security**.



A scheme for KASLR (*Kernel Address Space Layout Randomization*)

Classical virtual address space usage

Usage with randomization



KASLR has been merged into mainline Linux in 2014.



Process address space

Introduction

The paper from [Daniel Gruss et al.](#) (*KASLR is Dead: Long Live KASLR*) cites a **number of hardware-based attacks on KASLR** (*Kernel Address Space Layout Randomization*).

They use techniques like **exploiting timing differences in fault handling**, observing the behavior of **prefetch instructions**, or forcing **faults using the Intel TSX** (transactional memory) instructions.

The processor **responds differently** to a memory access attempt depending on whether the **target address is mapped in the page tables**, regardless of whether the running process can actually access that location.

These differences can be used to find where the **kernel has been placed** — without making the kernel aware that an attack is underway.

Strictly **splitting kernel space and user space** has been proposed to close these side channels.

This is not trivially possible due to **architectural restrictions** of the x86 platform.



Process address space

KASLR and KAISER (KPTI)

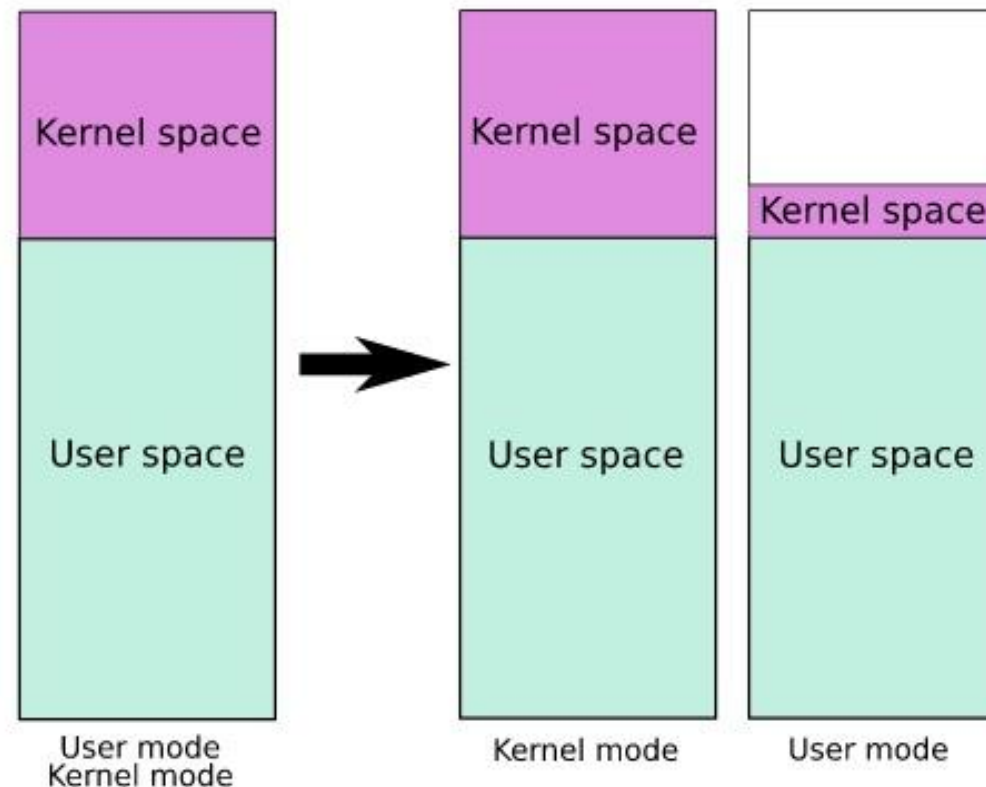
Additional reading

- [KASLR is Dead: Long Live KASLR](#) (June 2017) – paper introducing KAISER.
- [KAISER: hiding the kernel from user space](#), Jonathan Corbet, November 2017.
- [The current state of kernel page-table isolation](#), Jonathan Corbet, December 2017.
- [Linux Documentation in GitHub](#), January 2018.
- [Meltdown and Spectre](#), Piotr Zalas, March 2018.

