Linux Task Structure:

Contents

---

1. task_struct:

1.1. About task_struct:

In Linux kernel, the `task_struct` is the **central data structure** that represents a **process or thread**. Every running process or thread in the system has a `task_struct` associated with it.

1.1.1. 🔧 What is `task_struct`?

It is a C structure defined in the Linux kernel source code (in `include/linux/sched.h`). It stores **all the information** about a process (or thread)—everything the kernel needs to manage it.

1.1.2. 🧠 Why is it important?

The `task_struct` is how the kernel:

- Knows what a process is doing
- Manages scheduling
- Handles signals, memory, file descriptors, and more

Each process/thread in the system is represented by one `task_struct`, and the kernel operates on these.

1.1.3. 🧩 Key Fields in *task_struct*

Here are some major fields inside ***task_struct*** (simplified):

Field	**Purpose**
pid	Process ID
tgid	Thread Group ID (equal to `pid` for the main thread)
comm	Name of the process (16-char string)
state	Current state (e.g. running, sleeping, zombie)
mm	Pointer to memory descriptor (virtual memory)
files	Pointer to file descriptor table
signal	Signal-related info
sched_class	Scheduling policy and behavior
se	Scheduler entity, used by the CFS scheduler
parent, real_parent	Pointers to parent process
children, sibling	Linked list pointers to track child/sibling tasks
cpu	Last executed CPU
thread	Architecture-specific info (registers, etc.)

1.1.4. 🔄 How It's Used

1. The kernel uses a **linked list/tree** of all `task_structs` to manage and traverse processes.
2. When a process is created `fork()/clone()`..., the kernel allocates and init's a new `task_struct`.
3. The scheduler switches between processes by switching between `task_structs`.
4. The current process's `task_struct` can be accessed with the `current` macro.

1.1.5. 🧵 Processes vs Threads

In Linux, **threads are just processes** that share certain resources (like memory). They still have a `task_struct`.

The difference lies in:

- What they share (e.g., address space `mm`, file descriptors `files`, etc.)
- Controlled by flags to `clone()`

1.1.6. 🧪 Example

To see `task_struct` in action:

```c
#include <linux/sched.h>

struct task_struct *task;

for_each_process(task) {
    printk(KERN_INFO "Process: %s [PID: %d]\n", task->comm, task->pid);
}
```

This snippet iterates over all processes and prints their names and PIDs.

1.1.7. 📘 Source Location

- Header: `include/linux/sched.h`
- Implementation details: spread across `kernel/sched/`, `kernel/fork.c`, etc.

1.1.8. 🧠 Summary

`task_struct` is the core structure **for process/thread management.** It holds everything the kernel needs to manage and schedule a process. **It's created during `fork()`, updated during execution, and deleted during exit.** Understanding it is key to learning Linux internals, kernel programming, and system-level debugging.

---

2. task_struct Lifecycle from tracing view:

To explore `task_struct` from a **tracing perspective**, we need to see how it's **created**, **used**, and **destroyed** during a process's lifecycle, and how **you can trace it** through kernel mechanisms like tracepoints, ftrace, BPF or printk debugging.

2.1. 🔍 `task_struct` Lifecycle from a Tracing View

2.1.1. Process Creation (`fork`, `clone`)

When a process is created, the kernel:

- Allocates a new `task_struct`
- Initializes it based on the parent
- Sets up scheduling, memory, file descriptors, etc.

2.1.1.1. 🔧 Code Path:

```
syscall (e.g. clone/sys_clone)
 → kernel_clone()
   → copy_process()
     → alloc_task_struct()
     → dup_task_struct()
     → copy_*() helpers (copy_mm, copy_files, copy_fs, ...)
```

2.1.1.2. 🔎 Tracepoints and Probes:

- `trace_sched_process_fork` — triggered when a new process is created.
- You can trace it with:

```
sudo perf trace -e sched:sched_process_fork
```

2.1.1.3. 🐝 eBPF (BPFtrace):

```
sudo bpftrace -e 'tracepoint:sched:sched_process_fork { printf("New task: %s (%d)\n",
args->child_comm, args->child_pid); }'
```

—

2.1.2. Process Running / Scheduling

The scheduler switches between `task_structs`. The kernel uses fields in `task_struct` like `se` (scheduler entity), `state`, and CPU affinity to make decisions.

