Bridging Kernel Space and AI: Building an MCP Server for Linux Scheduler Observability

Daniel Hodges

Goal

Implement BPF enabled MCP tools

Plan

Building a BPF MCP Tool

scxtop: Scheduler MCP Tool

Best Practices

Simple Implementation Path

1 Tracepoint

Focused kernel instrumentation point

1 Ringbuffer

Single high-performance data channel

5 Tools

Concise set of operational utilities

Synchronous stdio

Direct, uncomplicated I/O stream

Building a BPF MCP Tool

Reference:
github.com/hodgesds/bpf-mcp-example

Topics not covered in this example

- BPF Event Types
- BPF Map Types
- Task Storage
- Stack Traces
- Sampling

Assumptions

Libbpf-rs

Modern Rust bindings for eBPF

JSON-RPC STDIO

Standard communication protocol

Many different BPF libraries/frameworks

Caution!

Do not let AI generate BPF bytecode in tools

Permissions/Security not considered in this example

Building a BPF MCP Tool

01

Define BPF events

Establish the data structures and event types for kernel monitoring.

02

Write BPF programs

Develop the C code that will run within the kernel context.

03

Set up ring buffer consumption

Configure high-performance data transfer from kernel to userspace.

04

Aggregate data for analysis

Process and summarize raw events into meaningful metrics.

05

Expose via MCP tools

Interface with the Model Context Protocol for AI-driven observability.

Define BPF Events

```c
enum event_type {
    SCHED_SWITCH_EVENT = 1,
};

struct event {
    __u32 type;
    __u32 cpu;
    __u64 ts;
    __u32 prev_pid;
    __u32 next_pid;
    __u32 prev_tgid;
    __u32 next_tgid;
    char prev_comm[TASK_COMM_LEN];
    char next_comm[TASK_COMM_LEN];
};
```

EVENT TYPES

event_type

General event categories

SPECIFIC EVENT DATA

latency_event

Individual trace instances

BPF Programs- Ringbuffers/Maps

```c
// bpf/main.bpf.c

struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 16 * 1024 * 1024);
} events SEC(".maps");
```

RING BUFFER

Ringbuf

→

High-performance event streaming to userspace

ADDITIONAL STORAGE

Maps

→

Persistent storage/aggregation of events (not covered)

BPF Programs- Event Handling

```
SEC("tp_btf/sched_switch")
int BPF_PROG(handle_sched_switch, bool preempt,
         struct task_struct *prev, struct task_struct *next)
{
   struct event *e;
   e = bpf_ringbuf_reserve(&events, sizeof(*e), 0);
   if (!e) return 0;

   e->type = SCHED_SWITCH_EVENT;
   e->cpu = bpf_get_smp_processor_id();
   e->ts = bpf_ktime_get_ns();
   e->prev_pid = BPF_CORE_READ(prev, pid);
   e->next_pid = BPF_CORE_READ(next, pid);
   e->prev_tgid = BPF_CORE_READ(prev, tgid);
   e->next_tgid = BPF_CORE_READ(next, tgid);
   BPF_CORE_READ_STR_INTO(&e->prev_comm, prev, comm);
   BPF_CORE_READ_STR_INTO(&e->next_comm, next, comm);

   bpf_ringbuf_submit(e, 0);
   return 0;
}
```

STEP 1: ALLOCATION

Reserve Ringbuf Space

Allocates memory in the shared ring buffer for the event.

STEP 2: POPULATION

Capture Context

Uses `BPF_CORE_READ` to safely extract process and timing info.

STEP 3: EMISSION

Submit to Userspace

Commits the reserved space, making it available for consumption.

