可拓展调度器类 sched_ext 技术

施鹏飞

中国科学院软件研究所

智能软件研究中心

2023 年 5 月 16 日

第 I 部分

背景简介

诞生背景

- 依托于 eBPF

诞生背景

- 依托于 eBPF
- BPF 背后的核心思想是它允许程序在运行时从用户空间加载到内核; 使用 BPF 进行调度有可能实现与现在在 Linux 系统中看到的截然不同的调度行为.

诞生背景

“可插拔”调度器的想法并不新鲜；例如，在 2004 年关于另一个来自 Con Kolivas 的补丁系列的讨论中，可插拔调度器的想法遭到强烈反对；有人认为，只有将精力集中在单个调度程序上，开发社区才能找到满足所有工作负载的方法，而不会使内核充满专用调度程序的混乱.

此外，应用程序可以在内核的 CFS、RT、DL 调度程序中进行选择，这确实在管理从嵌入式系统到超级计算机的各种工作负载方面确实做得很好. 人们总是希望获得更好的性能，但多年来几乎没有人提出对可插拔调度器机制的要求.

诞生背景

那么，为什么现在要提出可插拔调度器机制呢？补丁系列的封面信详细描述了这项工作背后的动机. 简而言之，该论点认为: 在 BPF 中编写调度策略的能力大大降低了试验新调度方法的难度.

由于引入了完全公平调度器 CFS，我们的工作负载及其运行的系统都变得更加复杂. 人们需要通过实验来开发适合当前系统的调度算法. 基于 BPF 的调度类则可以采用安全的方式进行实验，甚至不需要重新启动测试机. BPF 编写的调度器还可以提高小众的工作场景的性能，毕竟这些工作场景可能不值得在 mainline kernel 中实现相关支持.

eBPF 整体结构图如下：

- 用户空间程序负责加载 BPF 字节码至内核，如需要也会负责读取内核回传的统计信息或者事件详情；
- 内核中的 BPF 字节码负责在内核中执行特定事件，如需要也会将执行的结果通过 maps 或者 perf-event 事件发送至用户空间；

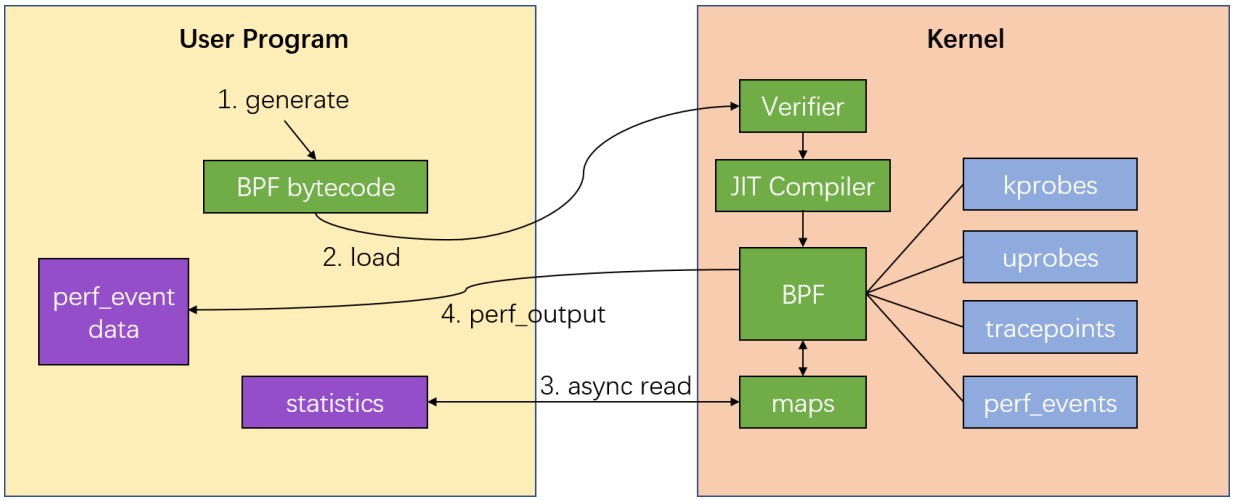


图 1: eBPF 程序架构图[1]

[1] https://cloudnative.to/blog/bpf-intro/#22-ebpf-%E6%9E%B6%E6%9E%84%E8%A7%82%E6%B5%8B

