KERNEL ARCHITECTURE
AND THE
PROCESS DESCRIPTOR

Important Notice : Courseware - Legal

Table of Contents

Time Scale....4
The HZ Value....5
Background Information : the Task List....6
Stacks and the thread_info structure....6
What exactly is 'current'?....11
The Process Descriptor- the task_struct structure....18

Time Scale

From now onward, we shall occasionally come across statements such as - “the timer interrupt fires once every 10ms”, or, “a typical timeslice for a task is between 100 – 200ms”. For a modern computer, these time intervals are actually quite a bit – the system can achieve a lot in that time. To get a better “human feel” for such timings, consider the table below – *a quick “thought experiment”*:

Item	**Computer (actual) Time**	**Human Time** (scaled 2 billion times slower)
Processor Cycle	0.5 ns (2 GHz)	1 s
Cache Access	1 ns	2 s
Context Switch	19 us [1]	10.55 hours
Disk Access	7 ms (7,000,000 ns)	162 days
Timer “tick” (interrupt)	10 ms (10,000,000 ns)	7.5 months
Quantum (timeslice)	100 ms (100,000,000 ns)	6.3 years

(Table source: "UNIX Systems Programming", Robbins & Robbins)

Order of Magnitude:
While we're at it, we also often hear statements like “disk speed is easily five orders of magnitude slower than RAM”. What does “orders of magnitude” really mean? See this page for a simple explanation. (Very quick summary: 'n' orders of magnitude => 'n' powers of 10).

1 A good article on context-switching time on modern Intel processors: *How long does it take to make a context switch?* . Paraphrasing: “… So, what's the context-switch time? The author says in conclusion: “Context switching is expensive. My rule of thumb is that it'll cost you about 30µs of CPU overhead. This seems to be a good worst-case approximation.”
A Linux Journal article mentions an average switching time of 19 us.”

Have you ever asked yourself: when does the OS actually run??
See this article: [“When does your OS run?”, by Gustavo Duarte].

The HZ Value

“The 2.6 Linux kernel (on x86) sets up a timer interrupt to fire once every millisecond” [1]

Linux programs a timer chip, the Programmable Interval Timer (PIT- usually the 8254 chip on x86 motherboards), to issue a clock "tick" once every millisecond. How many clock ticks occur in one second? : this value is what the kernel variable **HZ** is tuned to.

Nowadays, HZ is now a kernel build-time tunable (CONFIG_HZ and variations).

The value of HZ is basically a function of:

1. the processor architecture
2. the kernel version.

[1] Note that continually running the ‘timer tick’ hardware interrupt continually is considered high overhead and is now unnecessary! With *HRT (High Resolution Timer)* support, the kernel does not need to do this; *from the kernel documentation*:
“Once a system has switched to high resolution mode (early in the boot process), the periodic tick is switched off. This disables the per system global periodic clock event device - e.g. the PIT on i386 SMP systems.
The periodic tick functionality is provided by an per-cpu hrtimer. The callback function is executed in the next event interrupt context and updates jiffies and calls update_process_times and profiling.

...”

(This is why the # of hardware interrupts on IRQ 0 (‘timer tick’) is typically low (output below from a 4 cpu x86_64 box):

```
$ w
 12:50:19 up 4 days,  1:39,  1 user,  load average: 1.79, 1.90, 1.99
...
$ grep "timer$" /proc/interrupts
   0:         10          0          0          0  IR-IO-APIC    2-edge      timer
$ grep "Local timer interrupts$" /proc/interrupts
 LOC:   44548113   40704708   41749178   41343538   Local timer interrupts
$
```
).

Background Information : the Task List

- The “process descriptor” data structure holds all relevant status information about the process (or thread)

- ***It is critical to understand that for every thread that is alive on the Linux OS, the kernel maintains a corresponding “task structure” (or - the mis-named - process descriptor).***

  In other words, the mapping between a userspace and/or kernel thread and a kernel-space task_struct is 1:1.

- All process descriptors, i.e., all task_struct's, are organized using a linked list; experience has shown that using a circular doubly-linked list works best. This list is called the “task list”.

- In fact, this scheme (of using circular linked lists) is so common in usage that it is built-in to the mainline kernel: a header called “list.h” has the data structure and macro elements to support building and manipulating sophisticated linked lists without re-inventing the wheel.

Stacks and the thread_info structure

The kernel maintains ***two stacks*** *(one for each privilege level) – a user-mode and a kernel-mode stack.*

Thus, for every thread alive on the system, we have two stacks:
- a usermode stack
- a kernel-mode stack.

(The exeception to the above rule: kernel threads. Kernel threads see *only* kernel virtual address space; thus, they require only a kernel-mode stack).

When a process (or thread) executes code in userspace, it is automatically using the usermode stack. When it issues a system call, it switches to kernel-mode; now, the CPU “automatically”* uses the kernel-mode stack for that process (or thread).

** This is usually done via microcode in the processor. See the end of the topic for an example (IA-32).*

Keep in mind that while the user space stack can grow very large (typically 8-10 MB resource limit), the kernel-mode stack is *very small :* typically less than one (or at most two) page frames.

Kernel Mode Stack Size on a few architectures

x86 (IA-32)	*x86_64*	*ARM*		*PowerPC*	
		arm32	*arm64*	*ppc32*	*ppc64*
8 KB	16 KB	8 KB	16 KB	8 KB	16 KB

<<
Besides kernel text and data, the kernel dynamically allocates and manages space for several meta-data structures and objects, among them the memory pools, kernel stacks, paging tables, etc.

On a laptop with 8GB RAM running Ubuntu 17.04 (x86_64):

```
$ uname -r
4.10.0-32-generic
$ egrep "KernelStack|PageTables" /proc/meminfo
KernelStack:        17120 kB
PageTables:         98756 kB
$
```

On an Android phone (Aarch64):

```
herolte:/ $ uname -r
3.18.14-11104523
herolte:/ $ egrep "KernelStack|PageTables" /proc/meminfo
KernelStack:        52752 kB
PageTables:         80648 kB
herolte:/ $
```

>>

Q. How can we tell how big the kernel mode stack is?

The size of the kernel-mode stack is expressed as the macro 'THREAD_SIZE'.
It is arch-dependant. Take a look at the code:

```
$ find arch/ -type f -name '*.[ch]' |xargs grep -Hn '#define.*THREAD_SIZE'
arch/arm64/include/asm/memory.h:109:#define THREAD_SIZE        (UL(1) << THREAD_SHIFT)
  << THREAD_SHIFT is 14 without KASAN; thus, THREAD_SIZE is 2^14 = 16384 >>
...
arch/sparc/include/asm/thread_info_64.h:107:#define THREAD_SIZE (2*PAGE_SIZE)
arch/sparc/include/asm/thread_info_64.h:110:#define THREAD_SIZE PAGE_SIZE
...
arch/score/include/asm/thread_info.h:14:#define THREAD_SIZE_ORDER      (1)
...
arch/powerpc/include/asm/thread_info.h:23:#define THREAD_SIZE          (1 << THREAD_SHIFT)
arch/powerpc/include/asm/thread_info.h:69:#define THREAD_SIZE_ORDER    (THREAD_SHIFT -
PAGE_SHIFT)
arch/arm/include/asm/page-nommu.h:15:#define KTHREAD_SIZE (8192)
arch/arm/include/asm/page-nommu.h:17:#define KTHREAD_SIZE PAGE_SIZE
arch/arm/include/asm/thread_info.h:18:#define THREAD_SIZE_ORDER 1
arch/arm/include/asm/thread_info.h:19:#define THREAD_SIZE              8192
arch/arm/include/asm/thread_info.h:20:#define THREAD_START_SP          (THREAD_SIZE - 8)
arch/avr32/include/asm/thread_info.h:13:#define THREAD_SIZE_ORDER      1
...
arch/x86/include/asm/page_32_types.h:19:#define THREAD_SIZE     (PAGE_SIZE << THREAD_ORDER)
  << THREAD_ORDER = 1, by default >>
```

```
arch/x86/include/asm/thread_info.h:175:#define STACK_WARN       (THREAD_SIZE/8)
arch/x86/include/asm/page_64_types.h:5:#define THREAD_SIZE  (PAGE_SIZE << THREAD_ORDER)
...
arch/alpha/include/asm/thread_info.h:56:#define THREAD_SIZE_ORDER 1
arch/alpha/include/asm/thread_info.h:57:#define THREAD_SIZE (2*PAGE_SIZE)
arch/blackfin/include/asm/thread_info.h:25:#define THREAD_SIZE_ORDER   1
arch/blackfin/include/asm/thread_info.h:26:#define THREAD_SIZE         8192   /* 2 pages */
...
$
```

Keep in mind that this size includes *both* the thread_info structure *and* the kernel-mode stack space.

Resource:
["Kernel Small Stacks" on eLinux]

Includes information on existing kernel stack monitoring mechanisms.

The thread_info structure

Besides the kernel-mode stack of the task, the kernel also maintains another structure per task called the **thread_info** structure. It is used to cache frequently referenced system data and provide a quick way to access the task_struct.

It's a small struct (just around 40 bytes on the x86); it should be as ideally it should fit into a single cache line.