Ring Buffer Configuration

Build the ring buffer and configure the userspace callback to process events.

```
let stats = Arc::new(Mutex::new(Stats::new(self_tgid)));
let stats_rb = stats.clone();
let mut rb_builder = RingBufferBuilder::new();
rb_builder.add(&skel.maps.events, move |data| {
  if data.len() >= size_of::<Event>() {
    let event: Event = unsafe { ... };
    if event.type_ == SCHED_SWITCH_EVENT {
      if let Ok(mut s) = stats_rb.try_lock() {
        s.record_switch(&event);
      }
    }
  }
  0
})?;
let ring_buf = rb_builder.build()?;
```

INITIALIZATION

Setup Shared State

Creates a thread-safe Mutex for storing scheduler statistics.

CONSUMPTION

Define Callback

Registers the logic to handle raw bytes arriving from the BPF map.

PROCESSING

Record Events

Extracts the event data and updates the local performance model.

Ring Buffer Consumption

Consume from the ringbuffer in a background thread

```
let epoll_fd = ring_buf.epoll_fd();
thread::spawn(move || {
  use std::os::fd::BorrowedFd;
  let fd = unsafe { BorrowedFd::borrow_raw(epoll_fd)
  };
  let mut pollfd = [nix::poll::PollFd::new(fd,
  nix::poll::PollFlags::POLLIN)];
  loop {
    if nix::poll::poll(&mut pollfd,
    nix::poll::PollTimeout::NONE).unwrap_or(0) > 0 {
      let _ = ring_buf.consume();
    }
  }
});
```

ASYNCHRONOUS

Background Thread

Offloads consumption from the main application loop to ensure responsiveness.

EFFICIENCY

Epoll Waking

Uses OS-level event notification to wake only when data is actually available.

LOOP

Infinite Consumer

Continuously drains the buffer as events are emitted from the kernel.

Aggregate Data

Transform raw kernel events into **structured userspace data**

```
#[repr(C)]
#[derive(Clone, Copy)]
struct Event {
    type_: u32, cpu: u32, ts: u64,
    prev_pid: u32, next_pid: u32,
    prev_comm: [u8; 16],
    next_comm: [u8; 16],
}

struct Stats {
    total_events: u64,
    per_cpu_switches: HashMap<u32, u64>,
    per_process: HashMap<u32, ProcessInfo>,
    start_time: Instant,
}
```

CORE DATA

libbpf-rs Generation

Event structs are automatically generated to match kernel memory layouts.

STORAGE

In-Memory Maps

Uses HashMaps for efficient per-CPU and per-process metric tracking.

PERFORMANCE

Userspace Compute

Offloads complex arithmetic and formatting from the eBPF verifier.

Aggregate Data

Aggregate raw events

```
impl Stats {
	fn record_switch(&mut self, event: &Event) {
		self.total_events += 1;
		*self.per_cpu_switches.entry(event.cpu).or_insert(0) += 1;

		let prev_comm = comm_to_string(&event.prev_comm);
		let next_comm = comm_to_string(&event.next_comm);

		self.per_process.entry(event.prev_tgid).or_insert_with(
		  || ProcessInfo {
		      comm: prev_comm, switches_from: 0, switches_to: 0,
		}).switches_from += 1;

		self.per_process.entry(event.next_tgid).or_insert_with(
		  || ProcessInfo {
		      comm: next_comm, switches_from: 0, switches_to: 0,
		}).switches_to += 1;
	}
}
```

CORE DATA

libbpf-rs Generation

Event structs match kernel memory layouts automatically.

STORAGE

In-Memory Maps

Efficient per-CPU and per-process metric tracking via HashMaps.

PERFORMANCE

Userspace Compute

Offloads complex arithmetic from the constrained eBPF verifier.

Expose via MCP


```
// JSON definitions for MCP tools
fn tool_definitions() -> Value {
  json!([
    {
      "name": "get_scheduling_stats",
      "description": "Get overall scheduling statistics...",
      "inputSchema": { "type": "object", "properties": {} }
    },
    {
      "name": "enable_collection",
      "description": "Enable BPF event collection...",
      "inputSchema": { "type": "object", "properties": {} }
    },
    {
      "name": "disable_collection",
      "description": "Disable BPF event collection...",
      "inputSchema": { "type": "object", "properties": {} }
    }
  ])
}
```

INTEGRATION

Tool Discovery

Exposes kernel metrics via standard JSON-RPC tool definitions.