KASLR – Kernel Address Space Layout Randomization

KAISER – Kernel Address Isolation to have Side-channels Efficiently Removed (renamed later to KPTI)

Kernel page-table isolation



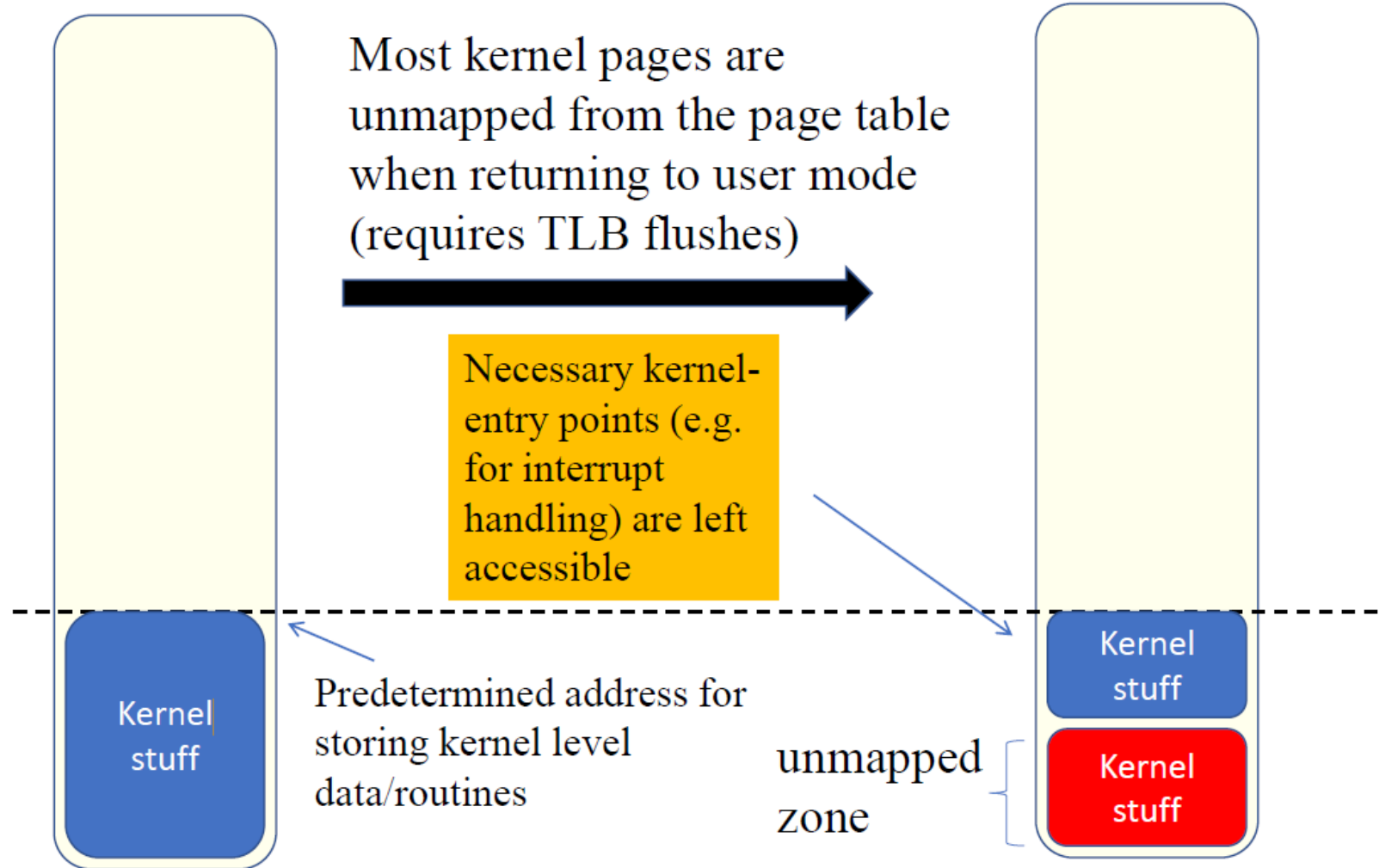
Kernel Page Table Isolation – KPTI (source: [Wikipedia](#))



