The Linux Kernel Data Structures Journey

version 1.0 (beta)
August-2023

By Shlomi Boutnaru

Created using Craiyon. AI Image Generato

Introduction

When starting to read the source code of the Linux kernel I believe that they are basic data structures that everyone needs to know about. Because of that I have decided to write a series of short writeups aimed at providing the basic vocabulary and understanding for achieving that.

Overall, I wanted to create something that will improve the overall knowledge of Linux kernel in writeups that can be read in 1-3 mins. I hope you are going to enjoy the ride.

Lastly, you can follow me on twitter - @boutnaru (https://twitter.com/boutnaru). Also, you can read my other writeups on medium - https://medium.com/@boutnaru.

Lets GO!!!!!!

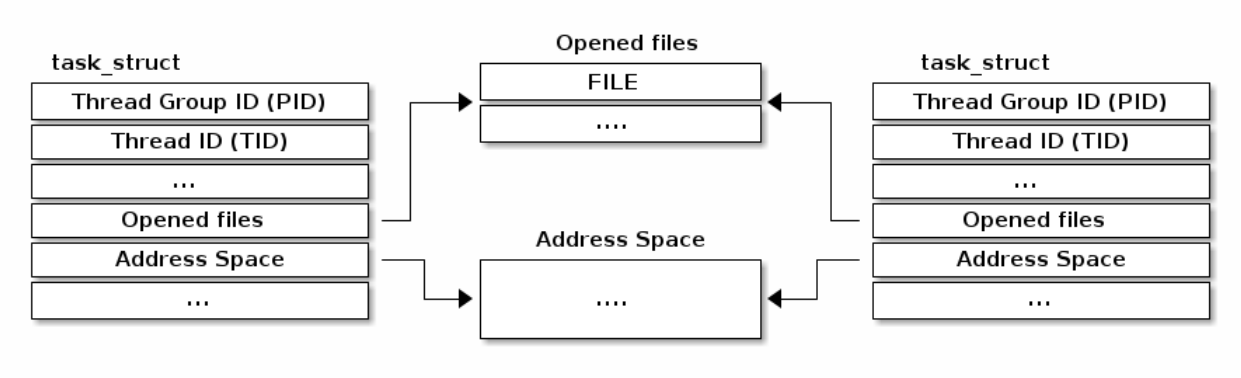
struct task_struct

Every operating system has a data structure that represents a "process[1] object" (generally called PCB - Process Control Block). By the way, "task_struct" is the PCB in Linux (it is also the TCB, meaning the Thread Control Block). As an example, a diagram that shows two processes opening the same file and the relationship between the two different "task_strcut" structures is shown below.

Overall, we can say that "task_struct" holds the data an operating system needs about a specific process. Among those data elements are: credentials ,priority, PID (process ID), PPID (parent process ID), list of open resources, memory space range information, namespace information[2], kprobes[3] instances and more.

Moreover, If you want to go over all of data elements I suggest going through the definition of "task_strcut" as part of the Linux source code[4]. Also, fun fact is that in kernel 6.2-rc1 "task_strcut" is referenced in 1398 files[5].

Lastly, familiarity with "task_struct" can help a lot with tracing and debugging tasks as shown in the online book "Dynamic Tracing with DTrace & SystemTap"[6]. Also, it is very handy when working with bpftrace. For example **sudo bpftrace -e 'kfunc:hrtimer_wakeup { printf("%s:%d\n",curtask->comm,curtask->pid); }',** which prints the pid and the process name of all processes calling the kernel function hrtimer_wakeup[7].



[1] https://medium.com/@boutnaru/linux-processes-part-1-introduction-283f5b5b4197
[2] https://medium.com/system-weakness/linux-namespaces-part-1-dcee9c40fb68
[3] https://medium.com/@boutnaru/linux-instrumentation-part-2-kprobes-b089092c4cff
[4] https://elixir.bootlin.com/linux/v6.2-rc1/source/include/linux/sched.h#L737
[5] https://elixir.bootlin.com/linux/v6.2-rc1/A/ident/task_struct
[6] https://myaut.github.io/dtrace-stap-book/kernel/proc.html
[7] https://medium.com/@boutnaru/the-linux-process-journey-pid-0-swapper-7868d1131316

struct mm_struct

The goal of "mm_struct" (aka "Memory Descriptor") is to used by the Linux kernel in order to represents the process' address space[8]. It is the user-mode part which belongs to the task/process[9]. By the way, until kernel version 2.6.23 the struct was defined under "/include/linux/sched.h"[10], since "2.6.24" it is located under "/include/linux/mm_types.h"[11]. Also, there is a pointer from the process' "task_struct"[12] that refers to the address space of the process ("mm_struct") which is stored in the "mm" field[13].