MCP Attach BPF Program

```
// Handling the "enable_collection" tool
"enable_collection" => {
   if skel.links.handle_sched_switch.is_some() {
      return json!({ "content": [{ "type": "text", "text": "Already
      enabled" }] });
   }
   match skel.progs.handle_sched_switch.attach() {
      Ok(link) => {
         skel.links.handle_sched_switch = Some(link);
         let filter_tgid = stats.lock().unwrap().filter_tgid;
         *stats.lock().unwrap() = Stats::new(filter_tgid);
         json!({ "content": [{ "type": "text", "text": "Enabled, stats
         reset" }] })
      }
      Err(e) => {
         json!({ "content": [...], "isError": true })
      }
   }
}
```

LIFECYCLE MANAGEMENT

Dynamic Attachment

Uses libbpf-rs to attach BPF programs to kernel tracepoints on-demand.

STATE SYNCHRONIZATION

Atomic Reset

Locks userspace stats and re-initializes them upon successful BPF attachment.

ERROR HANDLING

Fault Tolerance

Gracefully handles attachment failures and already-enabled states via JSON responses.

Recap & Discussion

Building BPF-enabled MCP Tools

Define BPF Events

Write BPF Programs

Setup Ringbuffers

Aggregate events

Expose via MCP Tools

Open Questions

How to handle security?

Can this pattern scale to large machines?

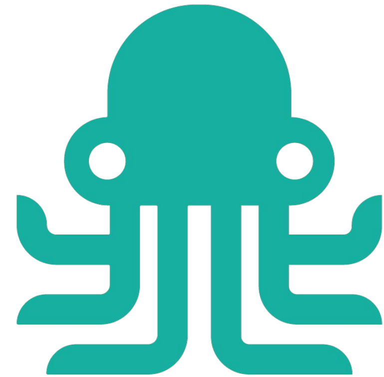
Large deployments vs. developer environments?

sched_ext: scheduler observability

sched_ext -> scheduler in BPF

How to observe?
    started with bpftrace

scxtop -> aggregate scheduler events



scxtop: BPF Scheduler Observability

USER SPACE

scxtop

TUI Mode
(ratatui)

MCP Mode
(JSON-RPC)

Trace Mode
(Perfetto)