A scheme for KAISER

Classical kernel mapping

Variation of the kernel mapping

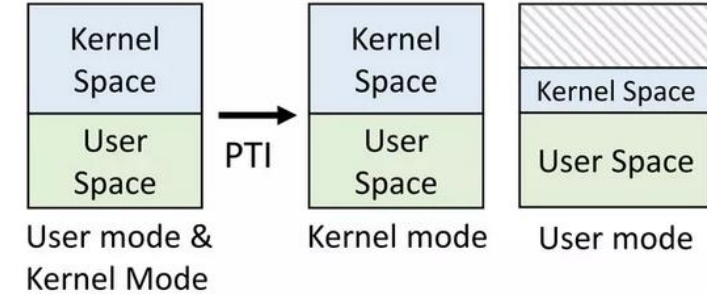




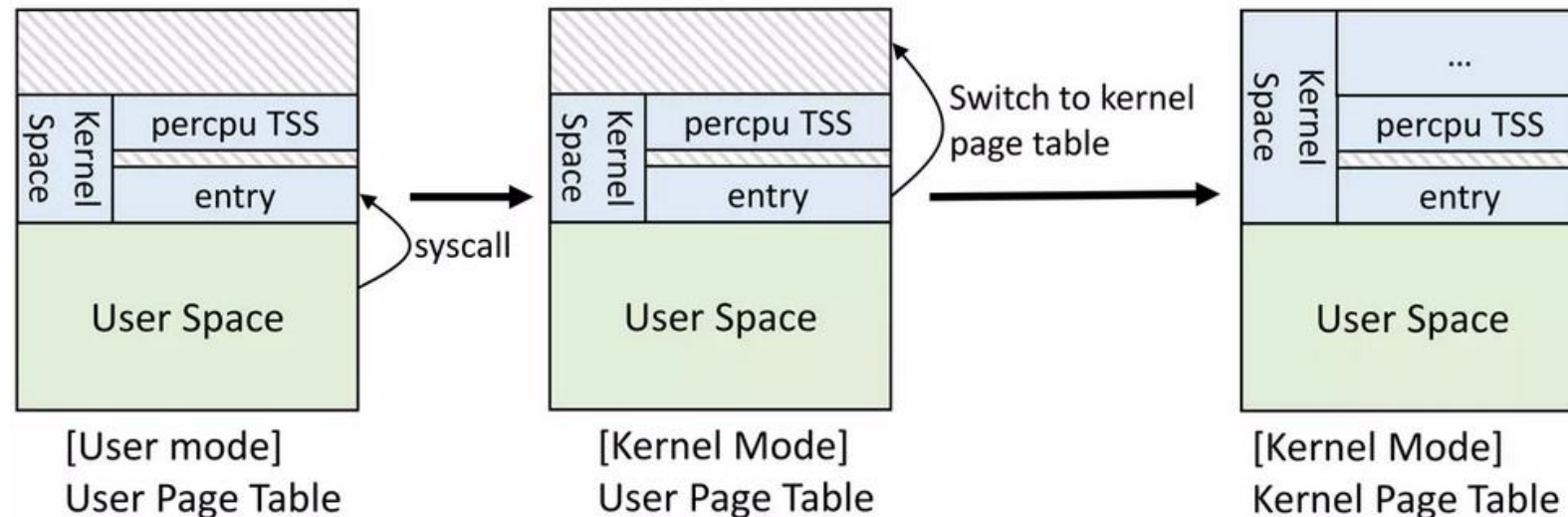
Start_kernel() – trap_init()

PTI: Concept

When the kernel is entered via syscalls, interrupts or exceptions, the page tables are switched to the full „kernel” copy.
Entry/exit functions and IDT (Interrupt Descriptor Table) are needed for userspace page table.



