

# Sprawozdanie z projektu na Oprogramowanie Systemowe

Łukasz Kołakowski 198000

28 Stycznia 2026

# 1 Opis środowiska

Planista został wykonany na system operacyjny Linux, wersja jądra wynosi 6.14 i tylko na tej wersji będzie działać mój planista. Dystrybucją Linuxa na, której robiłem projekt jest Linux Mint, nie jest to wymagane. Najlepiej by dystrybucja miała interfejs graficzny, co pozwoli na łatwe zauważenie działania planisty.

## 2 Zastosowane rozwiązanie techniczne

Implementacja *schedulera* nastąpiła przy pomocy frameworku eBPF wraz z dedykowanym rozszerzeniem do kernela *sched\_ext*<sup>1</sup>. Kod napisany jest w C, przy pomocy libbpf<sup>2</sup>, biblioteki implementującej BPF. Jest ona nowsza od BCC<sup>3</sup> i pozwala na większą ingerencję w struktury jądra. BCC jest dobry do monitorowania, nasłuchiwanie i zbierania informacji. Libbpf rozszerza te możliwości i ułatwia np. wyżej wspomniane dodawanie implementacji planisty, wymiana funkcjonalności systemu itd. .

## 3 Co udało się zrealizować

Udało mi się na początku zaimplementować minimalnego planistę jako test czy, *sched\_ext* działa jak powinien. Implementuje on politykę typu *round robin* ze zmienną porcją przydzielanego czasu. Kolejka jest wspólna dla wszystkich rdzeni procesora. Po czym postarałem się rozszerzyć planistę; dodatkowo zapisuje i wypisuje użytkownikowi informacje o procesach, którym przedziela czas. Zapisuje statystyki takie jak:

- przydzielony czas;
- pid;
- krótka nazwa procesu;
- łączny czas czekania na czas procesora;
- maksymalny czas czekania.

Zobaczyłem jak zachowuje się planista, jeżeli przydzielimy bardzo kawałek czasu (sekundy, setne sekund) lub jak będzie on bardzo mały (nanosekundny).

Dodatkowo chciałem zobaczyć czy da się zagłodzić niechciane zadania, celowo nie przydzielając im czasu procesora.

## 4 Architektura projektu

Projekt składa się praktycznie z dwóch plików, jeden zawiera kod wrzucany do jądra, realizujący wybraną politykę planisty. Drugi jest programem w *User space*, który załącza planistę, a potem wypisuje użytkownikowi informacje o procesach, którym przedzielany jest czas.

---

<sup>1</sup><https://github.com/sched-ext/scx>

<sup>2</sup><https://github.com/libbpf/libbpf>

<sup>3</sup><https://github.com/iovisor/bcc>

Kod jądra:

## 4.1 Zmienne, stałe i marka

```
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

// there is some problem with headers:
// vmlinux.h is kinda weird with that
extern s32 scx_bpf_create_dsq(u64 dsq_id, s32 node_id) __ksym;
extern void scx_bpf_dsq_insert(struct task_struct *p, u64 dsq_id, u64 slice,
                              u64 enq_flags) __ksym;
extern void scx_bpf_dispatch(struct task_struct *p, u64 dsq_id, u64 slice,
                              u64 enq_flags) __ksym;
extern void scx_bpf_consume(u64 dsq_id) __ksym;
extern s32 scx_bpf_dsq_nr_queued(u64 dsq_id) __ksym;
extern bool scx_bpf_dsq_move_to_local(u64 dsq_id) __ksym;
extern void scx_bpf_kick_cpu(s32 cpu, u64 flags) __ksym;
// extern __u64 (*const bpf_ktime_get_boot_ns)(void) = (void *)125;

// Define a shared Dispatch Queue (DSQ) ID
#define SHARED_DSQ_ID 2 // normal
#define PARKING_DSQ_ID 3 // for tasks we hate and don't want to run
#define DELAY_NS 1000000000ULL
char forbidden_name[6] = "hello";
int time_to_unpark = 0;

#define BPF_STRUCT_OPS(name, args...) \
    SEC("struct_ops/" #name) BPF_PROG(name, ##args)

#define BPF_STRUCT_OPS_SLEEPABLE(name, args...) \
    SEC("struct_ops.s/" #name) \
    BPF_PROG(name, ##args)

typedef struct global_sched_data {
    int call_function_counter;
    unsigned int last_used_index;
} global_sched_data;

typedef struct task_stats_ext {
    int call_function_counter;
    int last_used_index;

    u64 slice;
    int pid;
    int recent_used_cpu;
    long unsigned int last_switch_count;
    long unsigned int last_switch_time;

    char comm[16];
    u64 total_wait_ns;
    u64 max_wait_ns;
    u64 start_wait_ns;
    u64 wait_count;
} task_stats_ext;

// Array map definition
struct {
    __uint(type, 29); // <- this is BPF_MAP_TYPE_TASK_STORAGE
    __uint(map_flags, BPF_F_NO_PREALLOC);
    __type(key, int);
    __type(value, task_stats_ext);
} task_storage SEC(".maps");
```

```

struct parking_lot {
    struct bpf_timer timer;
};

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(max_entries, 1);
    __type(key, int);
    __type(value, struct parking_lot);
} timer_map SEC(".maps");
}

```

W tej części deklarujemy wszystkie zmienne z jakich będziemy korzystać. Był problem z funkcjami typu *scx\_bpf...* dlatego funkcje kernela są zadeklarowane u góry dodatkowo. Najważniejszym structem jest *task\_stats\_ext*, odpowiada on za dodatkową logikę naszego planisty. Nie jesteśmy w stanie zmieniać wartości zmiennych w jądrze, więc na potrzeby planisty eBPF dodali specjalną strukturę *BPF\_MAP\_TYPE\_STORAGE*, w której możemy przypisywać do każdego wątku dodatkowe parametry, na podstawie których będziemy dysponować czasem procesora.

## 4.2

```

// Initialize the scheduler by creating a shared dispatch queue (DSQ)
s32 BPF_STRUCT_OPS_SLEEPABLE(sched_init) {
    // All scx_ functions come from vmlinux.h
    scx_bpf_create_dsq(SHARED_DSQ_ID, -1);
    scx_bpf_create_dsq(PARKING_DSQ_ID, -1); // second queue

    return 0;
}

// Enqueue a task to the shared DSQ that wants to run,
// dispatching it with a time slice
int BPF_STRUCT_OPS(sched_enqueue, struct task_struct *p, u64 enq_flags) {

    u64 slice = 0u / scx_bpf_dsq_nr_queued(SHARED_DSQ_ID);

    if (p->pid == 6070 ||
        __builtin_memcmp(p->comm, forbidden_name, sizeof(forbidden_name)) == 0) {
        slice = 67;
        // scx_bpf_dsq_insert(p, SHARED_DSQ_ID, slice, enq_flags);
        bpf_printk("Parking %s for %d seconds...\n", forbidden_name,
            DELAY_NS / 1000000000);

        int key = 0;
        struct parking_lot *val = bpf_map_lookup_elem(&timer_map, &key);
        if (val) {
            bpf_timer_init(&val->timer, &timer_map, 1);
            bpf_timer_set_callback(&val->timer, timer_cb);
            bpf_timer_start(&val->timer, DELAY_NS, 0);
        }
        scx_bpf_dsq_insert(p, PARKING_DSQ_ID, slice, enq_flags);
    } else if (__builtin_memcmp(p->comm, forbidden_name,