2.1.2.1. 🔧 Code Path:

```
schedule()
 → context_switch()
   → switch_to()
```

2.1.2.2. 🔎 Tracepoints:

- `sched_switch` — very useful for viewing process context switches.

```
sudo perf trace -e sched:sched_switch
```

2.1.2.3. 🐝 BPFtrace Example:

```
sudo bpftrace -e 'tracepoint:sched:sched_switch { printf("%s (%d) → %s (%d)\n", args->prev_comm, args->prev_pid, args->next_comm, args->next_pid); }'
```

—

2.1.3. 3. Process Exit / `do_exit()`

When a process exits:

- `do_exit()` is called
- `exit_mm()`, `exit_files()`, etc., clean up
- The `task_struct` is marked as `EXIT_ZOMBIE`, then freed

2.1.3.1. 🔧 Code Path:

```
do_exit()
 → exit_notify()
 → schedule()
 → release_task()
 → free_task()
```

2.1.3.2. 🔎 Tracepoint:

- `sched_process_exit` — logs when a process exits.

```
sudo perf trace -e sched:sched_process_exit
```

2.2. 🔬 How to Access `task_struct` While Tracing

When writing custom kernel modules or using kprobes:

- You can use the `current` macro to access the `task_struct` of the running process.

```
struct task_struct *t = current;
printk("Running process: %s [%d]\n", t->comm, t->pid);
```

With BPF (via BCC or BPFtrace), you can use helpers like `bpf_get_current_task()` to access the pointer to `task_struct`.

2.3. 🔄 Common Fields to Trace in `task_struct`

If you're following a `task_struct` in traces, you might want to access:

Field	Purpose
`pid` / `tgid`	Process/thread ID
`comm`	Command name

Field	Purpose
`state`	Task state (running, sleeping, etc.)
`prio` / `static_prio`	Scheduling priority
`mm`	Memory management struct
`real_parent`	Who forked the task
`start_time`	When the process started

2.4. 🧭 Debug Tools for Tracing `task_struct`

2.4.1. 🔧 Ftrace

Enable and trace with:

```
cd /sys/kernel/debug/tracing
echo sched_switch > current_tracer
cat trace
```

2.4.2. 🔧 `perf`

Use `perf` to trace process creation/destruction:

```
sudo perf record -e sched:sched_process_* -a
sudo perf script
```

2.4.3. 🐝 `BPFtrace`/BCC

Write scripts to attach to tracepoints, kprobes, or uprobes and introspect the `task_struct`.

—

2.5. ✅ Summary: Tracing `task_struct`

Lifecycle Stage	Main Function	Key Tracepoint
Creation	`copy_process()`	`sched_process_fork`
Scheduling	`schedule()` / `context_switch()`	`sched_switch`
Termination	`do_exit()` / `release_task()`	`sched_process_exit`

With tracing tools like **perf**, **ftrace**, and **BPF**, you can monitor when a `task_struct` is created, scheduled, or destroyed — giving you deep visibility into process behavior in real time.

3. Ex eBPF script to traces `task_struct` fields across fork/switch/exit:

A complete **BPFtrace script** that traces process lifecycle events (`fork`, `context switch`, and `exit`) and logs some fields from the associated `task_struct`, such as:

- Process name (`comm`)
- PID
- Parent PID
- State (for exit)
- CPU (for switch)

—

3.1. 📜 Full BPFtrace Script: Trace `task_struct` Lifecycle

```
#!/usr/bin/env bpftrace

/**
 * Trace task_struct lifecycle:
 * - On fork: show new PID and parent
 * - On sched_switch: show context switch between processes
 * - On exit: show PID and exit state
 */

// When a new task is forked
tracepoint:sched:sched_process_fork
{
    printf("FORK: parent=%s (pid=%d) → child=%s (pid=%d)\n",
        str(args->parent_comm), args->parent_pid,
        str(args->child_comm), args->child_pid);
}

// On context switch (scheduler switches between tasks)
tracepoint:sched:sched_switch
{
    printf("SWITCH: %s (pid=%d, cpu=%d) → %s (pid=%d, cpu=%d)\n",
        str(args->prev_comm), args->prev_pid, args->prev_cpu,
        str(args->next_comm), args->next_pid, args->next_cpu);
}

// When a process exits
tracepoint:sched:sched_process_exit
{
    printf("EXIT: %s (pid=%d)\n", str(args->comm), args->pid);
}
```

