The Linux Kernel Data Structures Journey

Version 2.0
April-2024

By Dr. 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. Lastly, You can find my free eBooks at https://TheLearningJourneyEbooks.com.

Lets GO!!!!!!

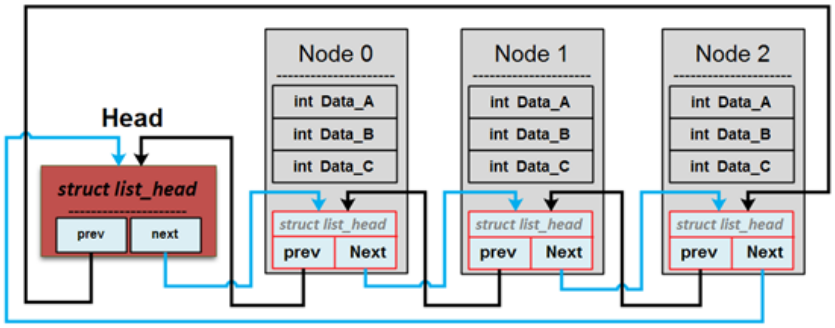
struct list_head

Overall, many times operating systems need to hold a list of data structures. In order reduce code duplication the Linux's kernel developers created a standard implementation of a circular/doubly-linked list[1]. This implementation was included in kernel 2.1.45[2]. At the beginning "struct list_head" was declared in "/include/linux.list.h" until kernel version 2.6.36 when it was moved to "/include/linux/types.h"[3].

Basically, "struct list_head" has two members. The first is "next" which is used to point to the next element in the list. The second is "prev" which points to the previous element in the list. As we can see the data is not included as part of this data structure, instead "struct list_head" is contained inside the data structures of the linked list - as shown in the diagram below[4].

Thus, in order to access the data from a specific "struct list_head" we can use the "container_of" macro that casts a member of a structure out to the containing structure[5]. This is provided by the "offsetof" macro[6] that leverages the "__builtin_offsetof" extension of GCC[7].

Moreover, the function/macros used to manipulate whole lists/single entries are part of "list.h"[8]. In order to initialize a "struct list_head" we use the "LIST_HEAD" macro[9]. For adding an element we can use "list_add" function to insert an entry after a specific head it is great for implementing stacks[10]. Also, the "list_add_tail" function allows inserting a new entry before a specific head, which is great for implementing queues[11]. Lastly, we can use the function "list_del" in order to delete a specific element from a list[12].



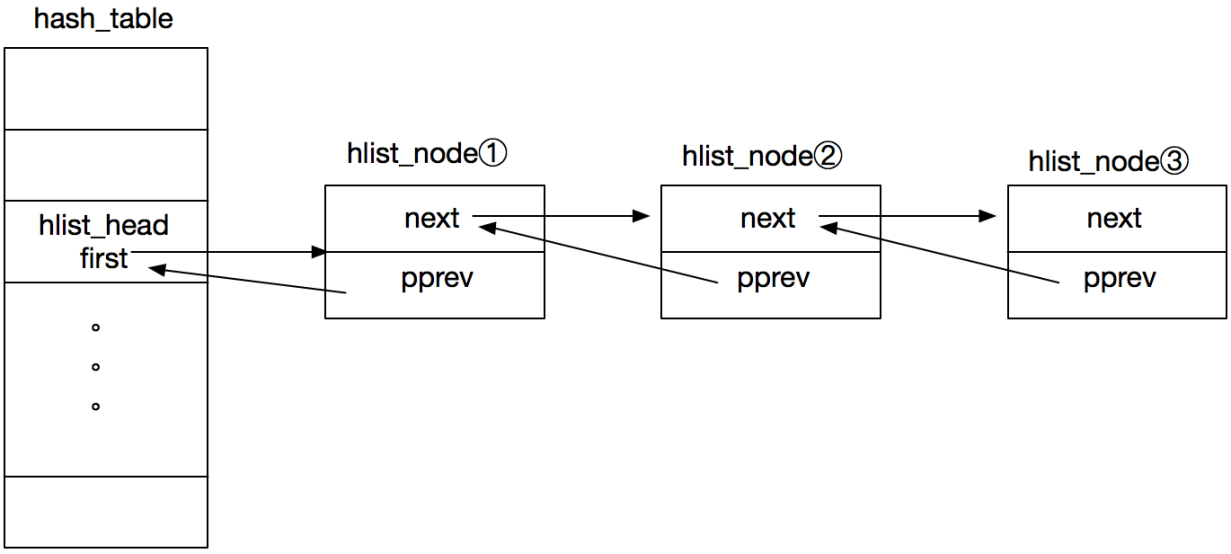
[1] https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch10s05.html
[2] https://elixir.bootlin.com/linux/2.1.45/source/include/linux/list.h
[3] https://elixir.bootlin.com/linux/v2.6.36/source/include/linux/types.h
[4] https://www.byteisland.com/linux-%E5%86%85%E6%A0%B8%E5%8F%8C%E5%90%91%E9%93%BE%E8%A1%A8/
[5] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/container_of.h#L11
[6] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/stddef.h#L16
[7] https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Offsetof.html
[8] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/list.h#L14
[9] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/list.h#L25
[10] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/list.h#L86
[11] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/list.h#L100
[12] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/list.h#L146