```

```

        sizeof(forbidden_name)) == 0) {
    } else {
        // Calculate the time slice for the task based on the number of tasks in
        // the queue
        slice = 50000000u; // (scx_bpf_dsq_nr_queued(SHARED_DSQ_ID) + 2);
        scx_bpf_dsq_insert(p, SHARED_DSQ_ID, slice, enq_flags);
    }

    // ----- stats for listener -----

    // global sched data
    static global_sched_data g_sched_data = {0, 0};

    // reference to task map
    task_stats_ext *stats = bpf_task_storage_get(&task_storage, p, NULL,
                                                BPF_LOCAL_STORAGE_GET_F_CREATE);

    // stats
    if (stats) {
        stats->call_function_counter = g_sched_data.call_function_counter;
        stats->slice = slice;
        stats->pid = p->pid;
        stats->recent_used_cpu = p->recent_used_cpu;
        stats->last_switch_count = p->last_switch_count;
        stats->last_switch_time = p->last_switch_time;
        stats->start_wait_ns = bpf_ktime_get_boot_ns(); // now
        bpf_probe_read_kernel_str(stats->comm, sizeof(stats->comm), p->comm);
    }
    return 0;
}

// Dispatch a task from the shared DSQ to a CPU,
int BPF_STRUCT_OPS(sched_dispatch, s32 cpu, struct task_struct *prev) {
    if (time_to_unpark == 1) {
        bpf_printk("Dispatching from Parking");

        if (scx_bpf_dsq_move_to_local(PARKING_DSQ_ID)) {
            time_to_unpark = 0;
            return 0;
        }
    }
    scx_bpf_dsq_move_to_local(
        SHARED_DSQ_ID); // <- this function is different in 6.12 and 6.12
    return 0;
}

```

To jest serce całego projektu, cały planista składa się z 3 funkcji:

- inicjalizacja kolejki
- wrzucanie zadań do kolejki
- przerzucanie zadań z kolejki na procesor

Jest to minimalna liczba funkcji, które trzeba zaimplementować.

W inicjalizacji tworzymy dwie kolejki, są to *shared dispatch queue*, czyli wspólna kolejka na wszystkie procesory. Tworzymy je dwie, jedna dla normalnych procesów, druga dla procesów które będziemy chcieli intencjonalnie głodzić.

Do **sched\_enqueue** trafiają gotowe zadania do uruchomienia na procesorze, naszym zadaniem jest przydzielenie im czasu i wywołanie funkcji **scx\_bpf\_dsq\_insert**. Zadania które chcemy by wykonywały się normalnie trafiają do kolejki *SHARED\_DSQ\_ID*. Jeżeli nasze zadanie ma dane *pid* lub nazwę to, zostaną wrzucone do drugiej kolejki *PARKING\_DSQ\_ID*, jak nazwa wskazuje działa jak parking, na który będą trafiać zadania, które odstawiamy na bok na by sobie poczekały (na tyle długo na ile pozwala nam jądro) zanim dostaną czas procesora.

Zadanie trafia na procesor w funkcji **sched\_dispatch**, warto zauważyć, że nie mamy tutaj dostępu do zadania, które zostanie wrzucone. Mamy dostęp tylko do zadania wyłączonego i dla niego możemy ustawić parametry. Funkcja **scx\_bpf\_dsq\_move\_to\_local** weźmie zadania znajdujące się na szczycie kolejki i przypisze je do procesora, mamy tutaj możliwość sprecyzowania z której kolejki dostarczymy zadanie.

### 4.3 Głodzenie niechcianych zadań

Głodzenie zadań polega na przypisaniu zadań do wyżej wymienionej *PARKING\_DSQ\_ID*, nie można pominąć zadań przy przydziale czasu procesora, ponieważ planista zaimplementowany przy pomocy *sched\_ext* jest sprawdzany, czy nie jest wadliwy. Planista musi przydzielić czas procesora w ciągu 30s (można tę wartość zmniejszyć) od czasu oczekiwania. Jeżeli minie wymieniony wyżej okres czasu, planista zostanie wyrzucony z pamięci i zostanie na nowo uruchomiony standardowy planista oparty o drzewo czerwono czarne.

Gdy zadanie trafi do rezerwowej kolejki zostanie uruchomiony licznik, który po czasie (w moim przypadku 1 sekunda) ustawi flagę, która spowoduje, że przy najbliższej okazji procesorowi zostanie przydzielony czas z tej rezerwowej kolejki.

### 4.4 Kod po stronie użytkownika

```
#include "selective.skel.h"
#include <bpf/bpf.h>
#include <bpf/libbpf.h>
#include <stdbool.h>
#include <stdio.h>
#include <sys/resource.h>
#include <unistd.h>

typedef unsigned long long u64;
typedef unsigned int u32;

typedef struct task_stats_ext {
    int call_function_counter;
    int last_used_index;

    u64 slice;
    int pid;
    int recent_used_cpu;
    long unsigned int last_switch_count;
```

```

    long unsigned int last_switch_time;

    // u64 voluntary_switch_count;
    // u64 involuntary_switch_count;

    u64 exec_max;

    u64 total_wait_ns;
    u64 max_wait_ns;
    u64 start_wait_ns;
    u64 wait_count;
} task_stats_ext;

static int libbpf_print_fn(enum libbpf_print_level level, const char *format,
                           va_list args)
{
    return vfprintf(stderr, format, args);
}

int main(int argc, char **argv)
{
    struct selective_bpf *skel;
    int err;

    /* Set up libbpf errors and debug info callback */
    libbpf_set_print(libbpf_print_fn);

    /* Open BPF application */
    skel = selective_bpf__open();
    if (!skel) {
        fprintf(stderr, "Failed to open BPF skeleton\n");
        return 1;
    }

    /* ensure BPF program only handles write() syscalls from our process */
    // skel->bss->my_pid = getpid();
    /* Load & verify BPF programs */
    err = selective_bpf__load(skel);
    if (err) {
        fprintf(stderr, "Failed to load and verify BPF skeleton\n");
        goto cleanup;
    }

    printf("attaching sched");
    struct bpf_link *link = bpf_map__attach_struct_ops(skel->maps.sched_ops);
    if (!link) {
        fprintf(stderr, "Failed to register scheduler: %d\n", -errno);
        goto cleanup;
    }
    skel->links.sched_ops = link;

    // int map_fd = bpf_map__fd(skel->maps.task_storage);

    /* Attach tracepoint handler */