Event Processing Layer

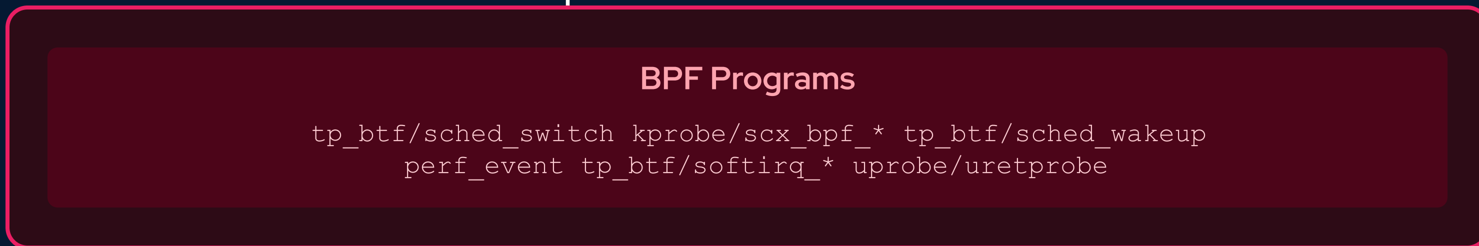
Analyzers

SharedStats

EventDispatch

Ring Buffer Polling (tokio)
RB0 RB1 RB2 RBn

KERNEL



filter — Processes (total: 422) — sample rate 128

TGID	Name	Last DSQ	Slice ns	Lat us Avg/Max	CPU	LLC	NUMA	Threads	CPU%
143084	claude	0x0	89513	0/0	2	0	0	23	24.0
143996	scxtop	0x0	14953	0/0	9	0	0	15	12.0
96986	Isolated	0x0	41914	0/0	4	0	0	40	8.02
98251	MainThre	0x0	116835	0/0	6	0	0	6	8.02
1584	chromium	0x0	43417	0/0	0	0	0	48	4.01
1095	kwin_way	0x0	81692	0/0	15	0	0	31	4.01
96970	Utility	0x0	34899	0/0	8	0	0	8	4.01
416	jbd2/dm-	0x0	1706	0/0	4	0	0	1	4.01
3638	tmux: se	0x0	194803	0/0	6	0	0	1	4.01
96540	firefox	0x0	57774	0/0	14	0	0	118	0.00
1310	plasmash	0x0	90665	0/0	15	0	0	91	0.00
144214	krunner	0x0	98063	0/0	11	0	0	47	0.00
3092	chromium	0x0	38634	0/0	5	0	0	30	0.00
96907	WebExten	0x0	6793	0/0	14	0	0	28	0.00
96690	Privileg		0	0/0	0			27	0.00
58110	chromium		0	0/0	5			26	0.00
96987	Web Cont		0	0/0	2			26	0.00
97000	Web Cont		0	0/0	4			26	0.00
97136	Web Cont	0x0	960	0/0	12	0	0	26	0.00
23855	chromium	0x0	2525	0/0	10	0	0	25	0.00
143680	chromium	0x0	70409	0/0	14	0	0	25	0.00
1846	chromium	0x0	1091	0/0	10	0	0	24	0.00

Node0 (cpu_total_util_percent) avg 4 max 28 min 0 — 250ms

CPU	Freq	Util	CPU	Freq	Util
CPU0	3.243GHz	8	CPU1	2.395GHz	0
CPU2	2.393GHz	4	CPU3	3.242GHz	0
CPU4	1.334GHz	4	CPU5	1.100GHz	0
CPU6	2.395GHz	4	CPU7	3.243GHz	0
CPU8	2.394GHz	7	CPU9	2.160GHz	4
CPU10	2.395GHz	4	CPU11	2.394GHz	0
CPU12	2.394GHz	8	CPU13	2.389GHz	0
CPU14	2.397GHz	4	CPU15	2.391GHz	3

Memory Statistics

Metric	Total	Free	Free%	Available	Cached	Buffers	Active	Inactive
Memory	27.05GB	13.84GB	51.2%	20.76GB	5.60GB	160.90MB	3.68GB	5.89GB

Metric	Total	Free	Free%	Cached	In	Out
Swap	27.05GB	24.98GB	92.4%	1.87MB	0	0

Metric	Minor	Major	Total
Page Faults	406	0	406

Network

Interface	RX Bytes	TX Bytes	RX Packets	TX Packets
wlan0	261B/s	234B/s	3/s	3/s
lo	0B/s	0B/s	0/s	0/s
enp1s0f0	0B/s	0B/s	0/s	0/s

Perfetto

Current Trace

scxtop_trace_0.proto (114 MB)

- Timeline
- Overview
- Data Explorer
- Query (SQL)
- Metrics
- Download

New Trace

- Open trace file
- Record new trace
- Open Android example
- Open Chrome example

Settings

- Settings
- Flags
- Plugins

Support

Documentation & Bugs

Convert trace

Convert to other formats

WSM 0 SW ON v53.0-e87365eda

Timeline Sync

Current Selection

CPU Sched Slice /usr/lib/firefox/firefox [22005]