内核调度类

系统中可能运行着各种类型的进程，用户对不同种类进程的期望值是不一样的，例如对于实时进程，我们希望进程具备超高的响应速度，进程一旦就绪就立马被调度到；而对于批处理进程，用户更多关注的是其吞吐量，只要他能够在后台默默运行完成即可．因此对于调度器而言，不同的进程类型意味着不同的调度逻辑．

内核调度类

Linux 在 2.6 版本中引入了 “调度类 (Sched Class)” 的概念，意在将调度逻辑模块化，内核通过 `struct sched_class`抽象出了调度类的通用行为:

```
/* file: kernel/sched/sched.h */
struct sched_class {
        void (*enqueue_task)(struct rq *rq, struct task_struct *p, int flags);
        void (*dequeue_task)(struct rq *rq, struct task_struct *p, int flags);
        void (*yield_task)(struct rq *rq);
        bool (*yield_to_task)(struct rq *rq, struct task_struct *p);
        struct task_struct *(*pick_next_task)(struct rq *rq);
        void (*put_prev_task)(struct rq *rq, struct task_struct *p);
        void (*set_next_task)(struct rq *rq, struct task_struct *p, bool first);
        int (*balance)(struct rq *rq, struct task_struct *prev, struct rq_flags *rf);
        ...
};
```

内核调度类

调度器按照顺序对具体的调度类进行遍历，依次调用每个调度类的 `pick_next_task`方法从指定 rq 中寻找下一个可执行任务，如果返回非空，则将该任务返回.

内核调度类

顺序遍历实际上意味着调度类之间存在优先级关系,Linux 总共实现了 5 种调度类，按照优先级有高到底排序依次为:

1. `stop_sched_class` Stop 是特殊的调度类，内核使用该调度类来停止 CPU. 该调度类用来强行停止 CPU 上的其他任务，由于该调度类的优先级最高，因此一旦生效就将抢占任何当前正在运行的任务，并且在运行过程中自己不会被抢占。该调度类只有在 SMP 架构的系统中存在，内核使用该调度类来完成负载均衡与 CPU 热插拔等工作。

内核调度类

2. `dl_sched_class` 有些任务必须在指定时间窗口内完成。例如视频的编码与解码，CPU 必须以特定频率完成对应的数据处理；这类任务是优先级最高的用户任务，CPU 应该首先满足。Deadline 调度类用来调度这类任务，dl 便是单词 Deadline 的缩写，因此该调度类的优先级仅仅低于 Stop 调度类。

内核调度类

3. `rt_sched_class` 实时任务 (Real-time Task) 对响应时间要求更高，例如编辑器软件，它可能由于等待用户输入长期处于睡眠之中，但一旦用户有输入动作，我们就期望编辑器能够立马响应，而不是等系统完成其它任务之后才开始反应，这一点对用户体验十分重要. RT 调度类用来调度这类任务，该调度类的优先级低于 DL.

内核调度类

4. `fair_sched_class` Fair 调度类用来调度绝大多数用户任务，CFS 实现的就是这种调度类，其核心逻辑是根据任务的优先级公平地分配 CPU 时间.

内核调度类

5. `idle_sched_class` 与 Stop 类似，Idle 调度类也是仅供内核使用的特殊调度类，其优先级最低，只有在没有任何用户任务时才会用到。内核会为每个 CPU 绑定一个内核线程（kthread）来完成该任务，该线程会在队列无事可做的情况下启动该任务，并将 CPU 的功耗降到最低。

第 II 部分

从一个例子开始

sched_ext 代码结构

总的来说，SCX 机制涉及的代码文件不多，主要在 3 个地方：

- `include/linux/sched/ext.h`

  内核与用户的接口文件，定义了绝大多数的函数接口.

- `kernel/sched/ext.c`

  内核拓展调度器核心代码.