```

```

err = selective_bpf__attach(skel);
if (err) {
    fprintf(stderr, "Failed to attach BPF skeleton\n");
    goto cleanup;
}

printf("Successfully started! The selective scheduler \n");

struct bpf_link *iter_link =
    bpf_program__attach_iter(skel->progs.dump_task_stats, NULL);

if (!iter_link) {
    fprintf(stderr, "Failed to attach iter_link\n");
    goto cleanup;
}

fprintf(stderr, ".");
while (true) {

    int iter_fd = bpf_iter_create(bpf_link__fd(iter_link));
    if (iter_fd < 0) {
        fprintf(stderr, "Failed to create iterator File Descriptor\n");
        break;
    }

    char buf[8192];
    int n = 0;
    fprintf(stderr, ".");
    while ((n = read(iter_fd, buf, sizeof(buf))) > 0) {
        write(STDOUT_FILENO, buf, n);
    }
    close(iter_fd);
    sleep(10);
}

cleanup:
selective_bpf__destroy(skel);
return -err;
}

```

Po stronie użytkownika kod nie jest ciekawy, zawiera on inicjalizację programu, próbuje załadować naszego planistę, potem próbuje uzyskać dostęp do pliku, w którym zapisywane będą dodatkowe informacje o zadaniach. Potem w pętli okresowo co 10 sekund odczytuje zawartość pliku, i wypisuje ją użytkownikowi.

Warty wspomnienia jest ten mechanizm komunikacji pomiędzy programem użytkownikiem a programem jądra. Programy eBPF jeżeli chcą zapisać jakieś informacje muszą korzystać w własnych dedykowanych struktur, są one zapisywane jako plik zapisany w pamięci RAM. Nie mogą nasze funkcje też wywoływać funkcji typu *malloc()*. Struktury danych jakie eBPF oferuje są różnorodne, ale zawiera szereg ograniczeń. Nasz program korzysta z dedykowanej struktury danych typu *BPF\_MAP\_TYPE\_TASK\_STORAGE*, która tworzy mapę typu *pid* -> *task\_stats\_ext*. Program użytkownika może ją odczytać, ale nie została ta struktura do tego stworzona, więc by było to możliwe trzeba utworzyć specjalny iterator po stronie kernela, który

za nas będzie po niej przechodzić, a my będziemy na bieżąco z czytywać i zapisywać informacje u siebie.

```
// helper for getting task_storage
SEC("iter/task")
int dump_task_stats(struct bpf_iter__task *ctx) {
    struct seq_file *seq = ctx->meta->seq;
    struct task_struct *task = ctx->task;
    if (!task)
        return 0;

    task_stats_ext *stats = bpf_task_storage_get(&task_storage, task, 0, 0);
    if (!stats)
        return 0;

    if (stats->pid == 0) {
        return 0;
    }

    if (stats->pid == 2137 || __builtin_memcmp(stats->comm, forbidden_name,
                                              sizeof(forbidden_name)) == 0) {

        BPF_SEQ_PRINTF(seq,
            "----- Name: %-16s Pid: %-8d slice: %-10lld "
            "max_wait(ms): %-10lld "
            "total_wait(ms): %-10lld"
            "wait_count: %-10lld\n",
            stats->sched_excomm, stats->pid, stats->slice,
            (stats->max_wait_ns) / 1000000,
            stats->total_wait_ns / 1000000, stats->wait_count);

        return 0;
    }

    BPF_SEQ_PRINTF(seq,
        "Name: %-16s Pid: %-6d slice: %-10lld max_wait(ms): %-5lld "
        "total_wait(ms): %-5lld"
        "wait_count: %-4lld\n",
        stats->comm, stats->pid, stats->slice,
        (stats->max_wait_ns) / 1000000, stats->total_wait_ns / 1000000,
        stats->wait_count);

    return 0;
};
```

## 5 Kompilacja programu

### 5.1 Wymagania systemowe

- **Wersja jądra:** 6.14 (wersje 6.13 oraz 6.12 wymagają drobnych poprawek w kodzie ze względu na zmiany w nazewnictwie funkcji).
- **Kompilator:** Clang z obsługą BPF.
- **Narzędzia BPF:**

```
sudo apt install clang llvm libelf-dev libz-dev
```

- Najnowsza biblioteka libbpf:

```
git clone https://github.com/libbpf/libbpf.git
cd libbpf/src && make
```

## 5.2 Budowanie i uruchamianie

1. Umieść bibliotekę libbpf w głównym folderze (root) tego repozytorium.
2. Wygeneruj nagłówki jądra Linux:

```
cd scheduler_ext/selective/
bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
```

3. Skompiluj kod jądra (eBPF):

```
clang -target bpf -g -O2 -c selective.bpf.c -o selective.bpf.o -I.
```

4. Wygeneruj szkielec (skeleton) dla programu w przestrzeni użytkownika:

```
bpftool gen skeleton selective.bpf.o > selective.skel.h
```

5. Skompiluj program użytkownika:

```
clang -g -O2 -Wall listener.c -o listener ../../libbpf/src/libbpf.a -lelf -lz
```

6. Uruchom planistę:

```
./listener
```

## 6 Działanie programu

Program działa od razu po uruchomieniu i zamienia aktualnie działającego planistę na naszego. Działanie programu będzie powiedziane opisowo ze względu na to, że efekty działania widać najlepiej poprzez obserwację interfejsu graficznego, co jest ciężkie do pokazania w pliku *pdf*.

### 6.1 Normalne działanie programu

Porcja czasu jest wyliczana następująco:

```
slice = 50000000u / (scx_bpf_dsq_nr_queued(SHARED_DSQ_ID) + 1);
```

Przydzielony czas to 50 milisekund podzielone przez liczbę aktualnie czekających procesów. Procesor na którym wykonywane są obliczenia to *Intel Core i5-1240P* posiadający 16 rdzeni. Przed włączenie planisty zostało uruchomione 8 procesów za pomocą komendy *stress*. Planista działał przez 60 sekund.