3.2. 🛠️ How to Run This

1. Save the script as `trace_tasks.bt`
2. Run it with sudo:

```
sudo bpftrace trace_tasks.bt
```

You'll see real-time output like:

```
FORK: parent=bash (pid=1234) → child=ls (pid=5678)
SWITCH: bash (pid=1234, cpu=0) → ls (pid=5678, cpu=0)
EXIT: ls (pid=5678)
```

3.3. 🧠 Optional Enhancements

You can extend this script to:

- Log `prio` and `state` via a kprobe on `do_exit` or `schedule`
- Monitor specific PIDs or command names
- Output timestamps with `@time = nsecs;`

3.4. BCC (Python) version for introspection of `task_struct` fields (e.g., state codes, memory usage, etc.)?

BCC (BPF Compiler Collection) with Python gives you **deeper access** into `task_struct`, allowing richer introspection (e.g., task state, memory, priority) than basic tracepoints alone.

Here's a **Python BCC script** that traces:

- `sched_process_fork`
- `sched_switch`
- `sched_process_exit`

And uses **direct access to `task_struct`** via BPF helpers for:

- Process name
- PID / PPID
- State
- Virtual memory size
- Priority

—

3.5. 📜 Full BCC (Python) Script — `trace_task_struct.py`

```
#!/usr/bin/env python3
from bcc import BPF
from time import sleep, strftime
from bcc.utils import printb

# BPF program
bpf_program = """
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>

struct data_t {
    u32 pid;
    u32 ppid;
    char comm[TASK_COMM_LEN];
    long state;
    u64 vmsize;
    int prio;
};

BPF_PERF_OUTPUT(events);

// Called on process exit
TRACEPOINT_PROBE(sched, sched_process_exit) {
    struct task_struct *task = (struct task_struct *)bpf_get_current_task();
    struct data_t data = {};

    data.pid = task->pid;
    data.ppid = task->real_parent->pid;
    data.state = task->state;
    data.prio = task->prio;
    data.vmsize = task->mm ? task->mm->total_vm << PAGE_SHIFT : 0;
    bpf_get_current_comm(&data.comm, sizeof(data.comm));

    events.perf_submit(args, &data, sizeof(data));
```

```
    return 0;
}

// Called on fork
TRACEPOINT_PROBE(sched, sched_process_fork) {
    struct data_t data = {};

    struct task_struct *child = (struct task_struct *)args->child;
    data.pid = child->pid;
    data.ppid = child->real_parent->pid;
    data.state = child->state;
    data.prio = child->prio;
    data.vmsize = child->mm ? child->mm->total_vm << PAGE_SHIFT : 0;
    bpf_probe_read_kernel_str(&data.comm, sizeof(data.comm), child->comm);

    events.perf_submit(args, &data, sizeof(data));
    return 0;
}
"""

# Initialize BPF
b = BPF(text=bpf_program)

# Header
print("%-10s %-6s %-6s %-5s %-12s %-8s" % ("EVENT", "PID", "PPID", "PRIO", "STATE",
"VMEM (KB)"))

# Event handler
def handle_event(cpu, data, size):
    event = b["events"].event(data)
    printb(b"%-10s %-6d %-6d %-5d %-12ld %-8d" % (
        b"TRACE",
        event.pid,
        event.ppid,
        event.prio,
        event.state,
        event.vmsize // 1024  # Convert to KB
    ))

b["events"].open_perf_buffer(handle_event)

# Loop
try:
    while True:
        b.perf_buffer_poll()
except KeyboardInterrupt:
    print("\nExiting...")
```

—

3.6. 🔧 How to Run

3.6.1. 1. Install BCC if not already installed

```
sudo apt install bpfcc-tools python3-bcc
```