PTI: High-level implementation





Process address space KAISER (KPTI)

Whereas current systems have a **single set of page tables for each process**, **KAISER (KPTI)** implements **two** (there are two **PGDs**).

One set is essentially unchanged; it includes both **kernel-space** and **user-space addresses**, but it is only used when the system is running in **kernel mode**.

The second "shadow" page table contains a copy of all of the **user-space mappings**, but **leaves out the kernel side**.

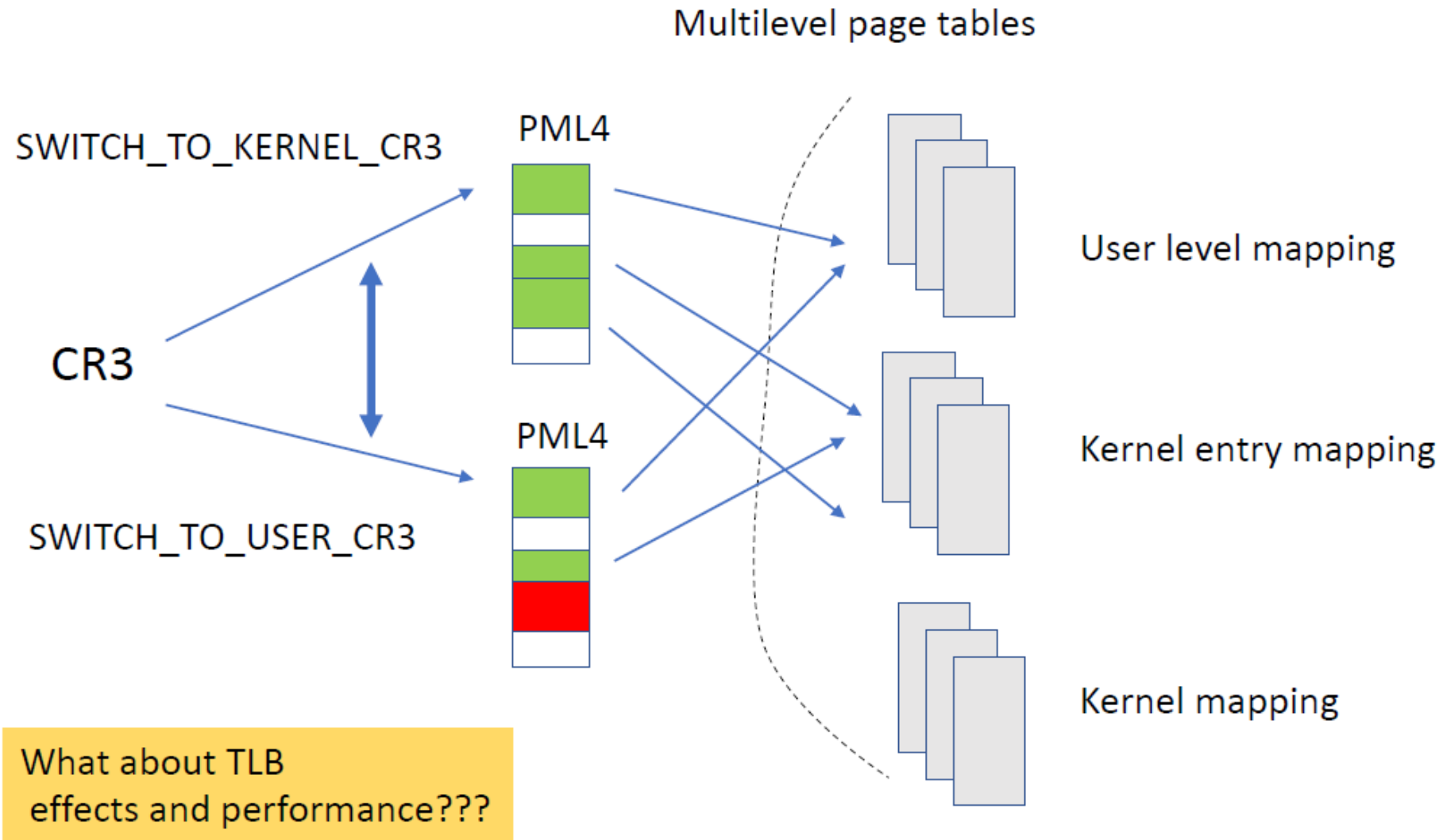
The processor responds to a **hardware interrupt** while running in **user mode**, the kernel code needed to deal with the interrupt will no longer exist in the address space.