Name: lazygit	Pid: 3138	slice: 25000000	max_wait(ms): 0	total_wait(ms): 0	wait_count: 4	slice sum (ms): 91
Name: lazygit	Pid: 3140	slice: 50000000	max_wait(ms): 0	total_wait(ms): 0	wait_count: 8	slice sum (ms): 350
Name: glean.dispatche	Pid: 4780	slice: 50000000	max_wait(ms): 7	total_wait(ms): 82	wait_count: 93	slice sum (ms): 2922
Name: Privileged Cont	Pid: 4877	slice: 25000000	max_wait(ms): 0	total_wait(ms): 0	wait_count: 1	slice sum (ms): 25
Name: IPC I/O Child	Pid: 4883	slice: 25000000	max_wait(ms): 0	total_wait(ms): 0	wait_count: 2	slice sum (ms): 37
Name: kworker/12:0	Pid: 5778	slice: 50000000	max_wait(ms): 50	total_wait(ms): 3074	wait_count: 144	slice sum (ms): 4418
Name: kworker/14:1	Pid: 6289	slice: 25000000	max_wait(ms): 52	total_wait(ms): 2747	wait_count: 153	slice sum (ms): 4457
Name: kworker/8:2	Pid: 6637	slice: 50000000	max_wait(ms): 51	total_wait(ms): 2978	wait_count: 146	slice sum (ms): 4261
Name: kworker/3:1	Pid: 6733	slice: 50000000	max_wait(ms): 50	total_wait(ms): 1088	wait_count: 47	slice sum (ms): 1283
Name: kworker/10:0	Pid: 7409	slice: 50000000	max_wait(ms): 52	total_wait(ms): 1740	wait_count: 97	slice sum (ms): 2731
Name: kworker/11:1	Pid: 7729	slice: 50000000	max_wait(ms): 50	total_wait(ms): 1689	wait_count: 93	slice sum (ms): 2268
Name: kworker/0:1	Pid: 7873	slice: 16666666	max_wait(ms): 50	total_wait(ms): 3422	wait_count: 153	slice sum (ms): 4883
Name: kworker/2:0	Pid: 7938	slice: 16666666	max_wait(ms): 51	total_wait(ms): 3519	wait_count: 167	slice sum (ms): 5237
Name: FTP_1_4	Pid: 7941	slice: 25000000	max_wait(ms): 4	total_wait(ms): 7	wait_count: 4	slice sum (ms): 72
Name: FTP_1_5	Pid: 7943	slice: 50000000	max_wait(ms): 3	total_wait(ms): 8	wait_count: 4	slice sum (ms): 141
Name: kworker/4:3	Pid: 7944	slice: 25000000	max_wait(ms): 50	total_wait(ms): 2957	wait_count: 143	slice sum (ms): 4643
Name: kworker/6:2	Pid: 8264	slice: 25000000	max_wait(ms): 50	total_wait(ms): 2376	wait_count: 130	slice sum (ms): 4133
Name: kworker/5:1	Pid: 8446	slice: 25000000	max_wait(ms): 50	total_wait(ms): 1022	wait_count: 46	slice sum (ms): 1496
Name: kworker/9:0	Pid: 8814	slice: 50000000	max_wait(ms): 50	total_wait(ms): 2280	wait_count: 120	slice sum (ms): 3114
Name: kworker/5:0	Pid: 8894	slice: 16666666	max_wait(ms): 0	total_wait(ms): 0	wait_count: 3	slice sum (ms): 66
Name: stress	Pid: 8961	slice: 50000000	max_wait(ms): 12	total_wait(ms): 2678	wait_count: 1539	slice sum (ms): 56326
Name: stress	Pid: 8962	slice: 50000000	max_wait(ms): 10	total_wait(ms): 2683	wait_count: 1554	slice sum (ms): 56319
Name: stress	Pid: 8963	slice: 50000000	max_wait(ms): 13	total_wait(ms): 2750	wait_count: 1568	slice sum (ms): 56244
Name: stress	Pid: 8964	slice: 50000000	max_wait(ms): 15	total_wait(ms): 2671	wait_count: 1546	slice sum (ms): 56304
Name: stress	Pid: 8965	slice: 50000000	max_wait(ms): 12	total_wait(ms): 2634	wait_count: 1549	slice sum (ms): 56349
Name: stress	Pid: 8966	slice: 50000000	max_wait(ms): 12	total_wait(ms): 2708	wait_count: 1569	slice sum (ms): 56258
Name: stress	Pid: 8967	slice: 50000000	max_wait(ms): 15	total_wait(ms): 2768	wait_count: 1581	slice sum (ms): 56226
Name: stress	Pid: 8968	slice: 50000000	max_wait(ms): 16	total_wait(ms): 2615	wait_count: 1544	slice sum (ms): 56370
Name: kworker/6:3	Pid: 8988	slice: 50000000	max_wait(ms): 50	total_wait(ms): 730	wait_count: 49	slice sum (ms): 1764
Name: kworker/12:1	Pid: 9224	slice: 16666666	max_wait(ms): 44	total_wait(ms): 44	wait_count: 1	slice sum (ms): 16
Name: StreamTrans #56	Pid: 9258	slice: 25000000	max_wait(ms): 1	total_wait(ms): 1	wait_count: 1	slice sum (ms): 25
Name: StreamTrans #28	Pid: 9262	slice: 50000000	max_wait(ms): 1	total_wait(ms): 1	wait_count: 1	slice sum (ms): 50
Name: kworker/5:2	Pid: 9264	slice: 16666666	max_wait(ms): 50	total_wait(ms): 512	wait_count: 37	slice sum (ms): 1199
Name: StreamTrans #46	Pid: 9267	slice: 50000000	max_wait(ms): 0	total_wait(ms): 0	wait_count: 1	slice sum (ms): 50
Name: StreamTrans #121	Pid: 9274	slice: 50000000	max_wait(ms): 7	total_wait(ms): 15	wait_count: 6	slice sum (ms): 275
Name: StreamTrans #122	Pid: 9275	slice: 25000000	max_wait(ms): 2	total_wait(ms): 2	wait_count: 1	slice sum (ms): 25

Rysunek 1: Przedstawia output z statystykami na temat działających programów, zbierane przez 60 sekund

- Name - Krótka nazwa procesu.
- Pid - Identyfikator procesu (*Process IDentifier*).
- slice - Czas przydzielony zadaniu w nanosekundach.
- max\_wait - Największy czas oczekiwania w kolejce.
- total\_wait - Łączny czas oczekiwania w kolejce, liczone od startu programu.
- wait\_count - Liczba mówiąca ile razy proces był w kolejce.
- slice sum - Suma przydzielonych czasów do zadania, liczone od startu programu.

Jeżeli chcemy wyliczyć efektywność naszego planisty wystarczy podzielić *slice\_sum* przez czas jaki program działał, w naszym przypadku 60s. Jest to wzór przybliżony wzór nieuwzględniający wyłączenia, ale na potrzeby tego sprawozdania jest wystarczający.

$$\text{efektywność} = \frac{\text{slice sum}}{\text{czas działania programu}} = 94\%$$

## 6.2 Działanie dla dużego *slice'a*

Jeżeli ustawimy kwant czasu procesora na wartość 1 sekundy, po włączeniu planisty nic się nie dzieje. Wszystkie elementy interfejsu są responsywne i można korzystać z systemu. Wynika to z faktu, że procesy systemowe i graficzne na Linuxie nie działają ciągle, tylko okresowo lub czekają na akcję użytkownika, dobrowolnie oddają czas. Komputer staje się nie użyteczny jeżeli włączymy programy, które nie oddają dobrowolnie czasu. Jeżeli uruchomimy komendę:

```
stereess -c 16
```