[On the IA32, the 'esp' register points to the thread_info structure; on the ARM, it's the 'sp' register that points here.]

For example, on the x86 architecture:
In https://elixir.bootlin.com/linux/v4.6/source/arch/x86/include/asm/thread_info.h#L55 [v4.6]

```
...
struct thread_info {
        struct task_struct      *task;          /* main task structure */
        __u32                   flags;          /* low level flags */
        __u32                   status;         /* thread synchronous flags */
        __u32                   cpu;            /* current CPU */
        mm_segment_t            addr_limit;
        unsigned int            sig_on_uaccess_error:1;
        unsigned int            uaccess_err:1;  /* uaccess failed */
};
...
```

and on the ARM architecture:

In https://elixir.bootlin.com/linux/v4.6/source/arch/arm/include/asm/thread_info.h#L49

```
...
/*
 * low level task data that entry.S needs immediate access to.
```

```
 * __switch_to() assumes cpu_context follows immediately after cpu_domain.
 */
struct thread_info {
        unsigned long           flags;          /* low level flags */
        int                     preempt_count;  /* 0 => preemptable, <0 => bug */
        mm_segment_t            addr_limit;     /* address limit */
        struct task_struct      *task;          /* main task structure */
        __u32                   cpu;            /* cpu */
        __u32                   cpu_domain;     /* cpu domain */
        struct cpu_context_save cpu_context;    /* cpu context */
        __u32                   syscall;        /* syscall number */
        __u8                    used_cp[16];    /* thread used copro */
        unsigned long           tp_value[2];    /* TLS registers */
...
```

FYI, on the ARM64:
https://elixir.bootlin.com/linux/v5.0/source/arch/arm64/include/asm/thread_info.h#L39

```
/*
 * low level task data that entry.S needs immediate access to.
 */

struct thread_info {
        unsigned long           flags;          /* low level flags */
        mm_segment_t            addr_limit;     /* address limit */
#ifdef CONFIG_ARM64_SW_TTBR0_PAN        << PAN- Privileged Access Never; don't allow kernel
                                                access to userspace >>
        u64                     ttbr0;          /* saved TTBR0_EL1 */
#endif
        union {
                u64             preempt_count;  /* 0 => preemptible, <0 => bug */
                struct {
#ifdef CONFIG_CPU_BIG_ENDIAN
                        u32     need_resched;
                        u32     count;
#else
                        u32     count;
                        u32     need_resched;
#endif
                } preempt;
        };
};
```

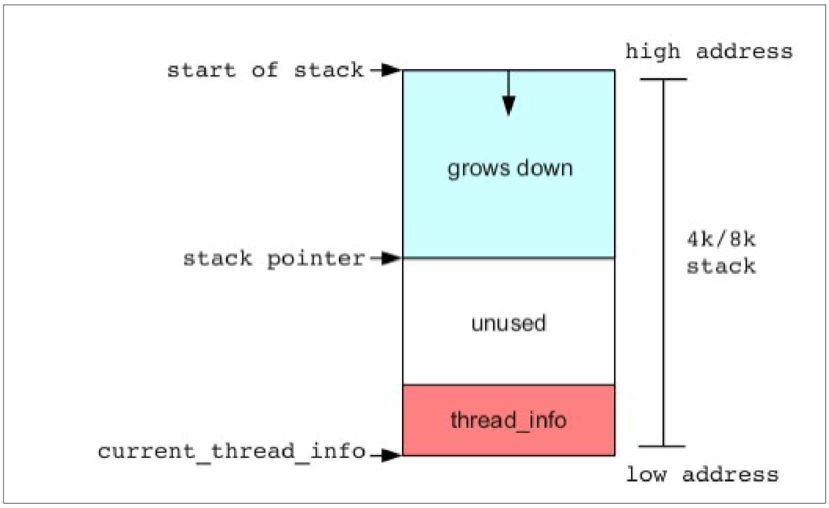
The thread info struct and kernel-mode stack are clubbed together in either a single or two contiguous physical memory pages.

In fact, fyi, on the 3.2.11 kernel codebase:

```
union thread_union {
#ifndef  CONFIG_THREAD_INFO_IN_TASK
        struct thread_info thread_info;
```

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

Diagramatically (*source*):



Note though that from 2.6 Linux onward, each kernel mode thread stack is either 2 pages (8k; 32-bit systems) or 4 pages (16k; 64-bit systems).

What exactly is 'current'?

For ease of understanding, just think of 'current' as a pointer to the task structure of the task that's currently running on the CPU core (in question) (analogous to “self”). So, for example, to look up the PID of the currently executing task, just do:
`current->pid`

or the thread name with:
`current->comm`

In reality, 'current' is a macro: it's definition is implementation- (meaning, architecture) dependant.

On the x86, as well as ARM platforms, current is defined as a macro (in assembly language) that locates the task structure by taking an appropriate size offset from the top of the (kernel-mode) stack; this is because (you recall that) the thread_info structure and kernel-mode stack are clubbed together in either a single or two contiguous physical memory pages, and the task_struct pointer is within the thread_info structure.

On the ARM platform (kernel ver 3.2): *From*
http://lxr.free-electrons.com/source/arch/arm/include/asm/current.h?v=3.2

```
...
static inline struct task_struct *get_current(void)
{
        return current_thread_info()->task;
}

#define current (get_current())


and from
```

http://lxr.free-electrons.com/source/arch/arm/include/asm/thread_info.h?v=3.2#L94

```
[...]
/*
 * how to get the thread information struct from C
 */
static inline struct thread_info *current_thread_info(void) __attribute_const__;

static inline struct thread_info *current_thread_info(void)
{
        register unsigned long sp asm ("sp");
        return (struct thread_info *)(sp & ~(THREAD_SIZE - 1));
}
...
```

<<
Explanation of above:
With THREAD_SIZE (meaning, kernel-mode stack size) = 4096 :

'sp' is the register (normally r13) holding the head (start) of the stack.

If THREAD_SIZE = 4096, then (above), return value of current_thread_info

= sp & ~4095
= sp & ~(0000 0000 0000 0000 0000 1111 1111 1111)
= sp & (1111 1111 1111 1111 1111 0000 0000 0000)
=> low 12-bits of address are zeroed out, which is equivalent to truncating it to the nearest (numerically lower) page boundary. This, in effect, yields the pointer to the thread_info structure (as the ti will be placed in the beginning of the page frame that holds both the ti structure and the kernel-mode stack)!

<<
With THREAD_SIZE (meaning, kernel-mode stack size) = 8192 :

'sp' is the register (normally r13) holding the head (start) of the stack.
If THREAD_SIZE = 8192, then (above), return value of current_thread_info

= sp & 8191
= sp & ~(0000 0000 0000 0000 0001 1111 1111 1111)
= sp & (1111 1111 1111 1111 1110 0000 0000 0000)
=> low 13-bits of address are zeroed out, which is equivalent to truncating it to the nearest (numerically lower) *two* page boundary. This, in effect, yields the pointer to the thread_info structure (as the ti will be placed in the beginning of the page frame that holds both the ti structure and the kernel-mode stack)!
>>

This is looked up (below, in inline function `get_current()`) with offset 'task', which yields the location of the task_struct.
>>

Sidebar | *'current' on ARM-32 on (a recent) Linux v 4.10.0-rc2 (as of 06 Jan 2017)*

include/asm-generic/current.h

```
#define get_current() (current_thread_info()->task)
#define current get_current()
```

arch/arm/include/asm/thread_info.h

```
/*
 * how to get the current stack pointer in C
 */
register unsigned long current_stack_pointer asm ("sp");

[...]

/*
 * how to get the thread information struct from C
 */
static inline struct thread_info *current_thread_info(void) __attribute_const__;

static inline struct thread_info *current_thread_info(void)
{
    return (struct thread_info *)
        (current_stack_pointer & ~(THREAD_SIZE - 1));
}
<< essentially identical to the earlier code >>
```

ARM64

Sidebar | *'current' on ARM64 on latest Linux v 4.10.0-rc2 (as of 06 Jan 2017)*

As a measure towards OS hardening (becoming critical nowadays), Mark Rutland's (of ARM) [patch entitled "arm64: split thread_info from task stack"] has been merged into mainline (recently: v4.9-rc4, 04 Nov 2016: so it's now in 4.10 onward). The commit explains:

"This patch moves arm64's struct thread_info from the task stack into task_struct. This protects thread_info from corruption in the case of stack overflows, and makes its address harder to determine if stack addresses are leaked, making a number of attacks more difficult. Precise detection and handling of overflow is left for subsequent patches.

Largely, this involves changing code to store the task_struct in sp_el0, and acquire the thread_info from the task struct. Core code now implements current_thread_info(), and as noted in <linux/sched.h> this relies on offsetof(task_struct, thread_info) == 0, enforced by core code.
[...]"