So there must be enough **kernel code mapped** in **user mode** to switch back to the kernel PGD and make the rest available. A similar situation exists for **traps, non-maskable interrupts, and system calls**.

This code is **small** and can be isolated from the rest, but there are a number of tricky details involved in handling that switch safely and efficiently.



Actual implementation in Linux (on x86)





Process address space KAISER (KPTI)

Copying the page tables may sound **inefficient**, but the copying only happens at the **top level** of the **page-table hierarchy**, so the bulk of that data is shared between the two copies. The patches were merged into the mainline in **January 2018 (4.15)**.

More recent processors offer **process-context identifiers (PCIDs)**. These identifiers tag entries in the TLB. Use of PCIDs eliminates the need to flush the TLB at context switches; that reduces the cost of switching page tables during system calls. The kernel got support for PCIDs in **4.14**.

[Greg Kroah-Hartman said](#) he's seen one report of a „*Linux user benchmarking recent kernel versions on a specific network-heavy load*” which showed that, without anti-Meltdown Linux's Kernel Page Table Isolation (KPTI) patches, **the Linux kernel, 4.15 is 7- to 9-percent faster than April 30, 2017's 4.11 release.**

That's the good news. The bad news is that, **with KPTI, 4.14 is 1- to 2-percent slower than 4.11.**





Process address space KAISER (KPTI)

Another potential **vulnerability** comes about if the kernel can ever be manipulated into **returning to user space without switching back to the sanitized PGD**.

Since the kernel-space PGD also maps user-space memory, such an omission could go **unnoticed** for some time.

The response here is to **map the user-space portion of the virtual address space as non-executable in the kernel PGD**. Should user space ever start running with the wrong page tables, it will immediately **crash** as a result.

While all existing x86 processors are seemingly affected by information-disclosure vulnerabilities, **future processors** may not be.

KPTI comes with a **measurable run-time cost**, estimated at about **5%**. That is a cost that some users may not want to pay, especially once they get **newer processors** that lack these problems.

There is a **nopti** command-line option to disable this mechanism at **boot time**.



Additional reading

- [Inside the Linux boot process](#), M. Tim Jones.
- [A quick history of early-boot memory allocators](#), Mike Rapoport, July 2018.
- [Differences Between UEFI and BIOS](#), Jenna Tsui, 2020.
- [UEFI boot: how does that actually work, then?](#) Adam Williamson, 2014.
- [What is UEFI, how it differs from BIOS](#), Marek Kowalski, 2015 (in Polish).
- [UEFI vs BIOS - the most important differences](#), Marcin Jaskólski, 2011 (in Polish).
- [Kernel Address Space Isolation](#), Alexandre Chartre (Oracle), James Bottomley (IBM), Mike Rapoport (IBM), Joel Nider (IBM Research), Linux Plumbers Conference, 2019.
- [Address Space Isolation in the Linux Kernel](#), James Bottomley, Mike Rapoport, FOSDEM 2020.
- [How programs get run](#), David Drysdale, January 2015.
- [How programs get run: ELF binaries](#), David Drysdale, February 2015.
- [Questions from the lecture](#) (in Polish).
- [Don't shoot down TLB shutdowns! \(paper\)](#), [Don't shoot down TLB shutdowns! \(presentation from EuroSys 2020\)](#)