utworzymy 16 procesów wykonywających w pętli obliczenia. Spowoduje to, że wszystkie rdzenie procesora na moim komputerze zostaną zajęte i wtedy system operacyjny się zawiesi.

```
Name: glean.dispatche Pid: 4780 slice: 1000000000 max_wait(ms): 14 total_wait(ms): 318 wait_count: 236 slice sum (ms): 236000
Name: kworker/12:0 Pid: 5778 slice: 1000000000 max_wait(ms): 1000 total_wait(ms): 18900 wait_count: 39 slice sum (ms): 39000
Name: kworker/14:1 Pid: 6289 slice: 1000000000 max_wait(ms): 984 total_wait(ms): 25160 wait_count: 65 slice sum (ms): 65000
Name: kworker/8:2 Pid: 6637 slice: 1000000000 max_wait(ms): 968 total_wait(ms): 6884 wait_count: 60 slice sum (ms): 60000
Name: kworker/3:1 Pid: 6733 slice: 1000000000 max_wait(ms): 999 total_wait(ms): 2292 wait_count: 29 slice sum (ms): 29000
Name: kworker/6:0 Pid: 7026 slice: 1000000000 max_wait(ms): 1000 total_wait(ms): 7893 wait_count: 28 slice sum (ms): 28000
Name: kworker/10:0 Pid: 7409 slice: 1000000000 max_wait(ms): 1000 total_wait(ms): 12164 wait_count: 50 slice sum (ms): 50000
Name: kworker/11:1 Pid: 7729 slice: 1000000000 max_wait(ms): 956 total_wait(ms): 5776 wait_count: 70 slice sum (ms): 70000
Name: kworker/0:1 Pid: 7873 slice: 1000000000 max_wait(ms): 995 total_wait(ms): 27674 wait_count: 59 slice sum (ms): 59000
Name: kworker/2:0 Pid: 7938 slice: 1000000000 max_wait(ms): 996 total_wait(ms): 36090 wait_count: 54 slice sum (ms): 55000
Name: FTP_1_4 Pid: 7941 slice: 1000000000 max_wait(ms): 0 total_wait(ms): 1 wait_count: 4 slice sum (ms): 4000
Name: FTP_1_5 Pid: 7943 slice: 1000000000 max_wait(ms): 12 total_wait(ms): 44 wait_count: 11 slice sum (ms): 11000
Name: kworker/4:3 Pid: 7944 slice: 1000000000 max_wait(ms): 974 total_wait(ms): 13225 wait_count: 65 slice sum (ms): 66000
Name: warpinator Pid: 8223 slice: 1000000000 max_wait(ms): 0 total_wait(ms): 0 wait_count: 1 slice sum (ms): 1000
Name: kworker/6:2 Pid: 8264 slice: 1000000000 max_wait(ms): 1000 total_wait(ms): 3002 wait_count: 3 slice sum (ms): 3000
Name: kworker/9:0 Pid: 8814 slice: 1000000000 max_wait(ms): 976 total_wait(ms): 6246 wait_count: 74 slice sum (ms): 75000
Name: kworker/5:0 Pid: 8894 slice: 1000000000 max_wait(ms): 0 total_wait(ms): 0 wait_count: 5 slice sum (ms): 5000
Name: stress Pid: 8961 slice: 1000000000 max_wait(ms): 2 total_wait(ms): 9 wait_count: 60 slice sum (ms): 60000
Name: stress Pid: 8962 slice: 1000000000 max_wait(ms): 4 total_wait(ms): 15 wait_count: 60 slice sum (ms): 60000
Name: stress Pid: 8963 slice: 1000000000 max_wait(ms): 7 total_wait(ms): 25 wait_count: 60 slice sum (ms): 60000
Name: stress Pid: 8964 slice: 1000000000 max_wait(ms): 2 total_wait(ms): 11 wait_count: 60 slice sum (ms): 60000
Name: stress Pid: 8965 slice: 1000000000 max_wait(ms): 0 total_wait(ms): 3 wait_count: 60 slice sum (ms): 60000
Name: stress Pid: 8966 slice: 1000000000 max_wait(ms): 2 total_wait(ms): 5 wait_count: 60 slice sum (ms): 60000
Name: stress Pid: 8967 slice: 1000000000 max_wait(ms): 2 total_wait(ms): 13 wait_count: 60 slice sum (ms): 60000
Name: stress Pid: 8968 slice: 1000000000 max_wait(ms): 4 total_wait(ms): 10 wait_count: 60 slice sum (ms): 60000
Name: kworker/12:1 Pid: 9224 slice: 1000000000 max_wait(ms): 1000 total_wait(ms): 17650 wait_count: 32 slice sum (ms): 32000
Name: kworker/5:2 Pid: 9264 slice: 1000000000 max_wait(ms): 278 total_wait(ms): 290 wait_count: 74 slice sum (ms): 74000
Name: kworker/6:1 Pid: 9326 slice: 1000000000 max_wait(ms): 1000 total_wait(ms): 9468 wait_count: 84 slice sum (ms): 84000
Name: listener Pid: 9428 slice: 1000000000 max_wait(ms): 0 total_wait(ms): 0 wait_count: 1 slice sum (ms): 1000
Name: StreamTrans #125 Pid: 9442 slice: 1000000000 max_wait(ms): 3 total_wait(ms): 4 wait_count: 2 slice sum (ms): 2000
Name: StreamTrans #46 Pid: 9444 slice: 1000000000 max_wait(ms): 1 total_wait(ms): 1 wait_count: 1 slice sum (ms): 1000
Name: StreamTrans #51 Pid: 9445 slice: 1000000000 max_wait(ms): 0 total_wait(ms): 0 wait_count: 1 slice sum (ms): 1000
Name: StreamTrans #64 Pid: 9446 slice: 1000000000 max_wait(ms): 2 total_wait(ms): 2 wait_count: 1 slice sum (ms): 1000
Name: StreamTrans #42 Pid: 9463 slice: 1000000000 max_wait(ms): 3 total_wait(ms): 3 wait_count: 1 slice sum (ms): 1000
Name: StreamTrans #47 Pid: 9464 slice: 1000000000 max_wait(ms): 1 total_wait(ms): 1 wait_count: 1 slice sum (ms): 1000
Name: StreamTrans #46 Pid: 9470 slice: 1000000000 max_wait(ms): 2 total_wait(ms): 2 wait_count: 1 slice sum (ms): 1000
Lukasz@Lukasz-Zenbook:~/Desktop/vscody/C-uper_programming/BPF_Scheduler/scheduler_ext/selective$
```

Rysunek 2: Przedstawia output z stylizacjami, dla stałej porcji 1s, przez 60 sekund działania programu

efektywność = 100%

Nasz wzór jest trochę zbyt optymistyczny *slice sum* mówi ile czasu my chcieliśmy dać procesowi nie uwzględniają czasu czekania i czasu potrzebnego na wywołanie procesu, ale po uwzględnieniu tych czynników efektywność byłaby rzędu ponad 99%.

Wydaje się, że duże porcje czasu są lepsze niż w normalnym przypadku, jednakże widać, że inne procesy musiały czekać całą sekundę, z tego wynika brak responsywności interfejsu graficznego. Niektóre procesy były niejako zagładzane, ponieważ za dużo czasu spędzały w kolejce a praca jaką musiały wykonać pewnie nie przekraczały paru nanosekund.