- `tools/sched_ext/` 通过实现的 SCX 调度器的例子.
  - `scx_example_dummy.c` 用户态的 BPF 装载器 (libbpf)
  - `scx_example_dummy.bpf.c` BPF 程序
  - ...

sched_ext 工作流程

根据目前现有的内核文档 (./Documentation/scheduler/sched-ext.rst), 尝试说明一些 scx 机制的工作流程.

dispatch queue 分发队列

在 sched_ext 架构中，有一个 dispatch queues(分发队列，dsq's) 的概念，提出 dsq 的原因是由于用户自定义的 BPF 调度器和核心调度器之间往往可能存在着不协调，而 dsq 本质上是一个简单的 FIFO 队列结构.

dispatch queue 分发队列

- 目前的实现中，默认存在一个全局 dsq(SCX_DSQ_GLOBAL)，以及一个每 CPU 的局部 dsq(SCX_DSQ_LOCAL).

dispatch queue 分发队列

- 目前的实现中，默认存在一个全局 dsq(`SCX_DSQ_GLOBAL`)，以及一个每 CPU 的局部 dsq(`SCX_DSQ_LOCAL`).
- 不过 BPF 调度器可以通过`scx_bpf_create_dsq`以及`scx_bpf_destroy_dsq`两个内核函数，自己定义任意数量的 dsq 结构.

dispatch queue 分发队列

- 目前的实现中，默认存在一个全局 dsq(`SCX_DSQ_GLOBAL`)，以及一个每 CPU 的局部 dsq(`SCX_DSQ_LOCAL`).
- 不过 BPF 调度器可以通过`scx_bpf_create_dsq`以及`scx_bpf_destroy_dsq`两个内核函数，自己定义任意数量的 dsq 结构.
- 一个任务总是会被分发 (dispatch) 到一个 dsq 中.

dispatch queue 分发队列

- 目前的实现中，默认存在一个全局 dsq(`SCX_DSQ_GLOBAL`)，以及一个每 CPU 的局部 dsq(`SCX_DSQ_LOCAL`).
- 不过 BPF 调度器可以通过`scx_bpf_create_dsq`以及`scx_bpf_destroy_dsq`两个内核函数，自己定义任意数量的 dsq 结构.
- 一个任务总是会被分发 (dispatch) 到一个 dsq 中.
- 一个任务的执行则必须依靠 CPU 主动去从 dsq 去消耗 (consume) 任务.

dispatch queue 分发队列

- 目前的实现中，默认存在一个全局 dsq(`SCX_DSQ_GLOBAL`)，以及一个每 CPU 的局部 dsq(`SCX_DSQ_LOCAL`).
- 不过 BPF 调度器可以通过`scx_bpf_create_dsq`以及`scx_bpf_destroy_dsq`两个内核函数，自己定义任意数量的 dsq 结构.
- 一个任务总是会被分发 (dispatch) 到一个 dsq 中.
- 一个任务的执行则必须依靠 CPU 主动去从 dsq 去消耗 (consume) 任务.
  - 在内部实现中，一个 CPU 会且仅会执行其局部 dsq 上的任务，因此`.consume()`方法实际上是将其他 dsq 上的任务移动到局部 dsq 上. 这也意味着仅有当 CPU 的局部 dsq 上为空时，才会去调用该方法夺取其余 dsq 上的任务.

调度周期

下面说明一个任务在 SCX 架构下是如何被调度 & 执行的:

- 当一个 task 被唤醒，首先执行的是`select_cpu`方法，其用于选择最优的 CPU，同时如果该 CPU 闲置则唤醒该 CPU.

调度周期

下面说明一个任务在 SCX 架构下是如何被调度 & 执行的:

- 当一个 task 被唤醒, 首先执行的是`select_cpu`方法, 其用于选择最优的 CPU, 同时如果该 CPU 闲置则唤醒该 CPU.
- 当选中一个 CPU 后, 将调用`enqueue`方法, 其将使用`scx_bpf_dispatch`将任务分发到全局/局部/用户自定义的 dsq 中.

调度周期

下面说明一个任务在 SCX 架构下是如何被调度 & 执行的:

- 当一个 task 被唤醒，首先执行的是`select_cpu`方法，其用于选择最优的 CPU，同时如果该 CPU 闲置则唤醒该 CPU.
- 当选中一个 CPU 后，将调用`enqueue`方法，其将使用`scx_bpf_dispatch`将任务分发到全局/局部/用户自定义的 dsq 中.
- 一旦一个 CPU 准备好进行调度了，其将首先查看自己的本地 dsq. 如果其为空，则将调用`consume`方法，进而调用若干次内核函数`scx_bpf_consume`来从其他 dsq 中取任务. 一旦函数返回成功则 CPU 将获得下一个用于执行的任务.