struct hlist_head

Overall, “struct hlist_head” is used (together with “struct hlist_node”) in the Linux kernel as part of hash tables. As opposed to “struct list_head”[13] “struct hlist_head” has only one data member which is “first”[14]. It points to the first node of a double linked list of “struct hlist_node”[15] - as shown in the diagram below[16].

Thus, we can say it enables the creation of a linked list with a single pointer list head. By doing so, we lose the ability to access the tail in O(1)[17]. We can create such a list head using the macro “HLIST_HEAD”[18], as opposed to “LIST_HEAD” (for “struct list_head”). There are a couple of functions that we can use to manipulate like “hlist_add_head” (adds a new entry at the beginning of the hlist), “hlist_move_list” (moves a list from one list head to another) - by the way those are not the only ones[19].

Lastly, as with “struct list_head” also “struct hlist_head” is part of a double linked list, however it is the head of the list that has only one pointer forward. Examples for using “struct hlist_head” is for holding list of data flows that are passing through a tun/tap device[20] and an open hash table implementation[21].



[13] https://medium.com/@boutnaru/the-linux-kernel-data-strctures-journey-struct-list-head-87fa91a5ce1c
[14] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/types.h#L188
[15] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/types.h#L192
[16] https://linux.laoqinren.net/kernel/hlist/
[17] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/list.h#L844
[18] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/list.h#L851
[19] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/list.h#L851
[20] https://elixir.bootlin.com/linux/v6.4.12/source/drivers/net/tun.c#L199
[21] https://elixir.bootlin.com/linux/v6.4.12/source/drivers/gpu/drm/drm_hashtab.c#L115

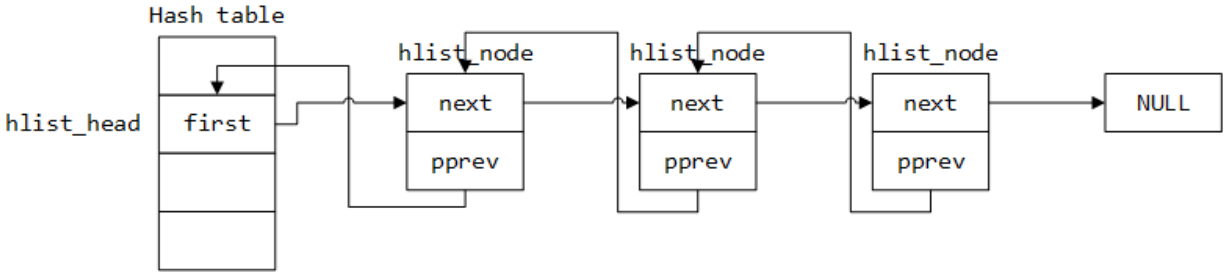
struct hlist_node

As mentioned in the writeup about “struct hlist_head”[22] as opposed to “struct list_head” in which all the elements in the linked list are from the same type, in the case of an “hlist” we also have “struct hlist_node” - as shown in the diagram below[23].