Overall, we can say that "mm_struct" holds the data Linux needs about the memory address space of the process. Among those data elements are: "mm_users" (the number of tasks using this address space), "map_count" (the number of virtual memory areas, VMAs, used by the task) and "total_vm" (the total number of pages mapped by the task). You can see part of the information stored in "mm_struct" by going over "/proc/[PID]/maps" ("man proc").

Moreover, If you want to go over all of data elements I suggest going through the definition of "mm_struct" as part of the Linux source code[14]. Fun fact is that in kernel 6.2-rc1 "task_strcut" is referenced in 656 files[15]. Based on LXR[16] it seems that "mm_struct" was added from kernel version 1.1.11[17] as we don't see it in previous versions[18].

Lastly, let us go over an example using bpftrace. We can use the following command: **sudo bpftrace -e 'kfunc:schedule { printf("%s,%d,%d\n",curtask->comm,curtask->pid,curtask->mm->map_count); }'.** As shown in the screenshot below, the command prints the name, pid and VMA count for the current task every time the scheduler function is triggered. From the screenshot we can see that kernel threads have a count of zero VMAs (in their case current->mm==NULL). Also, we can see that the "map_count" is one less[19] than the number of rows in "/proc/[PID]/maps".

[8] http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch14lev1sec1.html
[9] https://medium.com/@boutnaru/linux-memory-management-part-1-introduction-896f376d3713
[10] https://elixir.bootlin.com/linux/v2.6.23/source/include/linux/sched.h#L369
[11] https://elixir.bootlin.com/linux/v2.6.24/source/include/linux/mm_types.h#L156
[12] https://medium.com/@boutnaru/linux-kernel-task-struct-829f51d97275
[13] https://elixir.bootlin.com/linux/v6.2-rc1/source/include/linux/sched.h#L870
[14] https://elixir.bootlin.com/linux/v6.2-rc1/source/include/linux/mm_types.h#L601
[15] https://elixir.bootlin.com/linux/v6.2-rc1/C/ident/mm_struct
[16] https://elixir.bootlin.com
[17] https://elixir.bootlin.com/linux/1.1.11/source/include/linux/sched.h#L214
[18] https://elixir.bootlin.com/linux/1.1.10/A/ident/mm_struct
[19] It is due to the mechanism of vsyscall

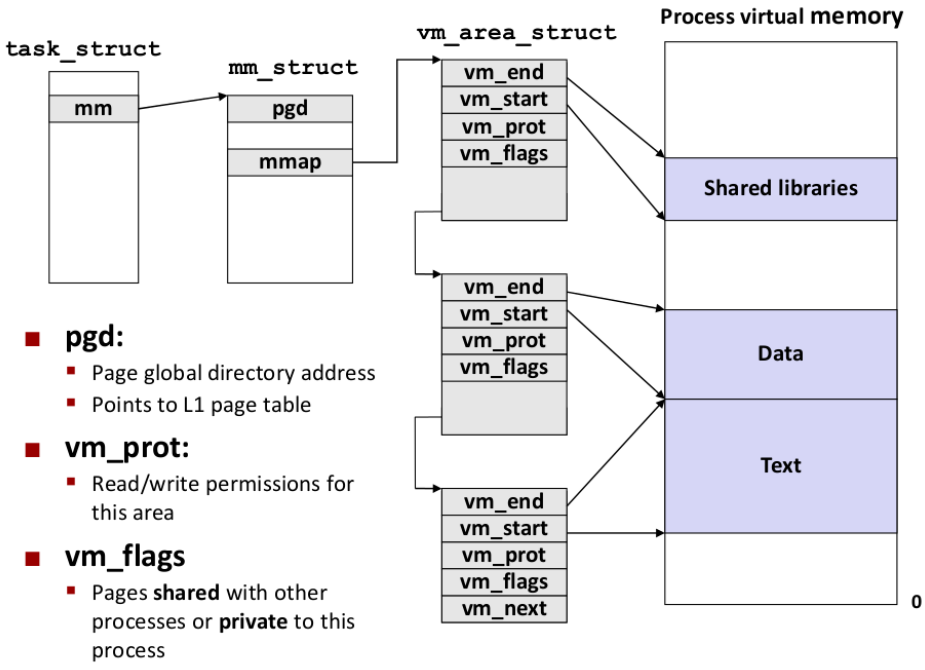
struct vm_area_struct

“vm_area_struct” represents a contiguous memory area in a process's address space (virtual memory areas) - as shown in the diagram below[20]. It is used to track the permissions, properties, and operations associated with each memory area[21].