调度周期

下面说明一个任务在 SCX 架构下是如何被调度 & 执行的:

- 当一个 task 被唤醒，首先执行的是`select_cpu`方法，其用于选择最优的 CPU，同时如果该 CPU 闲置则唤醒该 CPU.
- 当选中一个 CPU 后，将调用`enqueue`方法，其将使用 `scx_bpf_dispatch`将任务分发到全局/局部/用户自定义的 dsq 中.
- 一旦一个 CPU 准备好进行调度了，其将首先查看自己的本地 dsq. 如果其为空，则将调用`consume`方法，进而调用若干次内核函数 `scx_bpf_consume`来从其他 dsq 中取任务. 一旦函数返回成功则 CPU 将获得下一个用于执行的任务.
- 如果仍然没有任务可用于执行，将调用`dispatch`方法，进而调用若干次`scx_bpf_dispatch`函数来要求 BPF 调度器对任务进行分发. 如果有任务被成功分发了，则回到上一步进行处理.

调度周期

下面说明一个任务在 SCX 架构下是如何被调度 & 执行的:

- 当一个 task 被唤醒，首先执行的是`select_cpu`方法，其用于选择最优的 CPU，同时如果该 CPU 闲置则唤醒该 CPU.
- 当选中一个 CPU 后，将调用`enqueue`方法，其将使用`scx_bpf_dispatch`将任务分发到全局/局部/用户自定义的 dsq 中.
- 一旦一个 CPU 准备好进行调度了，其将首先查看自己的本地 dsq. 如果其为空，则将调用`consume`方法，进而调用若干次内核函数`scx_bpf_consume`来从其他 dsq 中取任务. 一旦函数返回成功则 CPU 将获得下一个用于执行的任务.
- 如果仍然没有任务可用于执行，将调用`dispatch`方法，进而调用若干次`scx_bpf_dispatch`函数来要求 BPF 调度器对任务进行分发. 如果有任务被成功分发了，则回到上一步进行处理.
- 如果仍然没有任务可用于执行，则则调用`consume_final` 方法. 该方法与`consume`等价，但是其在 CPU 即将闲置前才被进行调用. 这为调度器提供了一个 hook，可用于执行自定义的一些操作.