Moreover, “struct hlist_node” has two fields: “next” and “pprev”. “next” points to the next node in the linked list, while “pprev” points to the previous node in the linked list. We can find the declaration of this data structure in the Linux source code in the following location “/include/linux/types.h”[24].

Also, as of kernel 6.5 this data structure is reference in 505 files across the Linux source code as part of different components like: filesystems, networking stack, gpu drivers, memory management and more[25].

Lastly, we can use different functions to manipulate “hlist” nodes. “hlist_del” which deletes a “struct hlist_node” from its list[26]. “hlist_add_before” which adds a new entry before the one specified[27]. “hlist_add_behind” which adds a new entry after the specific one[28]. Of course those are not the only functions/macros that we can use, you can find more of them in “/include/linux/list.h”[29].



[22] https://medium.com/@boutnaru/the-linux-kernel-data-strctures-journey-struct-list-head-87fa91a5ce1c
[23] https://www.freesion.com/article/2435513265/
[24] https://elixir.bootlin.com/linux/v6.5/source/include/linux/types.h#L198
[25] https://elixir.bootlin.com/linux/v6.5/A/ident/hlist_node
[26] https://elixir.bootlin.com/linux/v6.5/source/include/linux/list.h#L905
[27] https://elixir.bootlin.com/linux/v6.5/source/include/linux/list.h#L951
[28] https://elixir.bootlin.com/linux/v6.5/source/include/linux/list.h#L951
[29] https://elixir.bootlin.com/linux/v6.5/source/include/linux/list.h

struct llist_head

Basically, “struct llist_head” is the head of a lock-less NULL terminated linked list[30]. Lock-less means that the data structure can be modified/accessed by multiple code flows without the need of locking. Moreover, “struct llist_head” is defined in the Linux source code in the following location “/include/linux/llist.h”[31] and not in “/include/linux/list.h” as we saw with “struct list_head”[32] and “struct hlist_head”[33]. As we can see in the source code it has only one field “first” of type “struct llist_node” which points to the list’s first node.

The cases in which locking is not needed are as follows. First, if we have multiple producers/consumers the producers can use “llist_add”[34] and the consumers can simultaneously “llist_del_all”[35] without locking. Second, in case of a single consumer which can use “llist_del_first”[36] while multiple producers can use “llist_add”[37].

However, locking is needed in the following case. If we have multiple consumers and one is calling “llist_del_first” we can’t simultaneously call “llist_del_first”/”llist_del_all” without a lock[38].

Thus, we can say that the lockless property of “llist” cases a reduce in functionality. The limitations as described above are: adding elements only at the start of the list and removing the first element/all elements[39]. A table that summarizes all the cases which were taken from the source code comments is shown below[40]. Lastly, as of kernel version 6.5 “struct llist_head” is referenced in 46 code files[41].

```
 * This can be summarized as follows:
 *
 *           |   add    | del_first |  del_all
 * add       |    -     |     -     |     -
 * del_first |          |     L     |     L
 * del_all   |          |           |     -
 *
 * Where, a particular row's operation can happen concurrently with a column's
 * operation, with "-" being no lock needed, while "L" being lock is needed.
 *
```

[30] https://elixir.bootlin.com/linux/v6.5/source/include/linux/llist.h#L60
[31] https://elixir.bootlin.com/linux/v6.5/source/include/linux/llist.h#L56
[32] https://medium.com/@boutnaru/the-linux-kernel-data-strctures-journey-struct-list-head-87fa91a5ce1c
[33] https://medium.com/@boutnaru/the-linux-kernel-data-strcture-journey-struct-hlist-head-19f4ceb71295
[34] https://elixir.bootlin.com/linux/v6.5/source/include/linux/llist.h#L219
[35] https://elixir.bootlin.com/linux/v6.5/source/include/linux/llist.h#L237
[36] https://elixir.bootlin.com/linux/v6.5/source/lib/llist.c#L53
[37] https://elixir.bootlin.com/linux/v6.5/source/include/linux/llist.h#L7
[38] https://elixir.bootlin.com/linux/v6.5/source/include/linux/llist.h#L14
[39] https://stackoverflow.com/questions/38771637/add-to-tail-of-lock-less-list
[40] https://elixir.bootlin.com/linux/v6.5/source/include/linux/llist.h#L25
[41] https://elixir.bootlin.com/linux/v6.5/C/ident/llist_head

struct llist_node

As mentioned in the writeup about “struct llist_head”[42] as like with “struct hlist_head”[43] with “llist” we also have a data structure for the node which is “struct llist_node”.