Code View:
include/linux/thread_info.h

```
[...]
#ifdef CONFIG_THREAD_INFO_IN_TASK    << will be true on recent ARM64 >>
/*
```

```
 * For CONFIG_THREAD_INFO_IN_TASK kernels we need <asm/current.h> for the
 * definition of current, but for !CONFIG_THREAD_INFO_IN_TASK kernels,
 * including <asm/current.h> can cause a circular dependency on some platforms.
 */
#include <asm/current.h>
#define current_thread_info() ((struct thread_info *)current)
#endif
[...]

arch/arm64/include/asm/current.h

/*
 * We don't use read_sysreg() as we want the compiler to cache the value where
 * possible.
 */
static __always_inline struct task_struct *get_current(void)
{
        unsigned long sp_el0;

        asm ("mrs %0, sp_el0" : "=r" (sp_el0));   << mrs: read from register;
         Above, the inline assembly reads from sp_el0 into X0 and updates the
          variable 'sp_el0'
                                                    msr: write to register >>
        return (struct task_struct *)sp_el0;
}

#define current get_current()
```

The arch-dependant *entry_task_switch()* code will ensure that the sp_e10 register is updated to 'next' every time (we're about to) context-switch.

(*Details: pl see* ['[v2] arm64: Introduce IRQ stack'](#) and https://stackoverflow.com/questions/29393677/armv8-exception-vector-significance-of-el0-sp).

The above arm32 code for obtaining *current* might at first look very optimized and cool; kernel (and hardware) folks beg to differ! Check this out [*source*:arm64: Introduce IRQ stack *[patch]*]:

"…

```
It is a core concept to directly retrieve struct thread_info from
sp_el0. This approach helps to prevent text section size from being
increased largely as removing masking operation using THREAD_SIZE
in tons of places.

[Thanks to James Morse for his valuable feedbacks which greatly help
to figure out a better implementation. - Jungseok]

...

+/*
+ * struct thread_info can be accessed directly via sp_el0.
+ */

 static inline struct thread_info *current_thread_info(void)
 {
```

```
-       return (struct thread_info *)
-               (current_stack_pointer & ~(THREAD_SIZE - 1));
+       unsigned long sp_el0;
+
+       asm ("mrs %0, sp_el0" : "=r" (sp_el0));   << mrs: read from register;
          Above, the inline assembly reads from sp_el0 into X0 and updates the
          variable 'sp_el0'
                                                      msr: write to register >>
+
+       return (struct thread_info *)sp_el0;
 }
 ...
```

Sept 2016: v4.9-rc1:
sched/core: Allow putting thread_info into task_struct
commit c65eacbe290b8141554c71b2c94489e73ade8c8d

sched/core: Allow putting thread_info into task_struct

If an arch opts in by setting `CONFIG_THREAD_INFO_IN_TASK_STRUCT`,
then thread_info is defined as a single 'u32 flags' and is the first
entry of task_struct. thread_info::task is removed (it serves no
purpose if thread_info is embedded in task_struct), and
thread_info::cpu gets its own slot in task_struct.
This is heavily based on a patch written by Linus.

Originally-from: Linus Torvalds <torvalds@linux-foundation.org>

Signed-off-by: Andy Lutomirski <luto@kernel.org>

...

Currently implemented on x86_64 and ARM64. Others?

Preventing kernel-stack leaks, LWN, 07 Mar 2018, Corbet
… the combination of `STACKLEAK, VMAP_STACK` (providing the guard pages) and `THREAD_INFO_IN_TASK` protects the kernel against known stack depth overflow attacks. …

*Trying to get **STACKLEAK** into the kernel, LWN, Sept 2018.*

Sidebar | *'current' on x86_64 on (latest) Linux v 4.10.0-rc2 (as of 06 Jan 2017)*

The modern x86 implementation (from 2.6.30) uses *per-cpu variables*; another very efficient way by which 'current' can be stored and accessed.

arch/x86/include/asm/current.h

```
DECLARE_PER_CPU(struct task_struct *, current_task);

static __always_inline struct task_struct *get_current(void)
{
    return this_cpu_read_stable(current_task);
}
```

```
#define current get_current()
...
```

On the other hand, on the PPC architecture implementation, the address of the task_struct is stored as part of hardware context (in a CPU register); this makes the lookup extremely efficient.

FYI / Note-

- *Recent: [28 Sept 2019] : 5.4 Linux kernel: [Src]*

  “Merge branch 'next-lockdown' of git://git.kernel.org/pub/scm/linux/kernel/git/jmorris/linux-security

```
Pull kernel lockdown mode from James Morris:
"This is the latest iteration of the kernel lockdown patchset, from Matthew Garrett,
David Howells and others. From the original description: This patchset introduces an
optional kernel lockdown feature, intended to strengthen the boundary between UID 0 and
the kernel. When enabled, various pieces of kernel functionality are restricted.
Applications that rely on low-level access to either hardware or the kernel may cease
working as a result - therefore this should not be enabled without appropriate
evaluation beforehand.
...
The new locked_down LSM hook is provided to allow LSMs to make a policy decision
around whether kernel functionality that would allow tampering with or examining
the runtime state of the kernel should be permitted. The included lockdown LSM
provides an implementation with a simple policy intended for general purpose use.
This policy provides a coarse level of granularity, controllable via the kernel
command line: lockdown={integrity|confidentiality} Enable the kernel lockdown
feature. If set to integrity, kernel features that allow userland to modify the
running kernel are disabled. If set to confidentiality, kernel features that allow
userland to extract confidential information from the kernel are also
disabled. ..."
```

- *From 4.9-rc1, we have a new kernel config directive – CONFIG_VMAP_STACK. This allocates and uses a (kernel-mode) stack from the vmalloc region, and with guards - importantly, this greatly helps cleanly catch and report kernel-mode stack overflows. Currently supported on x86_64 and ARM64.*

Here’s the commit if interested: [fork: Add generic vmalloced stack support].
From arch/Kconfig :

```
...
config HAVE_ARCH_VMAP_STACK
    def_bool n
    help
      An arch should select this symbol if it can support kernel stacks
      in vmalloc space.  This means:

      - vmalloc space must be large enough to hold many kernel stacks.
        This may rule out many 32-bit architectures.
```

```
        - Stacks in vmalloc space need to work reliably.  For example, if
          vmap page tables are created on demand, either this mechanism
          needs to work while the stack points to a virtual address with
          unpopulated page tables or arch code (switch_to() and
          switch_mm(), most likely) needs to ensure that the stack's page
          table entries are populated before running on a possibly
          unpopulated stack.

        - If the stack overflows into a guard page, something reasonable
          should happen.  The definition of "reasonable" is flexible, but
          instantly rebooting without logging anything would be unfriendly.

config VMAP_STACK
    default y
    bool "Use a virtually-mapped stack"
    depends on HAVE_ARCH_VMAP_STACK && !KASAN
    ---help---
      Enable this if you want the use virtually-mapped kernel stacks
      with guard pages.  This causes kernel stack overflows to be
      caught immediately rather than causing difficult-to-diagnose
      corruption.

      This is presently incompatible with KASAN because KASAN expects
      the stack to map directly to the KASAN shadow map using a formula
      that is incorrect if the stack is in vmalloc space.
```

Miscellaneous / FYI

You will often see the “container_of” macro being used in kernel code. What does it mean, how does it work?

Understanding container_of macro in linux kernel : on SO

container_of() macro usage in Kernel

In general, see:
MagicMacros on kernelnewbies : container_of() and ARRAY_SIZE().

Interesting:
The kernelnewbies FAQ page

> What does !!(x) mean in C (esp. the Linux kernel)?
> /ContainerOf What is *container_of* ? How does it work ?
> /DoWhile0 Why do a lot of #defines in the kernel use do { ... } while (0)?
>
> What does !!(x) mean in C (esp. the Linux kernel)?
>
> *Ans: “`!!(x)` forces it to be either 0 or 1. 0 remains 0, but any non-zero value (which would be 'true' in a boolean context) becomes 1.”, Paul Tomblin*
>
> etc etc.

The Process Descriptor- the task_struct structure

The structure **task_struct** represents a Linux task. It is called the **process descriptor**.

The task_struct structure (defined in *include/linux/sched.h*):

A powerful source-level debugger for the Linux kernel is KGDB.
Here, we make use of the sophisticated KGDB interactive kernel debugger tool to look up the task_struct of a process. What follows in this (blue) colour are some extracts from the task_struct of a process, helping us to actually "see" some of it's members that are relevant to our discussion.

We do this by setting up a breakpoint in the kernel code that creates a (child) process and look up the task_struct from within the debugger (gdb).