3.6.2. 2. Run the script with sudo:

```
sudo python3 trace_task_struct.py
```

—

3.7. 🧪 Example Output

```
EVENT      PID     PPID    PRIO   STATE         VMEM (KB)
TRACE      1234    1       120    0             4512
TRACE      1235    1234    120    0             2048
TRACE      1235    1234    120    0             0
```

—

3.8. 💡 Insights Gained

Field	Source in `task_struct`	Meaning
`pid`	`task->pid`	Process ID
`ppid`	`task->real_parent->pid`	Parent PID
`comm`	`task->comm`	Command name
`state`	`task->state`	Process state (e.g. running)
`prio`	`task->prio`	Static priority
`vmsize`	`task->mm->total_vm`	Virtual memory size (pages)

You can decode `state` using constants like `TASK_RUNNING (0)`, `TASK_INTERRUPTIBLE (1)`, etc.

3.9. Extend to track `sched_switch` and record time spent on CPU per task:

3.9.1. ✅ New Features:

1. **Track `sched_switch` events**
2. **Measure per-task time spent on CPU**
3. **Report:**

- Command name
- PID
- Time on CPU (in ms)
- From/To PID switches (like a live context switch log)

3.9.2. 📜 Extended BCC Python Script: `trace_task_struct_runtime.py`

```python
#!/usr/bin/env python3
from bcc import BPF
from time import sleep
from bcc.utils import printb

# BPF program
bpf_program = """
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>

struct switch_data_t {
    u32 pid;
    char comm[TASK_COMM_LEN];
    u64 delta_ns;
};
```

```
BPF_HASH(start, u32, u64);         // pid -> timestamp
BPF_PERF_OUTPUT(events);

// Track when a task is switched in
TRACEPOINT_PROBE(sched, sched_switch)
{
    u64 ts = bpf_ktime_get_ns();

    // record switch out time for prev task
    u32 prev_pid = args->prev_pid;
    start.update(&prev_pid, &ts);

    // calculate runtime for next task
    u32 next_pid = args->next_pid;
    u64 *tsp = start.lookup(&next_pid);
    if (tsp != 0) {
        u64 delta = ts - *tsp;
        struct switch_data_t data = {};
        data.pid = next_pid;
        data.delta_ns = delta;
        bpf_probe_read_kernel_str(&data.comm, sizeof(data.comm), args->next_comm);
        events.perf_submit(args, &data, sizeof(data));
    }

    return 0;
}
"""

# Load and initialize BPF
b = BPF(text=bpf_program)

print("%-6s %-16s %-12s" % ("PID", "COMM", "CPU Time (ms)"))

# Event callback
def handle_event(cpu, data, size):
    event = b["events"].event(data)
    ms = event.delta_ns / 1_000_000
    printb(b"%-6d %-16s %-12.3f" % (event.pid, event.comm, ms))

# Poll
b["events"].open_perf_buffer(handle_event)
try:
    while True:
        b.perf_buffer_poll()
except KeyboardInterrupt:
    print("\nExiting...")
```

3.9.3. 🧪 Example Output

```
PID    COMM             CPU Time (ms)
1234   bash             1.284
5678   ls               0.544
1234   bash             2.037
```

3.9.4. 🧠 How This Works

- On every `sched_switch`, we:
  - Record the time when a task is switched **out**.
  - Look up the last time the **next** task was scheduled **in**.
  - Subtract to find time delta = CPU time used.
- This gives a rough **active CPU time per task between switches**.

3.9.5. 🧰 How to Run

```
sudo python3 trace_task_struct_runtime.py
```

This will start printing runtime per task as context switches occur.

3.9.6. 💡 next [assignment]

We can extend this further to:

- Aggregate per-PID total runtime
- Count number of switches
- Track per-CPU execution
- Filter to specific processes only

---

4. Using Aya Rust eBPF framework:

Using **Aya** (a Rust eBPF framework) to trace `task_struct` lifecycle events and measure CPU time is a great choice for building production-grade, high-performance tracing tools.