Details

Process	/usr/lib/firefox/firefox [22005]
Thread	Isolated Web Co [22005] ↗
Cmdline	/usr/lib/firefox/firefox -contentproc -isForBrowser -prefsHandle 0:44460 -prefMapHandle 1:285400 -jsInitHandle 2:156136 -parentBuildID 20260323182030 -sandboxReporter 3 -chrootClient 4 -ipcHandle 5 -initialChannelId {f87a3ae3-a177-43d4-87d8-65ac66b3b9f5} -parentPid 18667 -crashReporter 6 -crashHelper 7 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser 32 tab
Start time	00:00:00.036 817 764
Duration	1ms 863µs 616ns
Priority	120
End State	Sleeping
SQL ID	sched[13919] ▾
Thread	Isolated Web Co 22005
Process	/usr/lib/firefox/firefox 22005

scxtop: MCP Analysis

Even CPUs (physical cores) are ~3x busier than **odd CPUs** (SMT siblings) — expected for a lightly-loaded system where the scheduler prefers primary threads.

Top Processes by Activity

Process	PID	Switches	Avg Latency	Max Latency
scxtop (2 threads)	177932, 178401	2,838	~5 µs	128 µs
claude (3 instances)	178377, 143084, 133019	753	4.1-5.1 µs	43 µs
kwin_wayland	1095	43	**13.6 µs**	**77 µs**
alacritty	8366	46	5.4 µs	14 µs
rcu_preempt	16	46	7.6 µs	14 µs
gfx_0.0.0 (GPU)	164620	52	3.5 µs	18 µs

Dispatch Queues (DSQs)

- **DSQ 0** (main): 3,845 enqueues / 3,811 dispatches (34 pending — small backlog)
- **Built-in SCX DSQ**: 3,629 enqueues / 3,659 dispatches (draining well)
- **Per-CPU DSQs**: minimal activity, all nearly empty

Key Takeaways

1. **System is very lightly loaded** (97.5% idle). The scheduler has no contention.
2. **scx_lavd is performing well** — wakeup-to-schedule latencies average ~5 µs across all CPUs, with rare spikes to ~128 µs (likely from deeper C-states).
3. **No cross-domain migrations** — tasks stay close to their data, good cache behavior.
4. **kwin_wayland has the highest average latency** (13.6 µs) — expected for a compositor that runs periodically and may wake from idle.
5. The biggest consumers of scheduling are **scxtop itself** and **claude** (this session), which is measurement overhead.

Problem: Multiple Events

MCP Dev Summit North America

Solution: Event Wrapping

Use a wrapper struct

Type enum for userspace parsing

Abstract common fields (type, size, ts, cpu)

```
struct bpf_event {
  int type;
  u32 size;
  u64 ts;
  u32 cpu;
  union {
    struct exit_event exit;
    struct fork_event fork;
    struct exec_event exec;
    ...
    struct sched_switch_event sched_switch;
    struct perf_sample_event perf_sample;
  } event;
};
```

Problem: Task state tracking

Solution

BPF_MAP_TYPE_TASK_STORAGE

Task-specific state tracking directly within the kernel, avoiding global lock contention and reducing overhead

```
static int update_task_ctx(
    struct task_struct *p, ...)
{
  struct task_ctx *tctx;
  if (!(tctx = try_lookup_task_ctx(p)))
    return -ENOENT;

  tctx->dsq_insert_time = bpf_ktime_get_ns();
  tctx->dsq_id = dsq;
  tctx->dsq_vtime = vtime;
  tctx->slice_ns = slice_ns;

  return 0;
}
```

Problem: PID not stable

Solution: Pointer to task_struct

Context

PID lifecycle is shorter than a `task_struct`. Reference task structs by converting pointers to stable 64-bit integers.

Reference

https://github.com/sched-ext/scx/issues/610

```
static __always_inline u64 t_to_tptr(
        struct task_struct *p)
{
  u64 tptr;
  int err;

  err = bpf_probe_read_kernel(&tptr,
              sizeof(tptr), &p);
  if (err)
    return 0;

  return tptr;
}
```

Problem: Dynamic Kernel Function Tracing

Attach to **any** kernel function dynamically at runtime, no predefined hook needed.