## 6.3 Działanie dla bardzo małego *slice'a*

Jeżeli ustawimy przydział czasu na bardzo mały rzędu nanosekund, system zostanie reaktywny bardzo długo.

```

Name: Web Content      Pid: 5701  slice: 10000  max_wait(ms): 0  total_wait(ms): 0  wait_count: 2  slice sum (ms): 0
Name: IPC I/O Child    Pid: 5707  slice: 10000  max_wait(ms): 0  total_wait(ms): 0  wait_count: 1  slice sum (ms): 0
Name: Timer            Pid: 5720  slice: 10000  max_wait(ms): 0  total_wait(ms): 0  wait_count: 1  slice sum (ms): 0
Name: kworker/8:2      Pid: 6637  slice: 10000  max_wait(ms): 2  total_wait(ms): 149  wait_count: 137  slice sum (ms): 1
Name: kworker/3:1      Pid: 6733  slice: 10000  max_wait(ms): 19  total_wait(ms): 126  wait_count: 76  slice sum (ms): 0
Name: kworker/6:0      Pid: 7026  slice: 10000  max_wait(ms): 2  total_wait(ms): 84  wait_count: 78  slice sum (ms): 0
Name: Web Content      Pid: 7282  slice: 10000  max_wait(ms): 0  total_wait(ms): 0  wait_count: 2  slice sum (ms): 0
Name: IPC I/O Child    Pid: 7288  slice: 10000  max_wait(ms): 0  total_wait(ms): 0  wait_count: 1  slice sum (ms): 0
Name: Timer            Pid: 7302  slice: 10000  max_wait(ms): 0  total_wait(ms): 0  wait_count: 1  slice sum (ms): 0
Name: kworker/11:1     Pid: 7729  slice: 10000  max_wait(ms): 2  total_wait(ms): 119  wait_count: 110  slice sum (ms): 1
Name: kworker/0:1      Pid: 7873  slice: 10000  max_wait(ms): 11  total_wait(ms): 185  wait_count: 137  slice sum (ms): 1
Name: kworker/2:0      Pid: 7938  slice: 10000  max_wait(ms): 2  total_wait(ms): 157  wait_count: 118  slice sum (ms): 1
Name: FTP_1_4          Pid: 7941  slice: 10000  max_wait(ms): 0  total_wait(ms): 1  wait_count: 8  slice sum (ms): 0
Name: FTP_1_5          Pid: 7943  slice: 10000  max_wait(ms): 0  total_wait(ms): 0  wait_count: 4  slice sum (ms): 0
Name: kworker/4:3      Pid: 7944  slice: 10000  max_wait(ms): 3  total_wait(ms): 167  wait_count: 128  slice sum (ms): 1
Name: kworker/9:0      Pid: 8814  slice: 10000  max_wait(ms): 2  total_wait(ms): 145  wait_count: 125  slice sum (ms): 1
Name: kworker/5:0      Pid: 8894  slice: 10000  max_wait(ms): 3  total_wait(ms): 112  wait_count: 69  slice sum (ms): 0
Name: stress           Pid: 8961  slice: 10000  max_wait(ms): 6  total_wait(ms): 29281wait_count: 32700 slice sum (ms): 327
Name: stress           Pid: 8962  slice: 10000  max_wait(ms): 6  total_wait(ms): 29225wait_count: 32698 slice sum (ms): 326
Name: stress           Pid: 8963  slice: 10000  max_wait(ms): 6  total_wait(ms): 29287wait_count: 32687 slice sum (ms): 326
Name: stress           Pid: 8964  slice: 10000  max_wait(ms): 6  total_wait(ms): 29273wait_count: 32694 slice sum (ms): 326
Name: stress           Pid: 8965  slice: 10000  max_wait(ms): 6  total_wait(ms): 29292wait_count: 32624 slice sum (ms): 326
Name: stress           Pid: 8966  slice: 10000  max_wait(ms): 6  total_wait(ms): 29246wait_count: 32655 slice sum (ms): 326
Name: stress           Pid: 8967  slice: 10000  max_wait(ms): 5  total_wait(ms): 29265wait_count: 32679 slice sum (ms): 326
Name: stress           Pid: 8968  slice: 10000  max_wait(ms): 6  total_wait(ms): 29243wait_count: 32685 slice sum (ms): 326
Name: kworker/12:1     Pid: 9224  slice: 10000  max_wait(ms): 2  total_wait(ms): 124  wait_count: 116  slice sum (ms): 1
Name: kworker/10:2     Pid: 9557  slice: 10000  max_wait(ms): 2  total_wait(ms): 107  wait_count: 99  slice sum (ms): 0
Name: kworker/14:0     Pid: 9773  slice: 10000  max_wait(ms): 1  total_wait(ms): 141  wait_count: 152  slice sum (ms): 1
Name: kworker/6:3      Pid: 9924  slice: 10000  max_wait(ms): 4  total_wait(ms): 130  wait_count: 112  slice sum (ms): 1
Name: Web Content      Pid: 9957  slice: 10000  max_wait(ms): 0  total_wait(ms): 0  wait_count: 2  slice sum (ms): 0
Name: IPC I/O Child    Pid: 9963  slice: 10000  max_wait(ms): 0  total_wait(ms): 0  wait_count: 1  slice sum (ms): 0
Name: Timer            Pid: 9975  slice: 10000  max_wait(ms): 0  total_wait(ms): 0  wait_count: 1  slice sum (ms): 0
Name: StreamT-ns #143  Pid: 10147 slice: 10000  max_wait(ms): 2  total_wait(ms): 194  wait_count: 168  slice sum (ms): 1
Name: StreamT-ns #144  Pid: 10148 slice: 10000  max_wait(ms): 0  total_wait(ms): 3  wait_count: 5  slice sum (ms): 0
Name: StreamTrans #54  Pid: 10150 slice: 10000  max_wait(ms): 1  total_wait(ms): 3  wait_count: 6  slice sum (ms): 0
Name: StreamTrans #50  Pid: 10168 slice: 10000  max_wait(ms): 0  total_wait(ms): 1  wait_count: 2  slice sum (ms): 0

```

Rysunek 3: Stała porcja czasu 10 mikrosekund, przez 60 sekund działania programu

10 mikrosekund jest to wystarczająco dużo czasu by zaszło przełączenie kontekstu, ale mimo to *slice sum* daje zaniżone wyniki, ponieważ przełączenie kontekstu nie nastąpi od razu, a w odpowiednim momencie zwanym *preemption point*. W naszym przypadku wydajniejsze będzie zobaczenie ile czasu spędziłyśmy czekając w stosunku do czasu jaki proces miał pracować:

$$\text{efektywność} = 49\%$$

Łatwo zauważyć, że dla małej porcji czasu nasz system zamiast dać zadaniom pracować, skupia się na przełączaniu wątków co zmniejsza efektywność pracy komputera.