BPF 程序结构

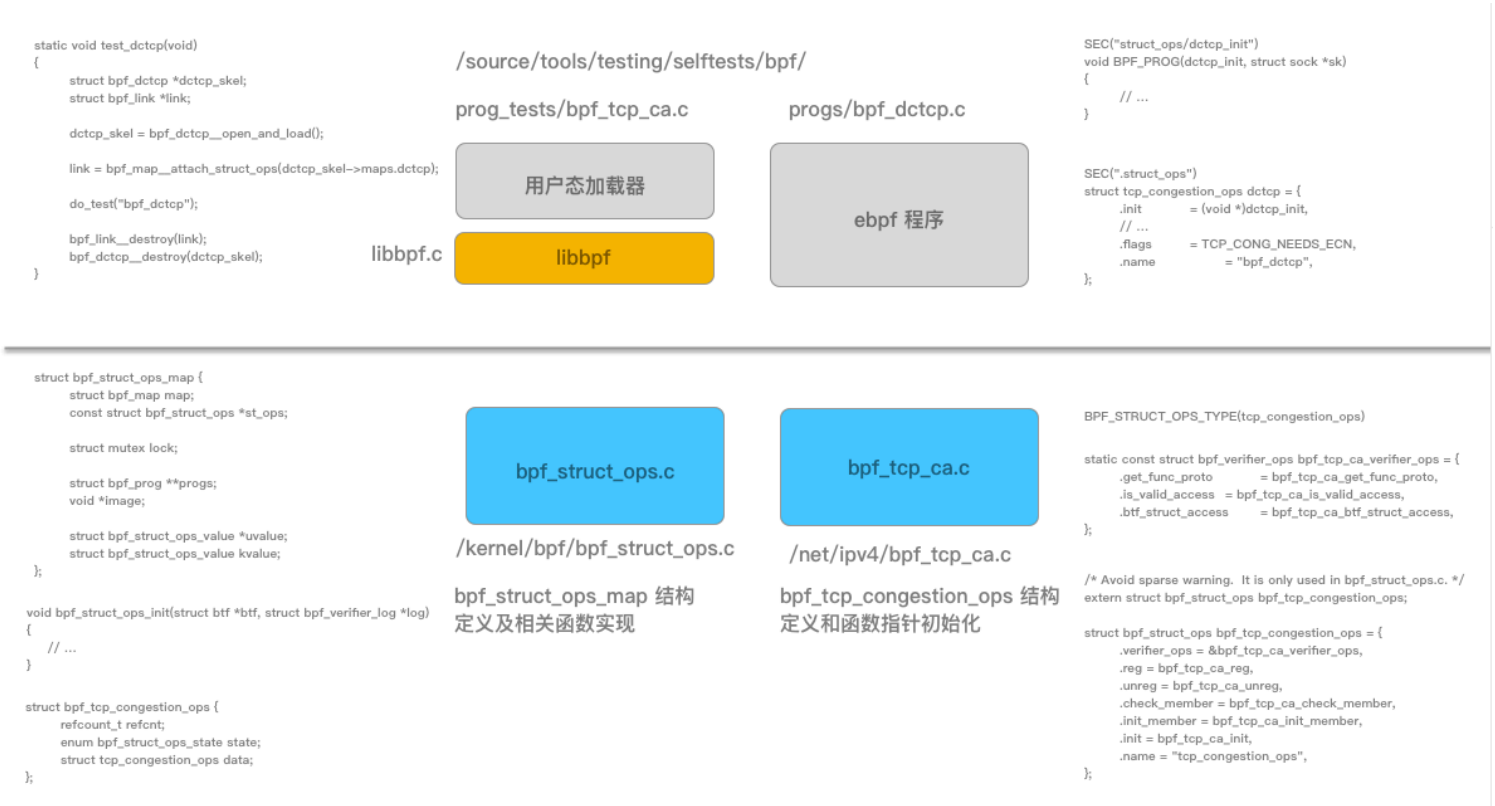


图 2: TCP 拥塞控制算法 +BPF [2]

[2] https://www.ebpf.top/post/ebpf_struct_ops/#3-%E8%84%9A%E6%89%8B%E6%9E%B6%E4%BB%A3%E7%A0%81%E7%9B%B8%E5%85%B3%E5%AE%9E%E7%8E%B0

一个简单的例子

以一个最小化的 sched_ext 调度器为例.

在调度方面, 其表现的像一个全局的 FIFO 一样. 同时其也会利用 BPF Map 记录一些统计信息, 例如已经调度的进程数等.

BPF 程序

./tools/sched_ext/scx_example_dummy.bpf.c

```
#include "scx_common.bpf.h"

char _license[] SEC("license") = "GPL";

const volatile bool switch_all;

struct user_exit_info uei;

struct {
        __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
        __uint(key_size, sizeof(u32));
        __uint(value_size, sizeof(u64));
        __uint(max_entries, 2);  /* [local, global]
        */
} stats SEC(".maps");

static void stat_inc(u32 idx)
{
        u64 *cnt_p = bpf_map_lookup_elem(&stats, &
        idx);
        if (cnt_p)
                (*cnt_p)++;
}

s32 BPF_STRUCT_OPS(dummy_init)
{
        if (switch_all)
                scx_bpf_switch_all();
        return 0;
}
```

```
void BPF_STRUCT_OPS(dummy_enqueue, struct
        task_struct *p, u64 enq_flags)
{
        if (enq_flags & SCX_ENQ_LOCAL) {
                stat_inc(0);
                scx_bpf_dispatch(p, SCX_DSQ_LOCAL,
        SCX_SLICE_DFL, enq_flags);
        } else {
                stat_inc(1);
                scx_bpf_dispatch(p, SCX_DSQ_GLOBAL,
        SCX_SLICE_DFL, enq_flags);
        }
}
void BPF_STRUCT_OPS(dummy_exit, struct scx_exit_info
         *ei) {
        uei_record(&uei, ei);
}

SEC(".struct_ops")
struct sched_ext_ops dummy_ops = {
        .enqueue        = (void *)dummy_enqueue,
        .init           = (void *)dummy_init,
        .exit           = (void *)dummy_exit,
        .name           = "dummy",
};
```