BPF C SIDE – STATIC KPROBE

```
SEC("kprobe/bpf_scx_reg")
int scx_sched_reg(struct pt_regs *ctx) {
  struct bpf_event *e = try_reserve_event(sizeof(*e));
  if (!e) return 0;
  e->type = SCHED_REG;
  e->ts = bpf_ktime_get_ns();
  bpf_ringbuf_submit(e, 0);
  return 0;
}
```

BPF C SIDE – GENERIC KPROBE (DYNAMIC)

```
SEC("kprobe/generic_kprobe")
int generic_kprobe(struct pt_regs *ctx) {
  struct bpf_event *e = try_reserve_event(sizeof(*e));
  if (!e) return 0;
  e->type = KPROBE;
  e->event.kprobe.ip = bpf_get_func_ip(ctx);
  bpf_ringbuf_submit(e, 0);
  return 0;
}
```

RUST SIDE – DYNAMIC ATTACHMENT BY FUNCTION NAME

```
// Attach to any kernel function at runtime
let link = skel.progs.generic_kprobe.attach_kprobe(false, "tcp_sendmsg")?;
// false = kprobe (entry), true = kretprobe (return)
```

Problem: Permissions

Challenges

BPF needs root/CAP_SYS_BPF+:
Loading BPF programs may require more privileges than desired for MCP tools

Local vs Remote:
May need to collect data on a remote machine

Debugging vs Continuous:
One off tool use vs always on collection

Solutions

Separate modes of operation

- `scxtop trace` -> collect data
- Tools with separate permissions
- Analysis -> no permissions

Separate processes

- Use RPC to control access

MCP Dev Summit North America

Problem: Event Aggregation

Challenges

High Event Volume:
`sched_switch` generates thousands of events/sec even when idle.

Throughput Pressure:
Ringbuffer sharding required to handle hundreds of MB/s.

Sampling/Tracing:
May need sampling/tracing that can handle tracing a sampled event.

Solutions

BPF Aggregation

- Per-CPU -> low overhead
- Aggregation in maps (bpftrace)
- Arenas -> unexplored

Userspace Aggregation

- in process aggregation (example)
- Perfetto -> structured processing
- Export to external data storage /tmp

MCP Dev Summit North America

Best Practices: BPF

Ringbuffer Sharding

Scale ring buffers across CPUs

Multiple RBs reduce contention

Cohort Sampling

Control overhead

Track complete task lifecycle

Task Storage

Task specific metadata

Timers

High resolution timing

Per-CPU Maps

Lock-free counters

No atomic contention

Syscall Programs

Userspace → BPF calls

`test_run()` for control

Best Practices: MCP Tools

BPF Lifecycle Control

Start and stop programs safely

Include duration and timeout mechanisms

Event Discovery

perf list, bpftrace -l, etc.

Must be searchable to limit token usage

Topology Awareness

Hardware-context for AI

Make AI aware of the underlying hardware

Data Storage Strategy

Handle high-frequency data

Need to manage large sample rates efficiently

Best Practices: Analysis

Dataset Integration

Often requires joining other data sets

Source Code
userspace/kernel

Performance
hardware counters, profiling

Events/Errors
dmesg, syslog etc

Tools & Agents

Aggregate across dimensions

May need interfaces to storage of raw or aggregated data for deeper analysis.

MCP
Dev Summit
North America

References/Other tools

sched_ext https://github.com/sched-ext/scx

libbpf-rs https://github.com/libbpf/libbpf-rs

scxtop- https://github.com/sched-ext/scx/tree/main/tools/scxtop

wprof- https://github.com/anakryiko/wprof

systing- https://github.com/josefbacik/systing

MCP Dev Summit
North America