[Also Note:

- *The details of KGDB setup/installation and usage are covered in the "LINUX Debugging Techniques" training].*
- *An alternative to using KGDB is to use KDB.*
- *[UPDATE!]*
  *Still another tool, (perhaps the best in terms of analysis capabilities) is the kexec/kdump facility in conjunction with the **crash utility**. Crash lets one look up detailed data structure, stack, memory, machine state, etc information.*
- *Shown below is sample output from tracing parts of the (now old) 2.6.17 kernel built with **kgdb support** on an IA-32 system.*
  ]

```
(gdb) info b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0xc01189fb in panic at kernel/panic.c:76
2   breakpoint     keep y   0xc0177a41 in sys_sync at fs/sync.c:41
        breakpoint already hit 2 times
3   breakpoint     keep n   0xc01069dd in timer_interrupt at arch/i386/kernel/time.c:161
        breakpoint already hit 4 times
(gdb) b do_fork
Breakpoint 8 at 0xc0117ed2: file kernel/fork.c, line 1358.
(gdb) c
Continuing.
```

<< Now, during our kgdb session, on the target system's shell type 'ps' (or any executable, in fact) >>

```
Breakpoint 8, do_fork (clone_flags=0x1200011, stack_start=0xbffa4d58, regs=0xc43a0fb8, stack_size=0x0,
```

```
    parent_tidptr=0x0, child_tidptr=0xb7fba708) at kernel/fork.c:1358
1358    {
(gdb) l
1353                  unsigned long stack_start,
1354                  struct pt_regs *regs,
1355                  unsigned long stack_size,
1356                  int __user *parent_tidptr,
1357                  int __user *child_tidptr)
1358    {
1359            struct task_struct *p;
1360            int trace = 0;
1361            struct pid *pid = alloc_pid();
1362            long nr;
(gdb) b 1378
Breakpoint 9 at 0xc0117f3b: file kernel/fork.c, line 1378.
(gdb) c
Continuing.

Breakpoint 9, do_fork (clone_flags=0x1200011, stack_start=0xbffa4d58, regs=<value optimized out>,
    stack_size=0x0, parent_tidptr=0x0, child_tidptr=0xb7fba708) at kernel/fork.c:1381
1381                    if (clone_flags & CLONE_VFORK) {
...
```

With crash:

<< running on an x86_64 >>

```
crash> set    ← set: show task in context
    PID: 28520                    task_struct ptr       thread_info ptr
COMMAND: "mmap_file_rw"
   TASK: ffff880094dd5180  [THREAD_INFO: ffff880102498000]
    CPU: 0
  STATE: TASK_INTERRUPTIBLE
crash>
crash> task_struct      ← displays the structure definition
struct task_struct {
    volatile long state;
    void *stack;
    atomic_t usage;
    unsigned int flags;
    unsigned int ptrace;

--snip--

crash> task           ← displays the structure contents
PID: 28520  TASK: ffff880094dd5180  CPU: 0   COMMAND: "mmap_file_rw"
struct task_struct {
  state = 1,
  stack = 0xffff880102498000,
  usage = {
    counter = 2
  },
  flags = 4218880,
  ptrace = 0,
  wake_entry = {
    next = 0x0
  },
  on_cpu = 0,
...
```

Some Points to Note-

- *Note- easy and search-able browsing (of basically any version) of the Linux kernel source tree can be done online:*
  - *using the LXR (Linux Cross Reference) tool: https://elixir.bootlin.com/linux/latest/source*
  - *mainline [Linux kernel source tree on Github](); owned by Torvalds*

- *The Linux kernel is a (very fast!) moving target. Therefore, the material below is bound to get outdated. The only way to "keep up" with the latest kernel source is to install git, clone and regularly pull in the latest version (see the 'xtra' material on using git).*

- *The text within the "<<" and ">>" below are comments or further information introduced by this author into the material for better understanding and are not part of the actual task_struct structure source.*

<< Below: (mostly) as of Linux kernel ver ***5.0.3*** *[Mar 2019] >>*
include/linux/sched.h

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK        << Recent: 4.9. Commit (now thread_info is just
                                                     one 32-bit 'flags' field) >>
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	volatile long state;	/* -1 unrunnable, 0 runnable, >0 stopped */
<<
```

- state: This can be one of the following defines that appear higher up in *sched.h* :

```
	...
/*
 * Task state bitmask. NOTE! These bits are also
 * encoded in fs/proc/array.c: get_task_state().
 *
 * We have two separate sets of flags: task->state
 * is about runnability, while task->exit_state are
 * about the task exiting. Confusing, but this way
 * modifying one set can't modify the other one by
 * mistake.
 */
#define TASK_RUNNING		0
#define TASK_INTERRUPTIBLE	1
#define TASK_UNINTERRUPTIBLE	2
#define __TASK_STOPPED		4
#define __TASK_TRACED		8
/* in tsk->exit_state */
#define EXIT_ZOMBIE		16
#define EXIT_DEAD		32
/* in tsk->state again */
#define TASK_DEAD		64
#define TASK_WAKEKILL		128
#define TASK_WAKING		256
#define TASK_STATE_MAX		512
	...
/* Convenience macros for the sake of set_task_state */
#define TASK_KILLABLE		(TASK_WAKEKILL | TASK_UNINTERRUPTIBLE) 204#define
TASK_STOPPED		(TASK_WAKEKILL | __TASK_STOPPED) 205#define TASK_TRACED
(TASK_WAKEKILL | __TASK_TRACED)

/* Convenience macros for the sake of wake_up */
#define TASK_NORMAL (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)
#define TASK_ALL (TASK_NORMAL | __TASK_STOPPED | __TASK_TRACED)

...
```

- type volatile implies that this member can be altered asynchronously from interrupt routines.
  << Good ref: http://www.netrino.com/Embedded-Systems/How-To/C-Volatile-Keyword >>

- *<< 2.6.25 – new feature; see* ***http://kernelnewbies.org/LinuxChanges*** *>>*
  Most Unix systems have two states when sleeping -- interruptible and uninterruptible. 2.6.25 adds a third state: **killable**. While interruptible sleeps can be interrupted by any signal, killable sleeps can only be interrupted by fatal signals. The practical implications of this feature is that NFS has been converted to use it, and as a result you can now kill -9 a task that is waiting for an NFS server that isn't contactable. Further uses include allowing the OOM killer to make better decisions (it can't kill a task that's sleeping uninterruptibly) and changing more parts of the kernel to use the killable state. ...

>>

```
<<
(gdb) p p                          << print 'p' <-- the new child's task_struct .
                                      As of now, it's a copy of the parent: bash >>
$8 = (struct task_struct *) 0xc3d61510
(gdb) p p.state
$12 = 0x0
(gdb) p /x p.flags
$13 = 0x400040
(gdb) p *p                                   << lets look it up >>
$9 = {state = 0x0, thread_info = 0xc2140000, usage = {counter = 0x2}, flags = 0x400040, ptrace = 0x0,
  lock_depth = 0xffffffff, load_weight = 0x80, prio = 0x73, static_prio = 0x78, normal_prio = 0x73,
run_list = {
    next = 0xc3d61538, prev = 0xc3d61538}, array = 0x0, ioprio = 0x0, sleep_avg = 0x35a4e900,
  timestamp = 0x3c1fc3738de, last_ran = 0x3c1fc34be7e, sched_time = 0x0, sleep_type = SLEEP_NORMAL,
  policy = 0x0, cpus_allowed = {bits = {0x1}}, time_slice = 0xc, first_time_slice = 0x1, tasks = {
    next = 0xc03c9d68, prev = 0xc3fe80d8}, ptrace_children = {next = 0xc3d61580, prev = 0xc3d61580},
...
(gdb)
(gdb) set print pretty
(gdb) p *p
$10 = {
  state = 0,
  stack = 0xd6ba8000,
  usage = {
    counter = 2
  },
  flags = 4202562,
  ptrace = 0,
  wake_entry = 0x0,
  on_cpu = 0,
  on_rq = 0,
  prio = 120,
  static_prio = 120,
  normal_prio = 120,
  rt_priority = 0,
  sched_class = 0xc159b420,
```

```
  se = {
    load = {
      weight = 1024,
      inv_weight = 4194304
    },
…
…
    memcg = 0x0,
    nr_pages = 0,
    memsw_nr_pages = 0
  },
  ptrace_bp_refcnt = {
    counter = 1
  }
}
```

>>

<<

From *include/linux/sched//signal.h* :

```
static inline int signal_pending(struct task_struct *p)
{
        return unlikely(test_tsk_thread_flag(p,TIF_SIGPENDING));
}
```

TIF_SIGPENDING is one of the flags inside the task's thread_info structure; it is used to detect if a signal is pending delivery upon the task. The above inline function returns True if a signal is pending, False otherwise.

<<

Ref:

[likely()/unlikely() macros in the Linux kernel - how do they work? What's their benefit?]()

[Why do we use __builtin_expect when a straightforward way is to use if-else]()

```
long __builtin_expect (long exp, long c)
Semantics: it's expected that exp == c

#define likely(x)    __builtin_expect((x),1)
#define unlikely(x)  __builtin_expect((x),0)
```

>>

<<

'[un]likely' are compiler optimization attributes; the programmer can provide a hint to the compiler regarding branch prediction via these statements.

The "[un]likely" compiler attributes will actually affect the code generation of the code where it's called from; this way we try and avoid getting off "hot" code paths.. We optimize towards the 'hot' path; will pay a performance penalty if the hint is wrong. But that's unlikely by definition!

See on kernelnewbies FAQ: likely() and unlikely()

Interestingly, **glibc** also uses them!
/usr/include/sys/cdefs.h:

```
…
#if __GNUC__ >= 3
# define __glibc_unlikely(cond) __builtin_expect ((cond), 0)
# define __glibc_likely(cond)   __builtin_expect ((cond), 1)
#else
# define __glibc_unlikely(cond) (cond)
# define __glibc_likely(cond)   (cond)
#endif
...
```

>>

<<

*Related: "… **What is the difference between terms: "Slow path" and "Fast path" ?***

In general, "fast path" is the commonly run code that should finish very quickly. For example, when it comes to spinlocks the fast path is that nobody is holding the spinlock and the CPU that wants it can just take it.

Conversely, the slow path for spinlocks is that the lock is already taken by somebody else and the CPU will have to wait for the lock to be freed.

The first case is the common one that should be optimized.

The second one is not as important and does not need to be optimized much at all.

In this example, the reason for not optimizing the spinlock code for dealing with lock contention is that locks should not be contended. If they are, we need to redesign the data structures or the code to avoid contention in the first place!

You will see similar tradeoffs in the page locking code, the scheduler code (common cases are fast, unlikely things are put out of line by the compiler and are "behind a jump") and many other places in the kernel."

>>

<<

Static Keys and Jump Labels in the Linux Kernel

Motivation: to avoid as much as is possible getting off the 'hot path', yet support 'unlikely-to-come-true' if conditions within a performance-sensitive kernel code path (a good example is the kernel tracepoint code; have to check conditionally 'is the tracepoint enabled' every time; static keys optimize this check!).

Source: https://www.kernel.org/doc/Documentation/static-keys.txt

```
...
Static keys allows the inclusion of seldom used features in
performance-sensitive fast-path kernel code, via a GCC feature and a code
patching technique. A quick example::
```

```
        DEFINE_STATIC_KEY_FALSE(key);

        ...

        if (static_branch_unlikely(&key))
                do unlikely code
        else
                do likely code

        ...
        static_branch_enable(&key);
        ...
        static_branch_disable(&key);
        ...
```

The static_branch_unlikely() branch will be generated into the code with as little impact to the likely code path as possible.

```
...
>>
```

[More on leveraging GCC within the kernel!]()

These are the *thread_info* flags (defined in *arch/x86/include/asm/thread_info.h*):

```
#define TIF_SYSCALL_TRACE       0       /* syscall trace active */
#define TIF_NOTIFY_RESUME       1       /* resumption notification
                                           requested */
#define TIF_SIGPENDING          2       /* signal pending */
#define TIF_NEED_RESCHED        3       /* rescheduling necessary */
#define TIF_SINGLESTEP          4       /* restore singlestep on return
                                           to user mode */
#define TIF_IRET                5       /* return with iret */
#define TIF_SYSCALL_AUDIT       7       /* syscall auditing active */
#define TIF_POLLING_NRFLAG      16      /* true if poll_idle() is polling
                                           TIF_NEED_RESCHED */

>>
...
```

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

        void *stack;
<< In dup_task_struct:
        ti = alloc_thread_info(tsk);
        ...
        tsk->stack = ti;  << 'ti' is the memory for the kernel-mode stack
                                    and thread_info structure >>
>>
        atomic_t usage;
        /* Per task flags (PF_*), defined further below: */
        unsigned int                    flags;
```

```
<<
/*
 * Per process flags
 */
#define PF_ALIGNWARN    0x00000001      /* Print alignment warning msgs */
                                        /* Not implemented yet, only for 486*/
#define PF_STARTING     0x00000002      /* being created */
#define PF_EXITING      0x00000004      /* getting shut down */
#define PF_EXITPIDONE   0x00000008      /* pi exit done on shut down */
#define PF_VCPU         0x00000010      /* I'm a virtual CPU */
#define PF_FORKNOEXEC   0x00000040      /* forked but didn't exec */
#define PF_SUPERPRIV    0x00000100      /* used super-user privileges */
#define PF_DUMPCORE     0x00000200      /* dumped core */
#define PF_SIGNALED     0x00000400      /* killed by a signal */
#define PF_MEMALLOC     0x00000800      /* Allocating memory */
#define PF_FLUSHER      0x00001000      /* responsible for disk writeback */
...
>>
 unsigned int ptrace;
<<
 usage = {
 counter = 0x2
 },
 flags = 0x400040,
 ptrace = 0x0,
>>

        ...
```

<< Several members that follow relate directly to the scheduler; seen later >>

```
        int                             on_rq;
        int                             prio;
        int                             static_prio;
        int                             normal_prio;
        unsigned int                    rt_priority;
<--  >= 2.6.23 : the CFS scheduler >
        const struct sched_class        *sched_class;
        struct sched_entity             se;
        struct sched_rt_entity          rt;

<<
  prio = 0x73,
  static_prio = 0x78,
  normal_prio = 0x73,
>>

...
      unsigned int                      policy;
```

<<

scheduling policy: one of:

SCHED_NORMAL or SCHED_OTHER (default non real-time);

SCHED_RR, SCHED_FIFO ((soft) real-time),
[SCHED_BATCH, SCHED_ISO (not implemented yet), SCHED_IDLE]
>>

```
        int                         nr_cpus_allowed;
        cpumask_t                   cpus_allowed;        << CPU affinity mask >>
      ...
      struct list_head tasks;
```

```
<< (The implementation of the doubly-linked circular task list)
From <linux/types.h> :
...
struct list_head {
      struct list_head *next, *prev;
};
...
  tasks = {
    next = 0xc03c9d68,
    prev = 0xc3fe80d8
  },
```

<<
Using the powerful 'crash' utility:
Lets use crash to cycle through the task list, printing the PID and name of each task. To do so, we'll need a starting point: lets look up the kernel virtual-address of init's task structure:

```
crash> ps |grep init  << task struct ptr >>
      1      0    0  ffff880232918000  IN   0.0   29536   3636  init
crash>
crash> list task_struct.tasks -s task_struct.pid,comm -h ffff880232918000
ffff880232918000
  pid = 1
  comm = "init\000nit\000\000\000\000\000\000\000"
ffff880232918a30
  pid = 2
  comm = "kthreadd\000\000\000\000\000\000\000"
ffff880232919460
  pid = 3
  comm = "ksoftirqd/0\000\000\000\000"
ffff88023291a8c0
  pid = 5
  comm = "kworker/0:0H\000\000\000"

--snip--

crash>
```

*Crash Tip: within crash, use the **help <command>** to get detailed and useful help, often with excellent examples!*

>>

Programatically, can use the macro for_each_process() to iterate through the processes (_not_ threads) on the task list:

```
#include <linux/sched/signal.h>                << recent kernel's >>
...
#define for_each_process(p) \
        for (p = &init_task ; (p = next_task(p)) != &init_task ; )
...
```

Well then, what about iterating through threads?
Use the macros *do_each_thread()* and *while_each_thread()* in pairs on a single loop, as in:

```
        struct task_struct *g, *t; // 'g' : process ptr; 't': thread ptr !
        do_each_thread(g, t) {
                printk(KERN_DEBUG "%d %d %s\n", g->tgid, t->pid, g->comm);
        } while_each_thread(g, t);
---
>>
...
        struct mm_struct *mm;
        struct mm_struct *active_mm;      << VM:
```

mm : user address-space mapping

active_mm : mapping for "anonymous" address space– kernel threads have it as NULL as they have no external mapping.

Details: https://github.com/torvalds/linux/blob/master/Documentation/vm/active_mm.txt

```
>>
<<
  mm = 0xcf5a5300,
  active_mm = 0xcf5a5300,
>>
...
        int                             exit_state;
        int                             exit_code;
        int                             exit_signal;
        /* The signal sent when the parent dies: */
        int                             pdeath_signal;
        /* JOBCTL_*, siglock protected: */
        unsigned long                   jobctl;

        /* Used for emulating ABI behavior of previous Linux versions: */
        unsigned int                    personality;
<<
  exit_state = 0x0,
  exit_code = 0x0,
  exit_signal = 0x11,      << 0x11=17=SIGCHLD >>
  pdeath_signal = 0x0,
>>
        /* Bit to tell LSMs we're in execve(): */
        unsigned                        in_execve:1;
        unsigned                        in_iowait:1;
...
```

```
        pid_t pid;
        pid_t tgid;
```

<<

(The following becomes clearer once we cover in-depth Linux's clone() and how it's used internally).

On Linux, there is no difference between a "thread" and "process" except for "share-ability" of resources. Every thread/process (to avoid confusion lets just call it a "task" :-) has a unique PID assigned to it – this is not the PID in the POSIX sense of the term (meaning, *every* thread has a unique PID which of course violates the POSIX notion that all threads of a process share the same PID).

In order to remain POSIX-compliant, a new member called TGID was introduced into the task_struct. The TGID (of every thread) is set to the POSIX PID of the creator (master) process; in addition the getpid() system call has been modified to return the TGID and not the PID.

>>

```
<<
  in_execve = 0,
  in_iowait = 0,
  sched_reset_on_fork = 0,
  sched_contributes_to_load = 0,
  pid = 0xe17,
  tgid = 0xe17,
...
(gdb) p /d p.pid
$24 = 3607
(gdb) p /d p.tgid
$25 = 3607
>>
```

<<

FAQ: Within the kernel, given a task's PID, how can we locate the corresponding task structure?

From here:

If you want to find the `task_struct` from a module, `find_task_by_vpid(pid_t nr)` etc. are not going to work since these functions are not exported.