BPF 装载器

./tools/sched_ext/scx_example_dummy.c

```c
int main(int argc, char **argv)
{
        struct scx_example_dummy *skel;
        struct bpf_link *link;
        u32 opt;

        signal(SIGINT, sigint_handler);
        signal(SIGTERM, sigint_handler);

        libbpf_set_strict_mode(LIBBPF_STRICT_ALL);

        skel = scx_example_dummy__open();
        assert(skel);

        while ((opt = getopt(argc, argv, "ah")) !=
       -1) {
                switch (opt) {
                case 'a':
                        skel->rodata->switch_all =
       true;
                        break;
                default:
                        fprintf(stderr, help_fmt,
       basename(argv[0]));
                        return opt != 'h';
                }
        }

        assert(!scx_example_dummy__load(skel));

        link = bpf_map__attach_struct_ops(skel->maps
       .dummy_ops);
        assert(link);

        while (!exit_req && !uei_exited(&skel->bss->
       uei)) {
                u64 stats[2];

                read_stats(skel, stats);
                printf("local=%lu global=%lu\n",
       stats[0], stats[1]);
                fflush(stdout);
                sleep(1);
        }

        bpf_link__destroy(link);
        uei_print(&skel->bss->uei);
        scx_example_dummy__destroy(skel);
        return 0;
}
```

BPF Map


```
struct {
        __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
        __uint(key_size, sizeof(u32));
        __uint(value_size, sizeof(u64));
        __uint(max_entries, 2);  /* [local, global] */
} stats SEC(".maps");
```

这段代码定义了一个 BPF_MAP_TYPE_PERCPU_ARRAY 类型的 map，用于存储一组 key-value 对：

- key 的大小为 sizeof(u32)
- value 的大小为 sizeof(u64)
- 最大 entry 数量为 2，分别为 local 和 global

BPF Map

- 关于 BPF Map?

BPF Map

- 关于 BPF Map?
- BPF Map 是一种数据结构，用于在 BPF 程序的不同部分之间共享数据. BPF Map 可以看作是 Linux 内核中的一个键值对数据库，它提供了高效的、线程安全的键值存储，可以被多个 BPF 程序同时访问.

BPF Map

- 关于 BPF Map?
- BPF Map 是一种数据结构，用于在 BPF 程序的不同部分之间共享数据. BPF Map 可以看作是 Linux 内核中的一个键值对数据库，它提供了高效的、线程安全的键值存储，可以被多个 BPF 程序同时访问.
- 要使用 BPF Map，需要在 BPF 程序中定义 Map，指定 Map 的类型、键类型和值类型，并根据需要设置其他选项. 然后，在 BPF 程序的各个部分中，可以使用 Map 来共享数据.

节 (Section)


```
struct {
        __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
        __uint(key_size, sizeof(u32));
        __uint(value_size, sizeof(u64));
        __uint(max_entries, 2);  /* [local, global] */
} stats SEC(".maps");
```

还是以 BPF Map 的定义为例, 其中 `SEC(".maps")` 表示这个 BPF Map 将被放置在.maps 节 (section) 中. 然后, 在 BPF 程序中, 即可使用这个 Map 来共享数据.

节 (Section)

- 关于节 (Section)?

节 (Section)

- 关于节 (Section)?
- 在 BPF 中, “节”(section) 是一种将代码和数据组织在一起的机制. 每个 BPF 程序都由多个节组成, 每个节都有自己的名称、类型和属性.

节 (Section)

- 关于节 (Section)?
- 在 BPF 中, “节”(section) 是一种将代码和数据组织在一起的机制. 每个 BPF 程序都由多个节组成, 每个节都有自己的名称、类型和属性.
- 以下是几种常见的 BPF 节类型:

节 (Section)

- 关于节 (Section)?
- 在 BPF 中, “节”(section) 是一种将代码和数据组织在一起的机制. 每个 BPF 程序都由多个节组成, 每个节都有自己的名称、类型和属性.
- 以下是几种常见的 BPF 节类型:
  - .text: 包含可执行代码的节. 用于存放 BPF 程序的主体代码.

节 (Section)

- 关于节 (Section)?
- 在 BPF 中, “节”(section) 是一种将代码和数据组织在一起的机制. 每个 BPF 程序都由多个节组成, 每个节都有自己的名称、类型和属性.
- 以下是几种常见的 BPF 节类型:
  - .text: 包含可执行代码的节. 用于存放 BPF 程序的主体代码.
  - .maps: 包含 BPF Map 定义的节.

节 (Section)

- 关于节 (Section)?
- 在 BPF 中, “节”(section) 是一种将代码和数据组织在一起的机制. 每个 BPF 程序都由多个节组成, 每个节都有自己的名称、类型和属性.
- 以下是几种常见的 BPF 节类型:
  - .text: 包含可执行代码的节. 用于存放 BPF 程序的主体代码.
  - .maps: 包含 BPF Map 定义的节.
  - .rodata: 包含只读数据的节, 常量和字符串可以放在.rodata 节中.