I'll walk you through how to write a **basic Aya-based Rust program** that:

- Hooks into the `sched_switch` tracepoint
- Tracks per-task CPU runtime similarly to the BCC example
- Prints the data to user space

—

4.1. Overview

Aya eBPF programs are split into two parts:

1. **eBPF program** (runs in kernel, written with `aya-bpf`)
2. **Userspace program** (runs in userland, written with `aya`)

4.1.1. Step 1: Setup

Make sure you have Rust + Cargo installed and Aya dependencies.

```
cargo new aya-trace --bin
cd aya-trace
```

Add dependencies in `Cargo.toml`:

```
[dependencies]
aya = { version = "0.20", features = ["full"] }
tokio = { version = "1", features = ["full"] }
anyhow = "1.0"
log = "0.4"
env_logger = "0.10"

[dependencies.aya-bpf]
```

```toml
version = "0.20"
features = ["maps", "tracepoint"]
default-features = false

[build-dependencies]
aya-build = "0.20"
```

4.1.2. Step 2: Write the eBPF program

Create a folder `src/bpf` and add `sched_switch.rs`:

```rust
#![no_std]
#![no_main]

use aya_bpf::{
    macros::tracepoint,
    maps::HashMap,
    programs::TracePointContext,
    helpers::bpf_ktime_get_ns,
};

use core::mem;

#[repr(C)]
pub struct SwitchData {
    pid: u32,
    delta_ns: u64,
    comm: [u8; 16],
}

#[map(name = "START")]
static mut START: HashMap<u32, u64> = HashMap::with_max_entries(10240, 0);

#[map(name = "EVENTS")]
static mut EVENTS: aya_bpf::maps::PerfEventArray<SwitchData> =
PerfEventArray::with_max_entries(1024, 0);

#[tracepoint(name = "sched_switch")]
pub fn sched_switch(ctx: TracePointContext) -> u32 {
    unsafe {
        match try_sched_switch(ctx) {
            Ok(ret) => ret,
            Err(_) => 1,
        }
    }
}

unsafe fn try_sched_switch(ctx: TracePointContext) -> Result<u32, i32> {
    #[repr(C)]
    struct Args {
        prev_comm: [u8; 16],
        prev_pid: u32,
        prev_prio: i32,
        prev_state: i32,
        next_comm: [u8; 16],
        next_pid: u32,
        next_prio: i32,
    }
```

```rust
    let args: Args = ctx.read_at(0)?;

    let ts = bpf_ktime_get_ns();

    // Record switch out time for prev task
    START.insert(&args.prev_pid, &ts, 0)?;

    // Calculate delta for next task
    if let Some(start_ts) = START.get(&args.next_pid) {
        let delta = ts - *start_ts;

        let data = SwitchData {
            pid: args.next_pid,
            delta_ns: delta,
            comm: args.next_comm,
        };

        EVENTS.output(&ctx, &data, mem::size_of::<SwitchData>() as u32)?;
    }

    Ok(0)
}
```

4.1.3. Step 3: Write the userspace program (`src/main.rs`)

```rust
use aya::{
    include_bytes_aligned,
    maps::perf::PerfEventArray,
    programs::TracePoint,
    Bpf,
};
use aya::util::online_cpus;
use std::{convert::TryInto, fs};
use tokio::{signal, task};
use anyhow::{Result, Context};
use log::info;

#[repr(C)]
struct SwitchData {
    pid: u32,
    delta_ns: u64,
    comm: [u8; 16],
}

#[tokio::main]
async fn main() -> Result<()> {
    env_logger::init();

    // Load the compiled BPF object (built separately)
    let mut bpf = Bpf::load_file("target/bpfel-unknown-none/release/sched_switch")?;

    // Attach tracepoint
    let program: &mut TracePoint =
bpf.program_mut("sched_switch").unwrap().try_into()?;
    program.load()?;
    program.attach("sched", "sched_switch")?;
```

```rust
    let mut perf_array = PerfEventArray::try_from(bpf.map_mut("EVENTS")?)?;

    for cpu_id in online_cpus()? {
        let mut buf = perf_array.open(cpu_id, None)?;
        task::spawn(async move {
            let mut buffers = vec![0u8; 1024];
            loop {
                let events = buf.read_events(&mut buffers).await.unwrap();
                for i in 0..events.read {
                    let data_ptr = &buffers[i * std::mem::size_of::<SwitchData>()..];
                    let data = unsafe { ptr_to_struct::<SwitchData>(data_ptr) };
                    let comm =
String::from_utf8_lossy(&data.comm).trim_end_matches('\0').to_string();
                    println!(
                        "PID: {:<6} COMM: {:<16} CPU Time: {:.3} ms",
                        data.pid,
                        comm,
                        data.delta_ns as f64 / 1_000_000.0
                    );
                }
            }
        });
    }

    signal::ctrl_c().await?;
    Ok(())
}

// Helper to convert pointer to struct
unsafe fn ptr_to_struct<T>(ptr: &[u8]) -> T {
    assert!(ptr.len() >= std::mem::size_of::<T>());
    let mut data = core::mem::MaybeUninit::<T>::uninit();
    let data_ptr = data.as_mut_ptr() as *mut u8;
    std::ptr::copy_nonoverlapping(ptr.as_ptr(), data_ptr, std::mem::size_of::<T>());
    data.assume_init()
}
```