In a module, you can use the following (exported) function instead:

```
pid_task(find_vpid(pid), PIDTYPE_PID);
```

>>

<-- new feature of gcc 4.2 or above >

```
#ifdef CONFIG_CC_STACKPROTECTOR                 << 2.6.23 onward >>
        /* Canary value for the -fstack-protector gcc feature */
        unsigned long stack_canary;
#endif
```

<--

*Relevant snippet from the **arch/Kconfig** help section:*

```
...
config HAVE_CC_STACKPROTECTOR
```

```
        bool
        help
          An arch should select this symbol if:
          - its compiler supports the -fstack-protector option
          - it has implemented a stack canary (e.g. __stack_chk_guard)

config CC_STACKPROTECTOR
        def_bool n
        help
          Set when a stack-protector mode is enabled, so that the build
          can enable kernel-side support for the GCC feature.

choice
        prompt "Stack Protector buffer overflow detection"
        depends on HAVE_CC_STACKPROTECTOR
        default CC_STACKPROTECTOR_NONE
        help
          This option turns on the "stack-protector" GCC feature. This
          feature puts, at the beginning of functions, a canary value on
          the stack just before the return address, and validates
          the value just before actually returning.  Stack based buffer
          overflows (that need to overwrite this return address) now also
          overwrite the canary, which gets detected and the attack is then
          neutralized via a kernel panic.
...
```

Indeed, in the task creation code, we see:

```
--snip--
 238#ifdef CONFIG_CC_STACKPROTECTOR
 239        tsk->stack_canary = get_random_int();
 240#endif
--snip--
```

-->

```
...
        struct list_head thread_group;
```

<<

Doubly linked list of all threads belonging to the process. Can iterate through the list with:

```
                struct task_struct *t = p;        // p = starting task_ptr
                do {
                        printk(KERN_DEBUG "thrd: %s(%d)\n",
                                t->comm, t->pid);
                        t = next_thread(t);
                } while (t != p);
        ...
```

Additional macros related to process/thread iteration within the task list:

Also note that, from kernel ver 4.11 onward, many of these task-accessor macros and funcitons have been *moved into a new header: <linux/sched/signal.h> ; so take this into account in your code with:*

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(4,11,0)
#include <linux/sched/signal.h>
#endif
```

```
...
#define next_task(p) \
        list_entry_rcu((p)->tasks.next, struct task_struct, tasks)

#define for_each_process(p) \
        for (p = &init_task ; (p = next_task(p)) != &init_task ; )

extern bool current_is_single_threaded(void);

/*
 * Careful: do_each_thread/while_each_thread is a double loop so
 *          'break' will not work as expected - use goto instead.
 */
#define do_each_thread(g, t) \
        for (g = t = &init_task ; (g = t = next_task(g)) != &init_task ; ) do

#define while_each_thread(g, t) \
        while ((t = next_thread(t)) != g)

#define __for_each_thread(signal, t)    \
        list_for_each_entry_rcu(t, &(signal)->thread_head, thread_node)

#define for_each_thread(p, t)           \
        __for_each_thread((p)->signal, t)

/* Careful: this is a double loop, 'break' won't work as expected. */
#define for_each_process_thread(p, t)   \
        for_each_process(p) for_each_thread(p, t)

static inline int get_nr_threads(struct task_struct *tsk)
{
        return tsk->signal->nr_threads;
}
...
```

```
>>
      struct list_head  thread_node;
      struct completion *vfork_done;   /* for vfork() */
...
        u64                             utime;
        u64                             stime;
#ifdef CONFIG_ARCH_HAS_SCALED_CPUTIME
        u64                             utimescaled;
        u64                             stimescaled;
#endif
        u64                             gtime;
        struct prev_cputime             prev_cputime;
#ifdef CONFIG_VIRT_CPU_ACCOUNTING_GEN
        struct vtime                    vtime;
#endif

...
/* Context switch counts: */
        unsigned long                   nvcsw;
```

```
	unsigned long			nivcsw;

	/* Monotonic time in nsecs: */
	u64				start_time;

	/* Boot based time in nsecs: */
	u64				real_start_time;

	/* MM fault and swap info: this can arguably be seen as either mm-specific or thread-specific: */
	unsigned long			min_flt;
	unsigned long			maj_flt;

#ifdef CONFIG_POSIX_TIMERS
	struct task_cputime		cputime_expires;
	struct list_head		cpu_timers[3];
#endif

	/* Process credentials: */
```

<< >= 2.6.24 (?)*:*

A broader notion of the security context of the task. Consists of a set of actionable objects, objective and subjective contexts. See *Documentation/credentials.txt* and *include/linux/cred.h* for details.

real_cred : objective part of this context is used whenever that task is acted upon.
task->cred : subjective context that defines the details of how that task is going to act upon another object. This may be overridden temporarily to point to another security context, but normally points to the same context as task->real_cred.

>>

```
	/* Objective and real subjective task credentials (COW): */
	const struct cred __rcu		*real_cred;

	/* Effective (overridable) subjective task credentials (COW): */
	const struct cred __rcu		*cred;
```

<<

(One) Implication: can't access (for example, the RUID, EUID) as p->euid ;

now must use

```
  __kuid_val(p->cred->uid)
  __kuid_val(p→cred→euid)
```

or, task_uid(p), task_euid(p);

Also, see [this StackOverflow answer](#) for more ...

Lots of convenient macros (like current_uid(), current_euid(), etc) here: *include/linux/cred.h*

>>

<< *With crash:*

```
crash> task -R cred
PID: 5374   TASK: ffff9d133d964380  CPU: 0   COMMAND: "bash"
  cred = 0xffff9d1334e58d80,

crash> cred 0xffff9d1334e58d80
struct cred {
  usage = {
    counter = 11
  },
  uid = {
    val = 1000
  },
  gid = {
    val = 1000
  },
  suid = {
    val = 1000
  },
  sgid = {
    val = 1000
  },
  euid = {
    val = 1000
  },
  egid = {
    val = 1000
  },
...
```

>>

<<

Security / Hack

Check out this artcile - “This is what a root debug backdoor in a Linux kernel looks like”, 09 May 2016.

Excerpts-

“A root backdoor for debugging ARM-powered Android gadgets managed to end up in shipped firmware – and we're surprised this sort of colossal blunder doesn't happen more often.

The howler is the work of Chinese ARM SoC-maker Allwinner, which wrote its own kernel code underneath a custom Android build for its devices.

Its Linux 3.4-based kernel code, on *Github here*, contains what looks to *The Register* like a debug mode the authors forgot to kill. Although it doesn't appear to have made it into the mainstream kernel source, it was picked up by firmware builders for various gadgets using Allwinner's chips.

It's triggered by writing <<the string>> “rootmydevice” to the special file */proc/sunxi_debug/sunxi_debug*. That gives the current running process root privileges. If that file is present on your device or single-board computer, then you need to get rid of it. This is the code that checks for the magic write:

```
        if(!strncmp("rootmydevice",(char*)buf,12)){
                cred = (struct cred *)__task_cred(current);
                cred->uid = 0;
                cred->gid = 0;
                cred->suid = 0;
                cred->euid = 0;
                cred->euid = 0;
                cred->egid = 0;
                cred->fsuid = 0;
                cred->fsgid = 0;
                printk("now you are root\n");
        }
```

Tkaiser, a moderator over at the forums of the Armbian operating system (a Linux distro for ARM-based development boards) notes there's a number of vulnerable systems in the field.

--snip--

There are probably other products out there using the Allwinner SoC and the dodgy code. Tkaiser pointed out that FriendlyARM was also quick to issue a patch.”

>>

<<

More on security / hacking [Optional]

- The Linux kernel code can always access userspace code/data regions
  - BUT as noted here: https://www.kernel.org/doc/html/latest/security/self-protection.html :
    “The kernel must never execute userspace memory. The kernel must also never access userspace memory without explicit expectation to do so. These rules can be enforced either by support of hardware-based restrictions (x86's SMEP/SMAP, ARM's PXN/PAN) or via emulation (ARM's Memory Domains). By blocking userspace memory in this way, execution and data parsing cannot be passed to trivially-controlled userspace memory, forcing attacks to operate entirely in kernel memory.”
- An attacker can carefully setup user memory with an attack payload, then
- (Re)Search the kernel for an exploitable bug(s)
  - Have the kernel run some buggy code*, that in turn, causes it to incorrectly access user regions
  - Remap the pointer(s) to point to the (userspace) attack code (the ‘shellcode’), which will run in kernel mode
- Voila! Privilege escalation (privesc) becomes easy

* Exploit buggy kernel code?
Yes, find a kernel bug: often useful, a null pointer dereference (or stack overflow). So, we craft

stuff: mmap() the null address in our usermode process space, memcpy() our attack code (like, `commit_creds(prepare_kernel_creds(0));` into that memory region. Trigger the kernel bug; when the kernel dereferences the null pointer, it leads to (the IP/PC is set to) our exploit shellcode, which then runs, possibly giving us a root shell!
[Note- Hardening countermeasure: modern kernels run with *mmap_min_addr* set, so you can't typically mmap the null page].
[See this PDF 'Writing Kernel Exploits', McAllister, Sept 2012]

```
>>

/*
         * executable name, excluding path.
         *
         * - normally initialized setup_new_exec()
         * - access it with [gs]et_task_comm()
         * - lock it with task_lock()
         */
        char                            comm[TASK_COMM_LEN];
<<
 comm = "bash\000-terminal\000",
…
(gdb) p p.comm
$11 = "bash\000-terminal\000"
>>

        struct nameidata                *nameidata;