Overall, both “struct llist_head”[44] and “struct llist_node”[45] are defined in the Linux source code at “/include/linux/llist.h”. Also, they both have a single field of type “struct llist_node” - as shown in the screenshot below.

Thus, although probably we could use the same data structure or use a union it is better to use different ones for different parts of the linked list. In case of a double linked list there is no choice cause the “next”/”prev” can point to a head or to a node[46].

Moreover, like with other linked list data structures (which are part of the Linux kernel) we use the “container_of” macro to get the “struct” holding of the “llist” entry[47]. For more operations on “llist” (like “llist_add_batch”, “llist_empty” and “llist_for_each_entry”) we can check out “/include/linux/llist.h”.

Lastly, “llist” is the lockless, NULL-terminated, singly-linked list implementation for the Linux kernel[48]. As of kernel version 6.5.8 “struct llist_node” is referenced in 97 code files[49].

```
struct llist_head {
        struct llist_node *first;
};

struct llist_node {
        struct llist_node *next;
};
```

[42] https://medium.com/@boutnaru/the-linux-kernel-data-strcture-journey-struct-llist-head-e6d33551c8fe
[43] https://medium.com/@boutnaru/the-linux-kernel-data-strcture-journey-struct-hlist-head-19f4ceb71295
[44] https://elixir.bootlin.com/linux/v6.5/source/include/linux/llist.h#L56
[45] https://elixir.bootlin.com/linux/v6.5/source/include/linux/llist.h#L56
[46] https://stackoverflow.com/questions/38748693/why-does-linux-kernel-lock-less-list-have-head-and-node-structs
[47] https://elixir.bootlin.com/linux/v6.5/source/include/linux/llist.h#L82
[48] https://drgn.readthedocs.io/en/latest/helpers.html
[49] https://elixir.bootlin.com/linux/v6.5.8/A/ident/llist_node

struct freelist_head

Overall, when managing the memory of our own application (like in cases we don't have garbage collection) it is an advantage to reuse objects instead of freeing them completely. Thus, when using dynamically allocated memory we can use the space of objects which are not needed anymore instead of allocating more memory - this concept is called a freelist[50]. The concept was adopted from a post titled "Solving the ABA Problem for Lock-Free Free Lists"[51].

Thus, "struct freelist_head" is the list head of a CAS (Compare-and-Swap) lock-free list - the semantics of a typical implementation of CAS is shown below[52]. Assuming nodes are never freed until after the free list is destroyed - it is simple and correct[53]. "struct freelist_head" has only one member called "head" which points to a "struct freelist_node"[54].

Lastly, as of kernel version 6.5.8 "struct freelist_head" is referenced in two header files beside its declaration[55]. It is used with "kretprobes"[56] and "rethooks" which a is a return hooking mechanism with list-based shadow stack[57]. By the way, since kernel version 6.7 "struct freelist_head" is not used any more[58] so the last relevant kernel version for it is 6.6.16[59].

Semantics of a typical implementation of CAS.

```
CAS (VAR var, expected, new)
{
    word baseval;
    atomic
        {
         baseval = var;
         if    (baseval == expected)
          {
              var  =  new;
          }
         }
    return baseval;
}
```

[50] https://academic-accelerator.com/encyclopedia/free-list
[51] https://moodycamel.com/blog/2014/solving-the-aba-problem-for-lock-free-free-lists
[52] https://www.embedded.com/is-lock-free-programming-practical-for-multicore/
[53] https://elixir.bootlin.com/linux/v6.5.8/source/include/linux/freelist.h#L10
[54] https://elixir.bootlin.com/linux/v6.5.8/source/include/linux/freelist.h#L23
[55] https://elixir.bootlin.com/linux/v6.5.8/A/ident/freelist_head
[56] https://medium.com/@boutnaru/linux-instrumentation-part-3-kretprobes-return-probes-bbbcacfd4289
[57] https://elixir.bootlin.com/linux/v6.6.16/source/include/linux/rethook.h#L3
[58] https://elixir.bootlin.com/linux/v6.7/A/ident/freelist_head
[59] https://elixir.bootlin.com/linux/v6.6.16/A/ident/freelist_head