—

4.1.4. Step 4: Build the eBPF program

You need to cross-compile the BPF program with `cargo xtask` or by setting the target:

```
cargo build --release --target=bpfel-unknown-none
```

Make sure your environment is set up for building eBPF programs (LLVM 11+, Clang, and Rust targets installed).

—

4.1.5. Step 5: Run

Run your userspace binary as root:

```
sudo ./target/release/aya-trace
```

You'll get output like:

```
PID: 1234   COMM: bash             CPU Time: 2.345 ms
PID: 5678   COMM: ls               CPU Time: 0.543 ms
...
```

4.1.6. Notes:

- Aya requires some setup for building BPF programs; see [Aya's getting started](https://aya-rs.dev/tutorials/getting-started) for details.
- This example only tracks CPU time on `sched_switch`, similar to the BCC example.
- You can extend this with additional tracepoints for fork and exit, reading `task_struct` fields similarly.
- Aya programs use no_std Rust in kernel and async Rust in userland, so be prepared to handle both environments.

4.2. Aya is a great fit for embedded systems

4.2.1. 1. No Python Dependency

- Unlike **BCC**, which requires:
  - Python 3
  - libbcc
  - LLVM
- Aya is **pure Rust** — no Python, no dynamic runtime dependencies.

> This makes it perfect for **minimal or embedded Linux environments**, like:

- Alpine-based containers
- IoT devices (e.g., routers, custom SBCs)
- Lightweight VMs
- Rust-based init systems

4.2.2. 2. Static Compilation

- Aya eBPF programs and userland control logic can be **statically compiled**.
- You can cross-compile both kernel and user parts for specific targets.
- This fits well with distroless or **statically linked embedded firmware images**.

4.2.3. 3. Fast, Safe, and Efficient

- Rust's safety guarantees extend to kernel/user communication.
- Aya generates **compact eBPF bytecode**, tuned for performance.
- More robust than hand-written C with raw libbpf.

4.2.4. 4. Async Capable Userland

- Aya's userspace side uses **Tokio (async)** by default.
- Efficient and scalable on resource-constrained CPUs.
- Runs fine in tight memory environments without GIL or CPython overhead.

—

4.2.5. 5. Fine-grained Control

- Aya supports:
  - Custom perf buffers
  - Direct `task_struct` inspection
  - Tracepoints, kprobes, uprobes, maps

- Without needing root in some configurations (via CAP_BPF / BPF LSM)

4.3. 🧱 When BCC Is Still Easier

Scenario	Use BCC?
Quick prototyping / scripting	✅ Yes
Rich kernel symbol introspection	✅ Yes
You already have Python + BCC stack	✅ Yes

4.4. 🚀 Summary

Feature	BCC	Aya (Rust)
Python Required	✅ Yes	❌ No
Embedded Friendly	❌ No	✅ Yes
Static Build	❌ Harder	✅ Easy
Rust Safe Code	❌ No	✅ Yes
Performance	Good	Great

- **Aya is ideal for embedded or production systems** where minimalism, performance, and Rust safety are valued.