#ifdef CONFIG_SYSVIPC
        struct sysv_sem                 sysvsem;
        struct sysv_shm                 sysvshm;
#endif
#ifdef CONFIG_DETECT_HUNG_TASK
        unsigned long                   last_switch_count;
        unsigned long                   last_switch_time;
#endif
        /* Filesystem information: */
        struct fs_struct                *fs;

        /* Open file information: */
        struct files_struct             *files;    << the process OFDT - Open File Desciptor
                                                      Table >>
<<
  fs = 0xc125a3c0,
  files = 0xc12738c0,
>>

        /* Namespaces: */
        struct nsproxy                  *nsproxy;

        /* Signal handlers: */
        struct signal_struct            *signal;
```

<< this has the nr_threads count, timers, some accounting stats, the array of struct rlimit[]'s, etc >>

<<

*Resource limit info is per-process based and is in *signal . Also, all threads of a process share the resource limits. Use '$ ulimit -a' to see the resource limits (for calling process)..*

```
crash> whatis task_struct |grep signal
    int exit_signal;
    int pdeath_signal;
    struct signal_struct *signal;
crash> task -R signal
PID: 5374   TASK: ffff9d133d964380  CPU: 0   COMMAND: "bash"
  signal = 0xffff9d13397f9c00,

crash> signal_struct -x 0xffff9d13397f9c00
...
  rlim = {{                                     << RLIMIT_CPU ; soft and hard limits >>
      rlim_cur = 0xffffffffffffffff,               <<  0xffffffffffffffff =>
                              'infinite', i.e., not artificially limited by the kernel >>
      rlim_max = 0xffffffffffffffff
    }, {                                        << RLIMIT_FSIZE >>
      rlim_cur = 0xffffffffffffffff,
      rlim_max = 0xffffffffffffffff
    }, {                                        << RLIMIT_DATA >>
      rlim_cur = 0xffffffffffffffff,
      rlim_max = 0xffffffffffffffff
    }, {                                        << RLIMIT_STACK >>
      rlim_cur = 0x800000,
      rlim_max = 0xffffffffffffffff
    }, {                                        << RLIMIT_CORE >>
      rlim_cur = 0x0,
      rlim_max = 0xffffffffffffffff
    }, {                                        << RLIMIT_RSS >>
      rlim_cur = 0xffffffffffffffff,
      rlim_max = 0xffffffffffffffff
    }, {                                        << RLIMIT_NPROC >>
      rlim_cur = 0xe99,
      rlim_max = 0xe99
    }, {                                        << RLIMIT_NOFILE >>
      rlim_cur = 0x400,
      rlim_max = 0x1000
    }, {                                        << RLIMIT_MEMLOCK >>
      rlim_cur = 0x1000000,
      rlim_max = 0x1000000
    }, {                                        << RLIMIT_LOCKS >>
      rlim_cur = 0xffffffffffffffff,
      rlim_max = 0xffffffffffffffff
    }, {                                        << RLIMIT_AS >>
      rlim_cur = 0xffffffffffffffff,
      rlim_max = 0xffffffffffffffff
    }, {                                        << RLIMIT_SIGPENDING >>
      rlim_cur = 0xe99,
      rlim_max = 0xe99
    }, {                                        << RLIMIT_MSGQUEUE >>
      rlim_cur = 0xc8000,
      rlim_max = 0xc8000
    }, {                                        << RLIMIT_NICE >>
      rlim_cur = 0x0,
      rlim_max = 0x0
    }, {                                        << RLIMIT_RTPRIO >>
```

```
      rlim_cur = 0x0,
      rlim_max = 0x0
    }, {                                    << RLIMIT_RTTIME >>
      rlim_cur = 0xffffffffffffffff,
      rlim_max = 0xffffffffffffffff
    }},
...
crash>
>>


        struct sighand_struct           *sighand;
<<
Signal Handling:

struct sighand_struct {
        atomic_t                count;
        struct k_sigaction      action[_NSIG];
        spinlock_t              siglock;
        wait_queue_head_t       signalfd_wqh;
};
>>

        sigset_t                        blocked;    << blocked sigmask, 'regular' and RT
                                                       signals >>
        sigset_t                        real_blocked;
        /* Restored if set_restore_sigmask() was used: */
        sigset_t                        saved_sigmask;
        struct sigpending               pending;    << signals pending delivery >>
        unsigned long                   sas_ss_sp;
        size_t                          sas_ss_size;
        unsigned int                    sas_ss_flags;


<<
A lot of members that follow, are compile-time turned ON if the corresponding
CONFIG_XXX directive is selected (at kernel configuration time).
>>
#ifdef CONFIG_RT_MUTEXES
        /* PI waiters blocked on a rt_mutex held by this task: */
        struct rb_root_cached           pi_waiters;
        /* Updated under owner's pi_lock and rq lock */
        struct task_struct              *pi_top_task;
        /* Deadlock detection and priority inheritance handling: */
        struct rt_mutex_waiter          *pi_blocked_on;
#endif

#ifdef CONFIG_DEBUG_MUTEXES
        /* Mutex deadlock detection: */
        struct mutex_waiter             *blocked_on;
#endif

<< ... a whole bunch of stuff here ... >>
```

```
#ifdef CONFIG_GCC_PLUGIN_STACKLEAK
        unsigned long                   lowest_stack;
        unsigned long                   prev_lowest_stack;
#endif

        /*
         * New fields for task_struct should be added above here, so that
         * they are included in the randomized portion of task_struct.
         */
        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 task_struct ends; finally! >>
```

<<

__thread_struct__:
Holds __hardware context__ of the task; is obviously arch-dependant. Used for context-switch, fault handling, etc.

```
crash> task -R thread -x          << the hardware context is in struct thread_struct thread >>
PID: 5374   TASK: ffff9d133d964380  CPU: 0   COMMAND: "bash"
  thread = {
    tls_array = {{
        limit0 = 0x0,
        base0 = 0x0,
        base1 = 0x0,
        type = 0x0,
        s = 0x0,
        dpl = 0x0,
        p = 0x0,
        limit1 = 0x0,
        avl = 0x0,
        l = 0x0,
        d = 0x0,
        g = 0x0,
        base2 = 0x0
      }, {
        limit0 = 0x0,
...
   }},
    sp = 0xffffb31643ec7cd0,
    es = 0x0,
    ds = 0x0,
    fsindex = 0x0,
    gsindex = 0x0,
```

```
    fsbase = 0x7fe027164740,
    gsbase = 0x0,
    ptrace_bps = {0x0, 0x0, 0x0, 0x0},
    debugreg6 = 0x0,
    ptrace_dr7 = 0x0,
    cr2 = 0x0,
    trap_nr = 0x0,
    error_code = 0x0,
    io_bitmap_ptr = 0x0,
    iopl = 0x0,
    io_bitmap_max = 0x0,
    addr_limit = {
      seg = 0x7ffffffff000
    },
...
 xmm_space = {0x0, 0xffff00, 0xffff0000, 0xffffffff, 0x6e617769, 0x48434554, 0x6172632f, 0x685f6873,
0x65706c65, 0x4f000072, 0x511, 0x0, 0x0, 0x0, 0x0, 0x0, 0xa0a0a0a, 0xa0a0a0a, 0xa0a0a0a, 0xa0a0a0a,
0x4e18fdc0, 0x5579, 0x4e1961b0, 0x5579, 0x4e175270, 0x5579, 0x4e1752d0, 0x5579, 0x0, 0x0, 0x0, 0x0,
0x75722f2e, 0x72635f6e, 0x687361, 0x746c, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
            padding = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
...
crash>
```

Suggested Assignments

1. *show_monolithic*: enhance the earlier "Hello, world" kernel module to print the process context (just show the process name and PID for now) that the init and cleanup code runs in.

2. Enhance the above kernel module to print out some process-context information; for example, print out the process name, PID (actually TGID), VM information (look up some members of the mm_struct, like start_data, end_data, etc etc).

   Also: print the kernel virtual addresses of some variables in the module.
   Print the current value of *jiffies* as well.

3. *show_threads:* Write a kernel module that iterates over all *threads* alive on the system printing out relevant details (as above).

Example kernel module – taskdtl – that, given a PID as parameter, dumps several task structure details

1. On an x86_64 (Ubuntu 18.04.2 LTS) VM