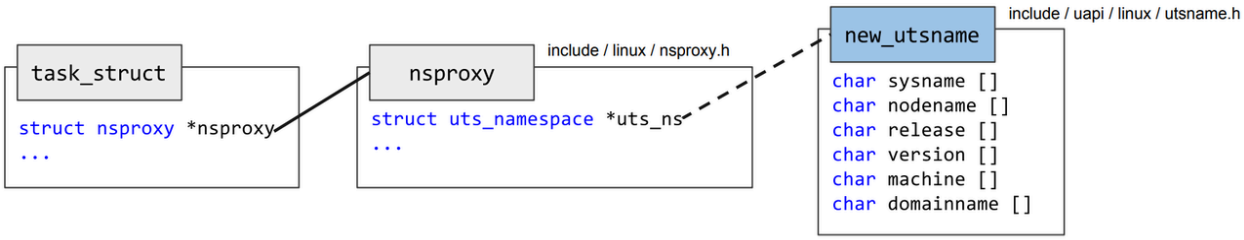
struct nsproxy

Overall, “struct nsproxy” (Namespace proxy) is a kernel data structure that contains a pointer to all per-process namespaces[60] like: mount (fs), uts, network, sysvipc and more. The PID namespace[61] is an exception because the field “pid_ns_for_children” is a pointer for the namespace information that the children will use[62]. By the way we can retrieve the PID namespace of a process/task using the function “task_active_pid_ns”[63].

Moreover, “struct nsproxy” is defined in “/include/linux/nsproxy.h”[64]. The “count” field contains the number of tasks holding a reference. “uts_ns” which holds a pointer to the information regarding the process UTS namespace[65]. “ipc_ns” which holds a pointer to the information regarding the process IPC namespace[66]. “mnt_ns” which holds a pointer to the information regarding the process mount namespace[67].

Also, “net_ns” which holds a pointer to the information regarding the process mount namespace[68]. “time_ns” which holds a pointer to the information regarding the process time namespace[69]. “time_ns_for_children” is a pointer for the time namespace information that the children will use. “cgroup_ns” which holds a pointer to the information regarding the process cgroup namespace[70].

Lastly, “struct nsproxy” is shared by tasks that share all namespaces. When a single namespace is cloned/unshared (like if using the clone/setns()/unshare() syscalls) the data structure is copied[71]. We can get to if from “struct task_struct” of the current process - as shown in the diagram below[72]. By the way, the user namespace information is stored in “struct cred”.



[60] https://systemweakness.com/linux-namespaces-part-1-dcee9c40fb68
[61] https://medium.com/@boutnaru/linux-namespaces-pid-namespace-e7e22f96ac3d
[62] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/nsproxy.h#L16
[63] https://elixir.bootlin.com/linux/v6.4.10/source/kernel/pid.c#L507
[64] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/nsproxy.h#L31
[65] https://medium.com/@boutnaru/linux-namespaces-uts-part-2-6073eacc82ae
[66] https://medium.com/@boutnaru/linux-namespaces-ipc-namespace-927f01cbcf3d
[67] https://medium.com/@boutnaru/linux-namespaces-mount-namespace-fca1e47d7a88
[68] https://medium.com/@boutnaru/linux-namespaces-network-namespace-part-3-7f8f8e06fef3
[69] https://medium.com/@boutnaru/linux-namespaces-time-namespace-part-3-1314b4c9cd32
[70] https://medium.com/@boutnaru/linux-cgroups-control-groups-part-1-358c636ffde0
[71] https://elixir.bootlin.com/linux/v6.4.10/source/include/linux/nsproxy.h#L27
[72] https://www.schutzwerk.com/en/blog/linux-container-namespaces05-kernel/