This struct defines a memory VMM memory area. There is one of these per VM-area/task. A VM area is any part of the process virtual memory space that has a special rule for the page-fault handler. Think about shared libraries and executable area[22].

Until kernel version 2.6.21 (including) “struct vm_area_struct” is defined in “/include/linux/mm.h”[23]. From kernel versions about it is defined in “/include/linux/mm_types.h”[24].

Lastly, by using “/proc/[PID]/maps” we can read the mapped regions and their access permissions (when using the mmap system call). For each region we can get the information about: its address range, pathname (in case mapped from a file), offset (in case mapped from a file), device (in case mapped from a file), inode (in case mapped from a file and permissions)[25].



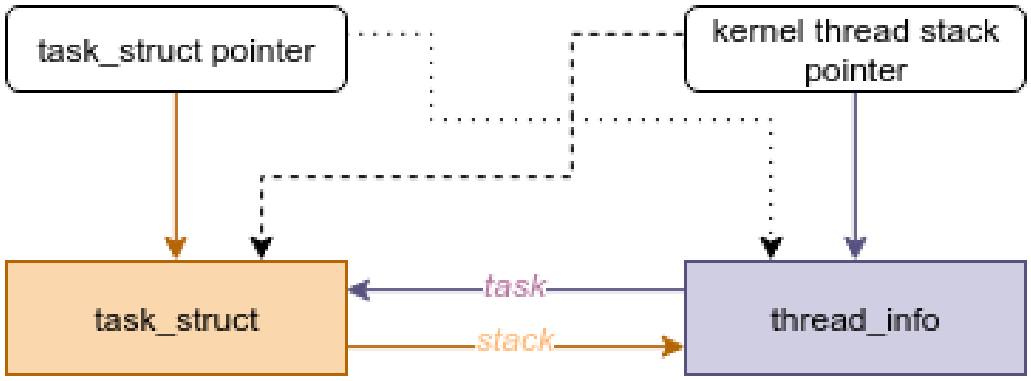
[20] https://don7hao.github.io/2015/01/28/kernel/mm_struct/
[21] http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch14lev1sec2.html
[22] https://elixir.bootlin.com/linux/v2.6.21/source/include/linux/mm.h#L55
[23] https://elixir.bootlin.com/linux/v2.6.21/source/include/linux/mm.h#L60
[24] https://elixir.bootlin.com/linux/v6.5-rc1/source/include/linux/mm_types.h#L490
[25] https://man7.org/linux/man-pages/man5/proc.5.html

struct therad_info

“struct thread_info” is a common low-level thread information accessors[26] (think about flags like signal pending, 32 address space on 64 bit, CPUID is not accessible in user mode and more). When “CONFIG_THREAD_INFO_IN_TASK” is defined, “struct thread_info” is the first element of the “struct task_struct”[27]. This means that each task has its own “struct thread_info”.

Moreover, until kernel version 4.8 (including it) “struct thread_info” contained a pointer to “struct task_struct”[28] - the old relationship is shown in the diagram below[29]. But because it wasted too much space to keep it like that. By putting “struct thread_info” in the start of “struct task_struct” it makes getting from the “kernel stack”->”struct task_struct” and from “struct task_struct”->”kernel stack” very easy[30]. The first is done using the “current_thread_info” macro[31]. It uses “current_task” which is a per-cpu variable[32].

Lastly, it is important to understand that this data structure is CPU dependent and thus it is defined in the source code in the following location “/arch/[CPU_ARCH]/include/asm/thread_info.h”. Examples for “CPU_ARCH” could be: “x86”[33], “mips”[34], “riscv”[35], “arm64”[36] and more.