## 6.4 Zagładzanie zadań

Zaimplementowane zostało intencjonalne zagładzanie niechcianych zadań, na podstawie jego pid lub krótkiej nazwy. Wyżej opisany mechanizm działa jak należy, i dla testowego programu wypisującego „*Hello world*” w pętli, program jest w stanie w swoim kawałku czasu wyświetlić około 100 napisów na możliwie najkrótszym przydziale czasu procesora. Ujawnia się tutaj tzw. *punkt wywłaszczania*. Proces na Linuxie nie zostanie wywłaszczony od razu, gdy jego czas minie. Zostanie to zrobione przy najbliższej możliwej okazji w odpowiednim miejscu:

- Powrocie do przestrzeni użytkownika.
- Powrocie do przestrzeni jądra.

Name: kworker/13:1	Pid: 14426	slice: 25000000	max_wait(ms): 3	total_wait(ms): 8	wait_count: 93	slice sum (ms): 4057
Name: kworker/6:0	Pid: 19824	slice: 50000000	max_wait(ms): 13	total_wait(ms): 32	wait_count: 102	slice sum (ms): 4830
Name: kworker/8:0	Pid: 26204	slice: 50000000	max_wait(ms): 0	total_wait(ms): 6	wait_count: 82	slice sum (ms): 3574
Name: lazygit	Pid: 27473	slice: 50000000	max_wait(ms): 0	total_wait(ms): 0	wait_count: 1	slice sum (ms): 50
Name: kworker/11:2	Pid: 28172	slice: 50000000	max_wait(ms): 0	total_wait(ms): 0	wait_count: 6	slice sum (ms): 208
Name: kworker/2:2	Pid: 28260	slice: 50000000	max_wait(ms): 3	total_wait(ms): 10	wait_count: 79	slice sum (ms): 3786
Name: kworker/14:0	Pid: 30947	slice: 50000000	max_wait(ms): 0	total_wait(ms): 4	wait_count: 74	slice sum (ms): 3308
Name: kworker/3:0	Pid: 32305	slice: 50000000	max_wait(ms): 12	total_wait(ms): 18	wait_count: 131	slice sum (ms): 6304
Name: kworker/1:1	Pid: 32658	slice: 50000000	max_wait(ms): 19	total_wait(ms): 37	wait_count: 103	slice sum (ms): 5008
Name: kworker/5:0	Pid: 32674	slice: 50000000	max_wait(ms): 6	total_wait(ms): 13	wait_count: 103	slice sum (ms): 4858
Name: kworker/u64:5	Pid: 33864	slice: 50000000	max_wait(ms): 0	total_wait(ms): 0	wait_count: 1	slice sum (ms): 50
Name: kworker/10:0	Pid: 34195	slice: 50000000	max_wait(ms): 3	total_wait(ms): 13	wait_count: 94	slice sum (ms): 4366
Name: kworker/12:0	Pid: 34367	slice: 25000000	max_wait(ms): 6	total_wait(ms): 11	wait_count: 85	slice sum (ms): 3717
Name: kworker/9:0	Pid: 34813	slice: 50000000	max_wait(ms): 4	total_wait(ms): 12	wait_count: 101	slice sum (ms): 4526
Name: kworker/0:0	Pid: 34961	slice: 50000000	max_wait(ms): 1	total_wait(ms): 12	wait_count: 171	slice sum (ms): 8416
Name: kworker/4:3	Pid: 34987	slice: 50000000	max_wait(ms): 0	total_wait(ms): 2	wait_count: 62	slice sum (ms): 2899
Name: kworker/7:2	Pid: 35283	slice: 50000000	max_wait(ms): 0	total_wait(ms): 6	wait_count: 101	slice sum (ms): 4866
Name: kworker/4:0	Pid: 35548	slice: 25000000	max_wait(ms): 0	total_wait(ms): 0	wait_count: 1	slice sum (ms): 25
Name: kworker/11:0	Pid: 35734	slice: 50000000	max_wait(ms): 3	total_wait(ms): 21	wait_count: 116	slice sum (ms): 5174
----- Name: hello	Pid: 35833	slice: 67	max_wait(ms): 1002	total_wait(ms): 59012	wait_count: 59	
Name: kworker/4:2	Pid: 35917	slice: 50000000	max_wait(ms): 2	total_wait(ms): 10	wait_count: 124	slice sum (ms): 5976
Name: StreamT-ns #272	Pid: 35960	slice: 50000000	max_wait(ms): 0	total_wait(ms): 0	wait_count: 1	slice sum (ms): 50
Name: StreamT-ns #395	Pid: 35964	slice: 50000000	max_wait(ms): 11	total_wait(ms): 11	wait_count: 1	slice sum (ms): 50

Rysunek 4: Wynik programu z zaznaczonym zagładzanym procesem *hello*, czas działania: 60 sekund

## 7 Problemy na jakie natrafiłem

### 7.1 Rozbieżność wersji

W przykładach nazwa funkcji do przełączenia zadań na procesorze nazywa się:

```
scx_bpf_consume(u64 dsq_id)
```

a w wersji jądra 6.14 nazwa została zmieniona na:

```
scx_bpf_dsq_move_to_local(u64 dsq_id)
```

by znaleźć to trzeba zajrzeć do dokumentacji dokładnej wersji i zobaczyć jak zmieniła się nazwa funkcji: <https://github.com/torvalds/linux/blob/v6.14/kernel/sched/ext.c#L6749>. Trzeba mieć na uwadze, że jądro Linuxa jest dynamicznie rozwijane i dokumentacja nie nadąża lub jest może być nieaktualna.

### 7.2 Output eBPF

Oto typowy output błędu przy kompilacji za pomocą eBPF:

```
libbpf: prog 'sched_running': BPF program load failed: -EACCES
libbpf: prog 'sched_running': -- BEGIN PROG LOAD LOG --
0: R1=ctx() R10=fp0
; int BPF_STRUCT_OPS(sched_running, struct task_struct *p) { @ selective.bpf.c:158
0: (79) r7 = *(u64 *) (r1 + 0)
```

```

func 'running' arg0 has btf_id 86 type STRUCT 'task_struct'
1: R1=ctx() R7_w=trusted_ptr_task_struct()
; task_stats_ext *stats = bpf_task_storage_get(&task_storage, p, NULL, @
selective.bpf.c:160
1: (18) r1 = 0xfffff8e975abf8400 ; R1_w=map_ptr(map=task_storage,ks=4,vs=88)
3: (bf) r2 = r7 ; R2_w=trusted_ptr_task_struct()
R7_w=trusted_ptr_task_struct()
4: (b7) r3 = 0 ; R3_w=0
5: (b7) r4 = 1 ; R4_w=1
6: (85) call bpf_task_storage_get#156 ;
R0_w=map_value_or_null(id=1,map=task_storage,ks=4,vs=88)
7: (bf) r6 = r0 ;
R0_w=map_value_or_null(id=1,map=task_storage,ks=4,vs=88)
R6_w=map_value_or_null(id=1,map=task_storag
e,ks=4,vs=88)
; if (!stats) { @ selective.bpf.c:163
8: (15) if r6 == 0x0 goto pc+18 ; R6_w=map_value(map=task_storage,ks=4,vs=88)
9: (b7) r1 = 23 ; R1_w=23
; p->on_cpu = 23; @ selective.bpf.c:166
10: (63) *(u32 *)(r7 +52) = r1
processed 10 insns (limit 1000000) max_states_per_insn 0 total_states 0 peak_states 0
mark_read 0
-- END PROG LOAD LOG --
libbpf: prog 'sched_running': failed to load: -EACCES
libbpf: failed to load object 'selective_bpf'
libbpf: failed to load BPF skeleton 'selective_bpf': -EACCES
Failed to load and verify BPF skeleton