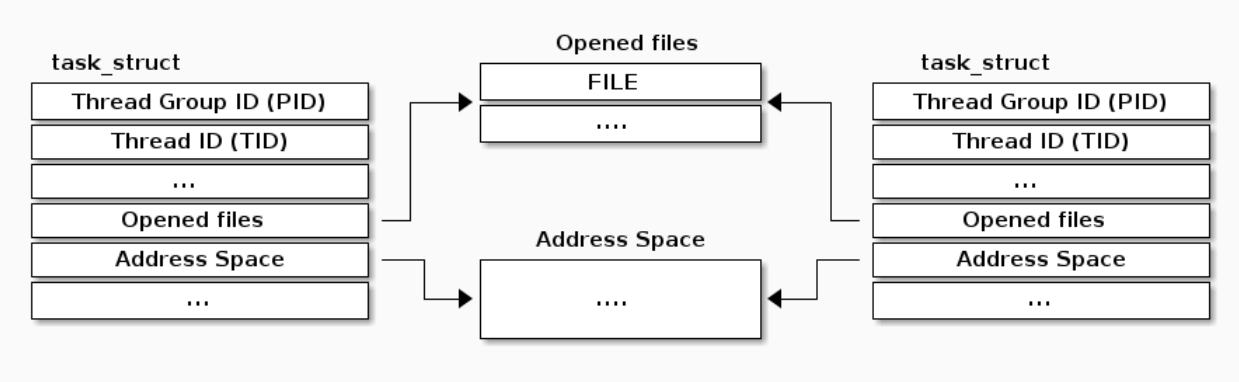
struct task_struct

Every operating system has a data structure that represents a "process[73] 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[74], kprobes[75] 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[76]. Also, fun fact is that in kernel 6.2-rc1 "task_strcut" is referenced in 1398 files[77].

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"[78]. 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[79].



[73] https://medium.com/@boutnaru/linux-processes-part-1-introduction-283f5b5b4197
[74] https://medium.com/system-weakness/linux-namespaces-part-1-dcee9c40fb68
[75] https://medium.com/@boutnaru/linux-instrumentation-part-2-kprobes-b089092c4cff
[76] https://elixir.bootlin.com/linux/v6.2-rc1/source/include/linux/sched.h#L737
[77] https://elixir.bootlin.com/linux/v6.2-rc1/A/ident/task_struct
[78] https://myaut.github.io/dtrace-stap-book/kernel/proc.html
[79] 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[80]. It is the user-mode part which belongs to the task/process[81]. By the way, until kernel version 2.6.23 the struct was defined under “/include/linux/sched.h”[82], since “2.6.24” it is located under “/include/linux/mm_types.h”[83]. Also, there is a pointer from the process’ “task_struct”[84] that refers to the address space of the process (“mm_struct”) which is stored in the “mm” field[85].

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[86]. Fun fact is that in kernel 6.2-rc1 “task_strcut” is referenced in 656 files[87]. Based on LXR[88] it seems that “mm_struct” was added from kernel version 1.1.11[89] as we don’t see it in previous versions[90].

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[91] than the number of rows in “/proc/[PID]/maps”.

[80] http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch14lev1sec1.html
[81] https://medium.com/@boutnaru/linux-memory-management-part-1-introduction-896f376d3713
[82] https://elixir.bootlin.com/linux/v2.6.23/source/include/linux/sched.h#L369
[83] https://elixir.bootlin.com/linux/v2.6.24/source/include/linux/mm_types.h#L156
[84] https://medium.com/@boutnaru/linux-kernel-task-struct-829f51d97275
[85] https://elixir.bootlin.com/linux/v6.2-rc1/source/include/linux/sched.h#L870
[86] https://elixir.bootlin.com/linux/v6.2-rc1/source/include/linux/mm_types.h#L601
[87] https://elixir.bootlin.com/linux/v6.2-rc1/C/ident/mm_struct
[88] https://elixir.bootlin.com
[89] https://elixir.bootlin.com/linux/1.1.11/source/include/linux/sched.h#L214
[90] https://elixir.bootlin.com/linux/1.1.10/A/ident/mm_struct
[91] It is due to the mechanism of vsyscall