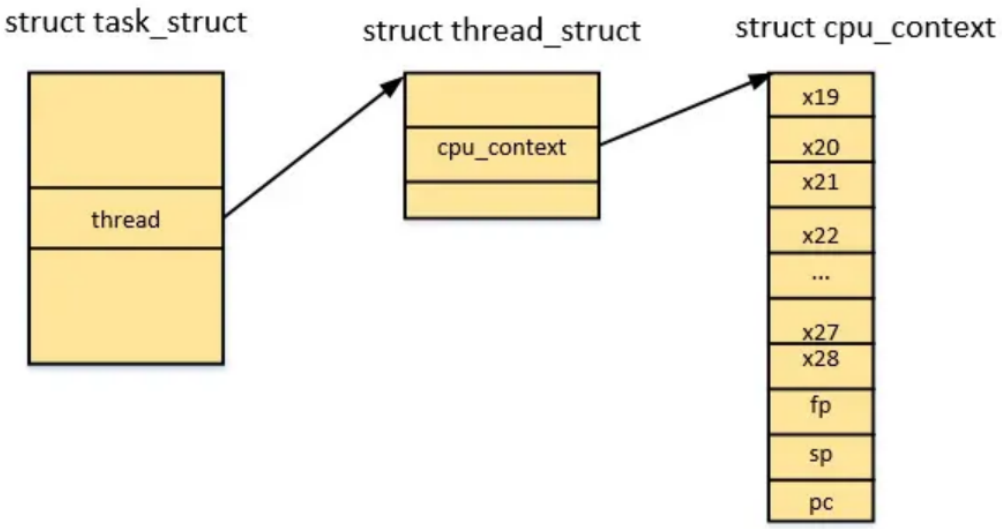
[26] https://elixir.bootlin.com/linux/v6.5-rc3/source/include/linux/thread_info.h#L2
[27] https://medium.com/@boutnaru/linux-kernel-task-struct-829f51d97275
[28] https://elixir.bootlin.com/linux/v4.8.17/source/arch/x86/include/asm/thread_info.h#L56
[29] https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part4.html
[30] https://stackoverflow.com/questions/61886139/why-thread-info-should-be-the-first-element-in-task-struct
[31] https://elixir.bootlin.com/linux/v6.5-rc3/source/include/linux/thread_info.h#L24
[32] https://elixir.bootlin.com/linux/v6.5-rc3/source/arch/x86/include/asm/current.h#L41
[33] https://elixir.bootlin.com/linux/v6.5-rc3/source/arch/x86/include/asm/thread_info.h#L56
[34] https://elixir.bootlin.com/linux/v6.5-rc3/source/arch/mips/include/asm/thread_info.h#L25
[35] https://elixir.bootlin.com/linux/v6.5-rc3/source/arch/riscv/include/asm/thread_info.h#L51
[36] https://elixir.bootlin.com/linux/v6.5-rc3/source/arch/arm64/include/asm/thread_info.h#L24

struct thread_struct

Overall, the goal of “struct thread_strcut” which hold CPU-specific state of a task[37]. Among the information “struct thread_struct” holds we can include things like page fault information (like the address that caused the page fault and fault code). Also, it can include a set of registers of the current CPU - as shown in the the diagram below[38].

On x86 the variable which is part of “struct task_struct”[39] must be at the end of the struct. The reason for that is it contain a variable-sized structure[40].

Moreover, like “struct thread_info”[41] also “struct thread_sturct” is CPU/Architecture dependent and thus it is defined in the source code in the following location “/arch/[CPU_ARCH]/include/asm/processor.h”[42]. For example x86[43] and arm64[44].