```

Cieężko jest za pierwszym razem zrozumieć jaki jest błąd. Występuje on po poprawnej kompilacji programu, więc nie jest to błąd składniowy. Błąd wynika z niezrozumienia jakiejś własności programów eBPF. Błąd *EACCES* może oznaczać wiele rzeczy, w wielu przypadkach błąd zwracany przez eBPF nie jest przydatny w debugowaniu. Wymaga dogłębnego przeanalizowania co poszło nie tak i dogłębnej wiedzy o tym jak działa eBPF, na co pozwala, a na co nie. eBPF jest ciągle rozwijany i jak korzysta się z najnowszych technologii, to nie wiadomo czy błąd na który się natknąłeś jest twoim brakiem wiedzy, czy bugiem który zostanie dopiero załatany

W tym przypadku jednym z rozwiązań jest zapytania chatbota, jego odpowiedź nie musi być prawidłowa, ale może nakierować nas na dobry trop. Dla powyższego kodu podaje poprawną odpowiedź:

👉 You are not allowed to write to arbitrary kernel memory, including fields of `struct task_struct`.

This is true **even in** `BPF_STRUCT_OPS`, **even in scheduler hooks**, and **even if the kernel would normally allow it in C**.

W kodzie chcemy zmienić fragment pamięci należącą do jądra:

```
p->on_cpu = 23; @ selective.bpf.c:166
```

Na co eBPF nie pozwala. Jest to błąd który pojawia się **w trakcie działania programu**.

Ważne w pracowaniu z eBPF jest częste **kompilowanie** i **uruchamianie** programu. Jeżeli napiszemy 300 linii kodu i dostaniemy nic nie mówiący błąd w trakcie trwania procesu, nie będziemy wiedzieć gdzie szukać błędu. Ważnym jest by jak najszybciej stworzyć coś działającego

i w małych porcjach dodawanie linijek kodu, by w razie błędu liczba linijek do sprawdzania była relatywnie mała.

### 7.3 Pomyłki w przykładach?

W przykładach pokazane jest zrobienie wspólnej kolejki na procesy (*shared\_dispatch\_queue*) z podaną linijką:

```
u64 slice = 5000000u / scx_bpf_dsq_nr_queued(SHARED_DSQ_ID);
scx_bpf_dispatch(p, SHARED_DSQ_ID, slice, enq_flags);
```

*scx\_bpf\_dsq\_nr\_queued*, zwraca liczbę procesów aktualnie znajdujących się w kolejce do procesorów. Problemem jest to, że kolejka może być pusta, co powodowałoby dzielenie przez 0. Dzielenie przez 0 w programach eBPF nie powoduje wyrzucenia błędu, lecz po prostu wynikiem jest 0. Powoduje to, że porcja czasu dostarczona do procesu wynosi 0. Nie oznacza to, że proces nie dostanie czasu procesora, a po prostu zostanie wywłaszczony przy najbliższej możliwej okazji. Ten błąd jednak nie powoduje żadnych problemów, w trakcie trwania planisty, ale jest to raczej nie zamierzane. Danie bardzo małego kawałka czasu jest nie efektywne, ale nie powoduje utraty responsywności komputera, dopóki liczba procesów wymagająca stale czasu procesora nie jest dużo większa od liczby rdzeni procesora.

Znalazłem to przez przypadek realizując mój projekt. Wypisując dane, które przypisuje *scheduler* zauważyłem, że *slice* wynosi 0. nie wydaje mi się by miał dla tej wartości specjalne własności. Trzeba wiedzieć, że dokumentacja, czy przykłady mogą mieć błędy, lub być nie optymalne.

### 7.4 Skąd wziąć nazwy funkcji?

Jest dokumentacja dotycząca funkcji eBPF od tym linkiem: [https://docs.ebpf.io/linux/program-type/BPF\\_PROG\\_TYPE\\_STRUCT\\_OPS/sched\\_ext\\_ops/](https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_STRUCT_OPS/sched_ext_ops/). Nie jest zbiór funkcji jądra, tyle funkcje do komunikacji z eBPF; nie ma tutaj najważniejszych funkcji, które faktycznie umożliwiają komunikację z jądrem w celu przydzielenia czasu procesora. By się tego dowiedzieć trzeba zobaczyć faktyczną implementację: <https://github.com/torvalds/linux/blob/v6.14/kernel/sched/ext.c>. Funkcje są tam dobrze opisane i faktycznie tłumaczą jakie mają zastosowanie.

## 8 Wnioski

Pisanie programów z pomocą eBPF jest zalecaną drogą dodawania funkcjonalności do jądra systemu Linux. Nie jest to narzędzie w pełni uniwersalne i ma swoje ograniczenia, ale z roku na rok dodawanych jest coraz więcej funkcjonalności.

Problem jest z dokumentacją, często jest ona wybrakowana lub pewne rzeczy zostały zmienione pomiędzy poszczególnymi wersjami jądra, i nie jest do udokumentowane. Wyjście w razie błędu wygenerowane przez eBPF często jest nie zrozumiałe i mało pomocne, więc trzeba cofać się do najbliższej działającej wersji by nabrać pojęcia gdzie leży błąd.

Gdyby nie dedykowane rozszerzenie *sched\_ext* napisanie tego w eBPF byłoby nie możliwe, i musiałbym ręcznie dodać planistę do jądra i za każdym razem kompilować całe jądro. Ten proces byłby o wiele dłuższy i nużący. Jakikolwiek błąd kończył, by się zawieszeniem pracy komputera, albo tzw. *kernel panic!*.

Planista działa tak jak powinien, ciekawe było do obserwacji zachowanie systemu pod wpływem samego napisanego planisty. Spędziłem dużo czasu na próbach wyjaśnienia czemu działa tak, a nie inaczej (na przykład zachowania dla bardzo dużej porcji czasu).

## 9 Literatura i źródła

- Kod źródłowy: [https://github.com/Satan-Vicovaro/BPF\\_Scheduler](https://github.com/Satan-Vicovaro/BPF_Scheduler)
- <https://cilium.isovalent.com/hubfs/Learning-eBPF%20-%20Full%20book.pdf>
- <https://github.com/torvalds/linux/blob/v6.14/kernel/sched/ext.h>
- <https://github.com/parttimenerd/minimal-scheduler?tab=readme-ov-file>
- [https://docs.ebpf.io/linux/program-type/BPF\\_PROG\\_TYPE\\_STRUCT\\_OPS/sched\\_ext\\_ops/](https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_STRUCT_OPS/sched_ext_ops/)