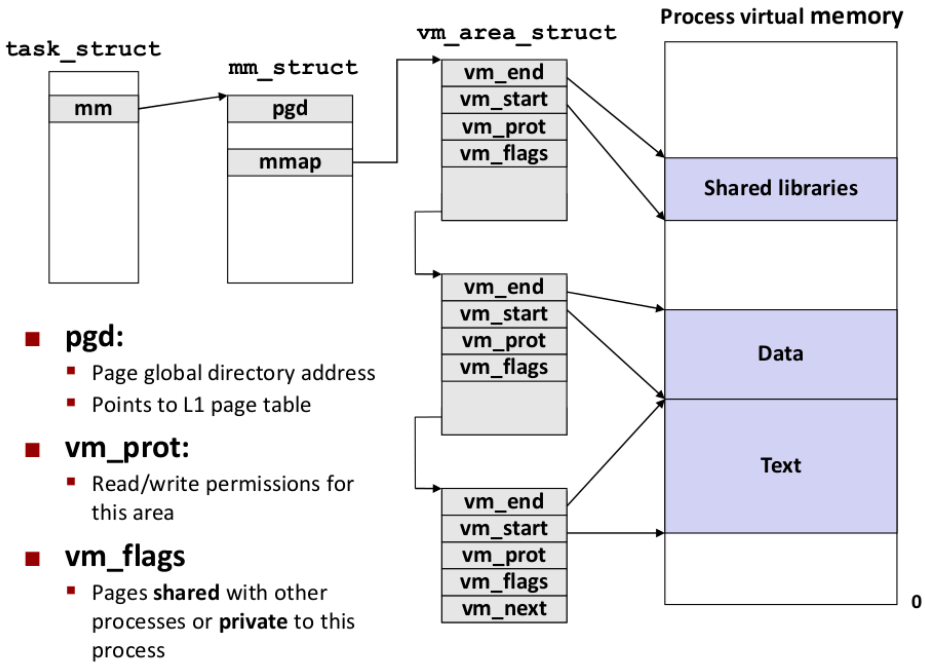
struct vm_area_struct

“vm_area_struct” represents a contiguous memory area in a process's address space (virtual memory areas) - as shown in the diagram below[92]. It is used to track the permissions, properties, and operations associated with each memory area[93].

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[94].

Until kernel version 2.6.21 (including) “struct vm_area_struct” is defined in “/include/linux/mm.h”[95]. From kernel versions about it is defined in “/include/linux/mm_types.h”[96].

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)[97].



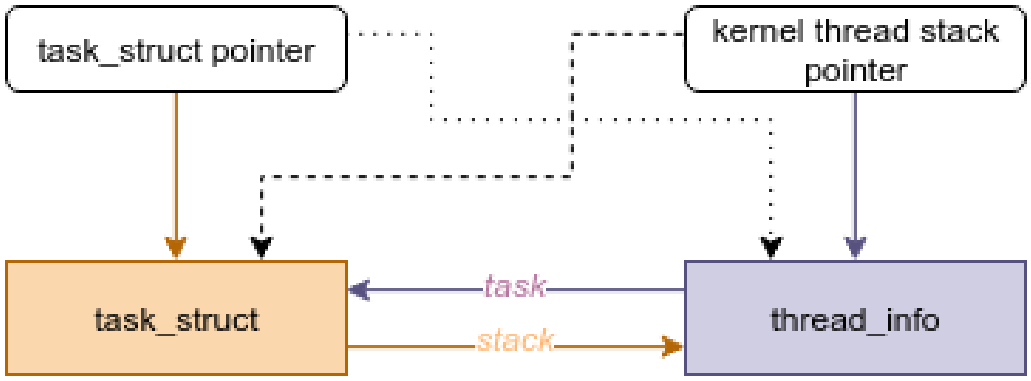
[92] https://don7hao.github.io/2015/01/28/kernel/mm_struct/
[93] http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch14lev1sec2.html
[94] https://elixir.bootlin.com/linux/v2.6.21/source/include/linux/mm.h#L55
[95] https://elixir.bootlin.com/linux/v2.6.21/source/include/linux/mm.h#L60
[96] https://elixir.bootlin.com/linux/v6.5-rc1/source/include/linux/m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

struct therad_info

“struct thread_info” is a common low-level thread information accessors[98] (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”[99]. 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”[100] - the old relationship is shown in the diagram below[101]. 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[102]. The first is done using the “current_thread_info” macro[103]. It uses “current_task” which is a per-cpu variable[104].