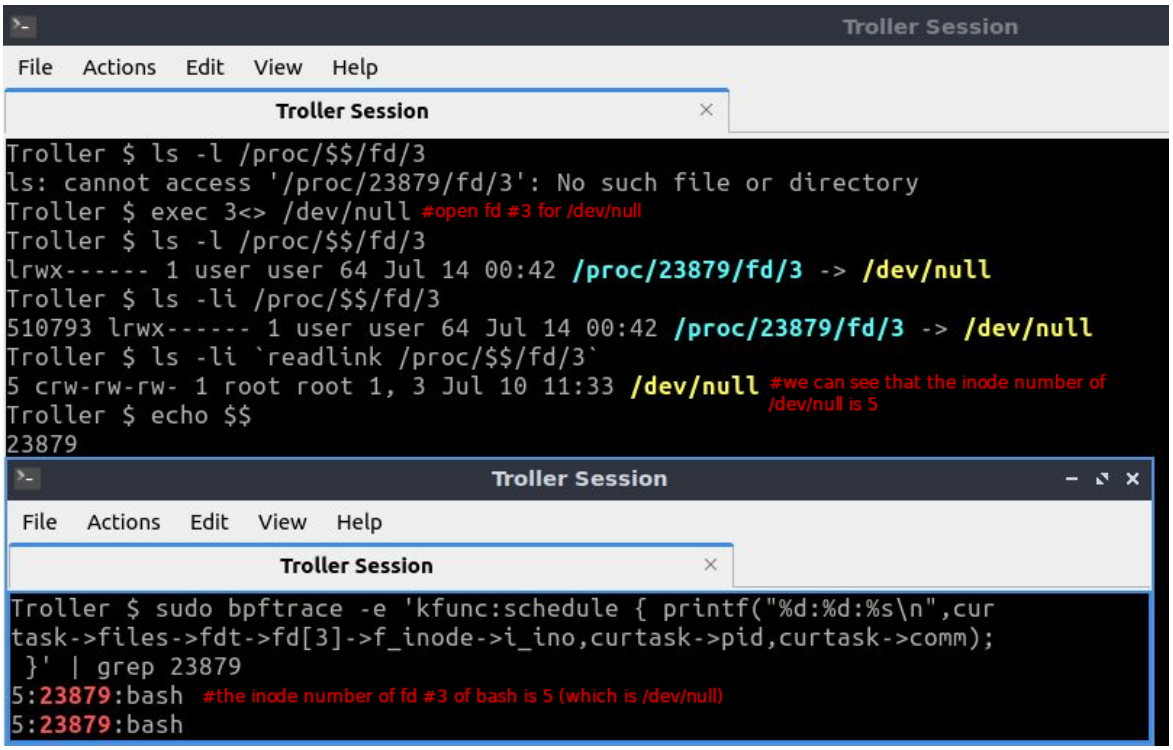
[37] https://elixir.bootlin.com/linux/v6.5-rc4/source/include/linux/sched.h#L1540
[38] https://kernel.0voice.com/forum.php?mod=viewthread&tid=2920
[39] https://medium.com/@boutnaru/linux-kernel-task-struct-829f51d97275
[40] https://elixir.bootlin.com/linux/v6.5-rc4/source/include/linux/sched.h#L1544
[41] https://medium.com/@boutnaru/the-linux-kernel-data-structure-journey-struct-thread-info-4e70bc20d279
[42] https://elixir.bootlin.com/linux/v6.5-rc4/C/ident/thread_struct
[43] https://elixir.bootlin.com/linux/v6.5-rc4/source/arch/x86/include/asm/processor.h#L414
[44] https://elixir.bootlin.com/linux/v6.5-rc4/source/arch/arm64/include/asm/processor.h#L147

struct inode

Overall, the goal of an inode[45] object is to hold all the data needed by the kernel in order to perform actions on directories/files. In case of Unix/Linux style filesystem (like ext4) it can read from the on-disk inode[46]. Moreover, inode is an essential component of Unix filesystem and of VFS (Virtual File system). We can say that inode exists both in memory as part of VFS and as an entity on disk. The inode as part of VFS is represented by "struct inode"[47].

Also, the "struct inode" is defined in "/include/linux/fs.h" in the source tree of Linux[48]. It contains different members such as: jiffies of first dirtying, LRU list, backing dev writeback list, inode number, a mask of notify events that the inode cares about and more.

Lastly, let us go over an example using bpftrace - as shown in the screenshot below. First we open fd 3 to "/dev/null" (using exec). Then we check and see that the inode of "/dev/null" is 5. We can use the following command: **sudo bpftrace -e 'kfunc:schedule { printf("%d:%d:%s\n",curtask->files->fdt->fd[3]->f_inode->i_ino,curtask->pid,curtask->comm); }'.** As shown in the screenshot below it goes over different structs until getting to the inode of fd(3) and prints its number (which is 5). By the way, the example was conducted on kernel version 5.15.



[45] https://medium.com/@boutnaru/linux-what-is-an-inode-7ba47a519940
[46] http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch12lev1sec6.html
[47] https://linux-kernel-labs.github.io/refs/heads/master/labs/filesystems_part2.html
[48] https://elixir.bootlin.com/linux/v6.5-rc1/source/include/linux/fs.h#L608