```
pid=1, tp = 0xffff97dbfe30db00
Task struct @ 0xffff97dbfe30db00 ::
              Process/Thread:          systemd, TGID      1, PID      1
                                        RealUID     :      0, EffUID :      0
                                        login UID   :     -1
Task state (1) :
 S: interruptible sleep
thread_info (0xffff97dbfe30db00) is within the task struct itself
stack      : 0xffffb5cc80194000 ; vmapped? yes
              flags : 0x400100
              sched ::
                curr CPU    :   0
                on RQ?      : no
                prio        : 120
                static prio : 120
                normal prio : 120
                RT priority :   0
                vruntime    : 416420432
  policy      : Normal/Other
  cpus allowed:   2
                # times run on cpu: 5415
                time waiting on RQ: 638590770
mm info ::
               not a kernel thread; mm_struct   : 0xffff97dbf5577380
 PGD base addr : 0xffff97dbfd084000
               mm_users = 1, mm_count = 1
               PTE page table pages = 212992 bytes
               # of VMAs                                 =        151
               Highest VMA end address                   = 0x7ffd40783000
               High-watermark of RSS usage               =      2211 pages
               High-water virtual memory usage           =     56313 pages
               Total pages mapped                        =     39929 pages
               Pages that have PG_mlocked set            =         0 pages
               Refcount permanently increased            =         0 pages
               data_vm: VM_WRITE & ~VM_SHARED & ~VM_STACK =      4597 pages
               exec_vm:   VM_EXEC & ~VM_WRITE & ~VM_STACK =      2838 pages
               stack_vm:                         VM_STACK =        33 pages
               def_flags                                 = 0x0
mm userspace mapings (high to low) ::
               env        : 0x7ffd406e7f3d - 0x7ffd406e7fed  [   176 bytes]
               args       : 0x7ffd406e7f2b - 0x7ffd406e7f3d  [    18 bytes]
               start stack: 0x7ffd406e5e70
               heap       : 0x5569ca69b000 - 0x5569ca81a000  [  1532 KB,      1 MB]
               data       : 0x5569c94fad10 - 0x5569c9532158  [   221 KB,      0 MB]
               code       : 0x5569c91ad000 - 0x5569c9532158  [  3604 KB,      3 MB]
in execve()? no
in iowait  ? no
              stack canary  : 0x1da41ddab7911700
              utime, stime  : 112000000, 872000000
              # vol c/s, # invol c/s :   4574,    841
              # minor, major faults  :  10571,    108
              task I/O accounting ::
               read bytes               : 19726336
               written (or will) bytes  :       0
```

```
                cancelled write bytes    :       0
                # read syscalls          :    3429
                # write syscalls         :    5025
                accumulated RSS usage    : 752429652 (734794 KB)
                accumulated  VM usage    : 10392980423 (10149394 KB)
 Hardware ctx info location is thread struct: 0xffff97dbfe30ed80
               X86_64 ::
                thrd info: 0xffff97dbfe30db00
                 sp  : 0xffffb5cc80197d08
                 es  : 0x0, ds  : 0x0
                 cr2 : 0x0, trap #  : 0x0, error code  :  0x0
                 mm: addr limit (user boundary) : 0x7ffffffff000 (134217727 GB, 131071
TB)
```

2. On an Raspberry Pi 3 Model B+ Aarch64 (Ubuntu 18.04.2 LTS) system

```
rpi64 taskdtl $ lscpu
Architecture:        aarch64
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  4
Socket(s):           1
Vendor ID:           ARM
Model:               4
Model name:          Cortex-A53
Stepping:            r0p4
CPU max MHz:         1400.0000
CPU min MHz:         600.0000
BogoMIPS:            38.40
Flags:               fp asimd evtstrm crc32 cpuid
rpi64 taskdtl $ ./run_taskdtl 1246
[   69.114923] taskdtl: loading out-of-tree module taints kernel.
[   69.115099] taskdtl: module verification failed: signature and/or required key
missing - tainting kernel
[   69.115431] pid=1246, tp = 0xffffcf9632ed0000
[   69.115438] Task struct @ 0xffffcf9632ed0000 ::
               Process/Thread:            sshd, TGID   1246, PID   1246
                                        RealUID    :      0, EffUID :      0
                                        login UID  :     -1
[   69.115441] Task state (1) :
[   69.115443]  S: interruptible sleep
[   69.115446] thread_info (0xffffcf9632ed0000) is within the task struct itself
[   69.115452] stack      : 0xffff00000b060000 ; vmapped? yes
               flags : 0x400100
               sched ::
                 curr CPU    :   3
                 on RQ?      : no
                 prio        : 120
                 static prio : 120
                 normal prio : 120
                 RT priority :   0
                 vruntime    : 28149081
[   69.115454]   policy      : Normal/Other
[   69.115457]   cpus allowed:   4
                 # times run on cpu:     5
                 time waiting on RQ:     97032
```

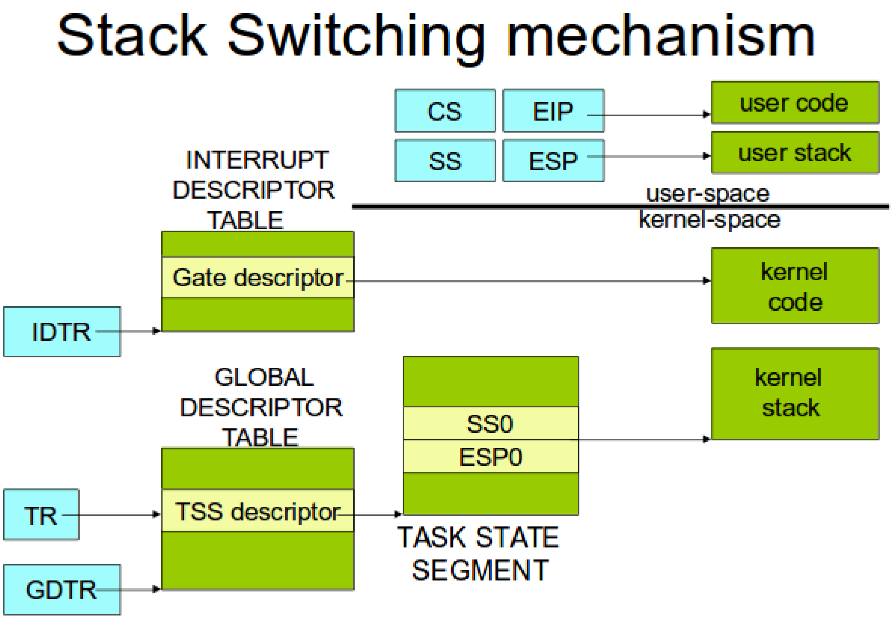
```
[   69.115460] mm info ::
                not a kernel thread; mm_struct   : 0xffffcf9638e4c380
[   69.115470]  PGD base addr : 0xffffcf96361fa000
                mm_users = 1, mm_count = 1
                PTE page table pages = 57344 bytes
                # of VMAs                                 =        146
                Highest VMA end address                   = 0xfffffa0f9000
                High-watermark of RSS usage               =      1266 pages
                High-water virtual memory usage           =      2612 pages
                Total pages mapped                        =      2603 pages
                Pages that have PG_mlocked set            =         0 pages
                Refcount permanently increased            =         0 pages
                data_vm: VM_WRITE & ~VM_SHARED & ~VM_STACK =       188 pages
                exec_vm:   VM_EXEC & ~VM_WRITE & ~VM_STACK =      1851 pages
                stack_vm:                         VM_STACK =        33 pages
                def_flags                                 = 0x0
[   69.115480] mm userspace mapings (high to low) ::
                env       : 0xfffffa0f8f27 - 0xfffffa0f8fe9  [   194 bytes]
                args      : 0xfffffa0f8f15 - 0xfffffa0f8f27  [    18 bytes]
                start stack: 0xfffffa0f8b50
                heap      : 0xaaab24677000 - 0xaaab24698000  [   132 KB,      0 MB]
                data      : 0xaaaae92f1488 - 0xaaaae92f49a8  [    13 KB,      0 MB]
                code      : 0xaaaae9238000 - 0xaaaae92f49a8  [   754 KB,      0 MB]
[   69.115491] in execve()? no
               in iowait  ? no
               stack canary  : 0xa29dd6a5df417000
               utime, stime  : 24000000, 8000000
               # vol c/s, # invol c/s :      4,      1
               # minor, major faults  :    443,      0
               task I/O accounting ::
                read bytes               :       0
                written (or will) bytes  :       0
                cancelled write bytes    :       0
                # read syscalls          :      67
                # write syscalls         :      12
                accumulated RSS usage    :  28734372 ( 28060 KB)
                accumulated  VM usage    :  78730463 ( 76885 KB)
[   69.115502] Hardware ctx info location is thread struct: 0xffffcf9632ed1980
               ARM64 ::
                thrd info: 0xffffcf9632ed0000
                 addr limit : 0xffffffffffff (268435455 MB, 262143 GB)
                Saved registers ::
   X19 = 0xffffcf96395d1d80  X20 = 0xffffcf9632ed0000   X21 = 0x0
   X22 = 0xffff1591b6cdd000  X23 = 0xffff1591b7209000   X24 = 0xffffcf9638e4c380
   X25 = 0xffffcf9632ed06a0  X26 = 0xffffcf963abc8b80   X27 = 0xffff1591b66066d8
   X28 = 0xffffcf9632ed0000
    FP = 0xffff00000b063700   SP = 0xffff00000b063700    PC = 0xffff1591b5c87fc4
                 fault_address  : 0x0, fault code (ESR_EL1): 0x0
                arm64 pointer authentication absent.
rpi64 taskdtl $
```

[OPTIONAL / FYI]

IA-32 : How does the CPU switch the 'sp' register to kernel mode stack when entering kernel?

Done within the processor microcode.

Source
Special CPU segment-register: TR
TR is the 'Task Register'
TR holds 'selector' for a GDT descriptor
Descriptor is for a 'Task State Segment'
So TR points indirectly to current TSS
TSS stores address of kernel-mode stack



http://kaiwantech.in

kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here: http://bit.ly/ktcorp