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”[105], “mips”[106], “riscv”[107], “arm64”[108] and more.



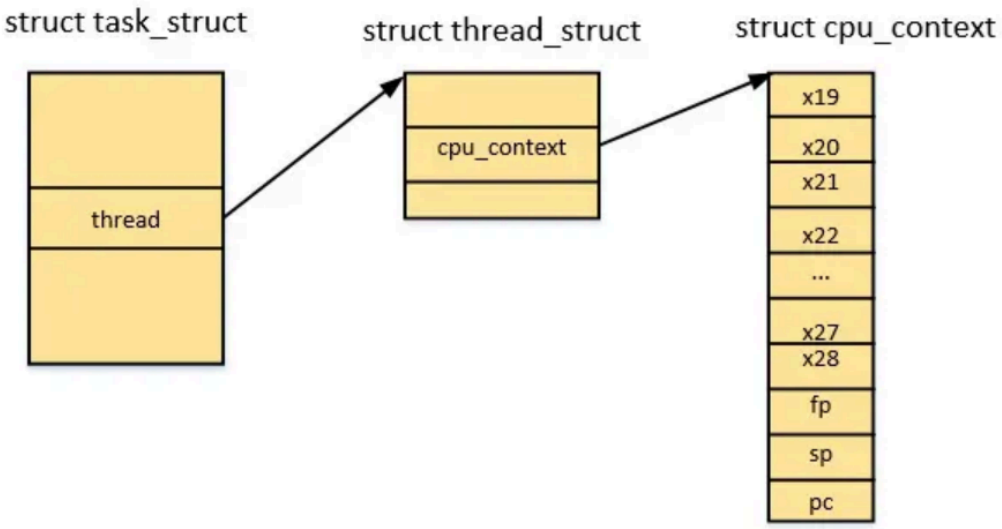
[98] https://elixir.bootlin.com/linux/v6.5-rc3/source/include/linux/thread_info.h#L2
[99] https://medium.com/@boutnaru/linux-kernel-task-struct-829f51d97275
[100] https://elixir.bootlin.com/linux/v4.8.17/source/arch/x86/include/asm/thread_info.h#L56
[101] https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part4.html
[102] https://stackoverflow.com/questions/61886139/why-thread-info-should-be-the-first-element-in-task-struct
[103] https://elixir.bootlin.com/linux/v6.5-rc3/source/include/linux/thread_info.h#L24
[104] https://elixir.bootlin.com/linux/v6.5-rc3/source/arch/x86/include/asm/current.h#L41
[105] https://elixir.bootlin.com/linux/v6.5-rc3/source/arch/x86/include/asm/thread_info.h#L56
[106] https://elixir.bootlin.com/linux/v6.5-rc3/source/arch/mips/include/asm/thread_info.h#L25
[107] https://elixir.bootlin.com/linux/v6.5-rc3/source/arch/riscv/include/asm/thread_info.h#L51
[108] 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[109]. 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[110].

On x86 the variable which is part of “struct task_struct”[111] must be at the end of the struct. The reason for that is it contain a variable-sized structure[112].

Moreover, like “struct thread_info”[113] 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”[114]. For example x86[115] and arm64[116].



[109] https://elixir.bootlin.com/linux/v6.5-rc4/source/include/linux/sched.h#L1540
[110] https://kernel.0voice.com/forum.php?mod=viewthread&tid=2920
[111] https://medium.com/@boutnaru/linux-kernel-task-struct-829f51d97275
[112] https://elixir.bootlin.com/linux/v6.5-rc4/source/include/linux/sched.h#L1544
[113] https://medium.com/@boutnaru/the-linux-kernel-data-structure-journey-struct-thread-info-4e70bc20d279
[114] https://elixir.bootlin.com/linux/v6.5-rc4/C/ident/thread_struct
[115] https://elixir.bootlin.com/linux/v6.5-rc4/source/arch/x86/include/asm/processor.h#L414
[116] https://elixir.bootlin.com/linux/v6.5-rc4/source/arch/arm64/include/asm/processor.h#L147