节 (Section)

- 关于节 (Section)?
- 在 BPF 中, “节”(section) 是一种将代码和数据组织在一起的机制. 每个 BPF 程序都由多个节组成, 每个节都有自己的名称、类型和属性.
- 以下是几种常见的 BPF 节类型:
  - .text: 包含可执行代码的节. 用于存放 BPF 程序的主体代码.
  - .maps: 包含 BPF Map 定义的节.
  - .rodata: 包含只读数据的节, 常量和字符串可以放在.rodata 节中.
  - .data: 包含可读写数据的节, 全局变量可以放在.data 节中.

节 (Section)

- 关于节 (Section)?
- 在 BPF 中, “节”(section) 是一种将代码和数据组织在一起的机制. 每个 BPF 程序都由多个节组成, 每个节都有自己的名称、类型和属性.
- 以下是几种常见的 BPF 节类型:
  - .text: 包含可执行代码的节. 用于存放 BPF 程序的主体代码.
  - .maps: 包含 BPF Map 定义的节.
  - .rodata: 包含只读数据的节, 常量和字符串可以放在.rodata 节中.
  - .data: 包含可读写数据的节, 全局变量可以放在.data 节中.
- 用户也可以自定义节, 比如在 sched_ext 技术中就使用了名为 `.struct_ops` 的节来存放自定义的数据结构.

节 (Section)

- 内核如何访问 BPF 定义的节?

节 (Section)

- 内核如何访问 BPF 定义的节?
- Linux 内核中对 BPF 程序和 Map 的访问是通过文件系统接口进行的，内核会将 BPF 程序和 Map 挂载到/sys/fs/bpf 目录下，并以文件的形式暴露出来. 这样，用户空间的应用程序就可以通过打开相应的文件并进行读写操作，与内核中的 BPF 程序和 Map 进行交互.

节 (Section)

- 内核如何访问 BPF 定义的节?
- Linux 内核中对 BPF 程序和 Map 的访问是通过文件系统接口进行的，内核会将 BPF 程序和 Map 挂载到/sys/fs/bpf 目录下，并以文件的形式暴露出来. 这样，用户空间的应用程序就可以通过打开相应的文件并进行读写操作，与内核中的 BPF 程序和 Map 进行交互.
- 通常 BPF 程序和 Map 只能由特权用户创建和加载. 因此，在实际使用中，通常需要使用特权用户身份运行应用程序，以便访问 BPF 程序和 Map.

节 (Section)

- 如何定义节?

节 (Section)

- 如何定义节?
- 上述出现的SEC(...) 其实是一个 C 语言宏, 其又进一步根据 gcc 或 clang 拓展为不同的编译器标记.

```
#if __GNUC__ && !__clang__
#define SEC(name) __attribute__((section(name), used))
#else
#define SEC(name) \
        _Pragma("GCC diagnostic push")                                  \
        _Pragma("GCC diagnostic ignored \"-Wignored-attributes\"")      \
        __attribute__((section(name), used))                            \
        _Pragma("GCC diagnostic pop")                                   \
#endif
```

struct_ops 结构体

在 sched_ext 的设计架构中，任意一个对结构体 `struct sched_ext_ops` 的实现都可以被载入内核作为调度器. 该结构体位于 include/linux/sched/ext.h 中.

struct_ops 结构体


```
struct sched_ext_ops {
        s32 (*select_cpu)(struct task_struct *p, s32 prev_cpu, u64 wake_flags);
        void (*enqueue)(struct task_struct *p, u64 enq_flags);
        void (*dequeue)(struct task_struct *p, u64 deq_flags);
        void (*dispatch)(s32 cpu, struct task_struct *prev);
        void (*consume)(s32 cpu);
        void (*runnable)(struct task_struct *p, u64 enq_flags);
        void (*running)(struct task_struct *p);
        void (*stopping)(struct task_struct *p, bool runnable);
        void (*quiescent)(struct task_struct *p, u64 deq_flags);
        bool (*yield)(struct task_struct *from, struct task_struct *to);
        char name[SCX_OPS_NAME_LEN];
        ...
};
```

对于 scx 机制而言，唯一必须的字段只有.name 字段，并要求是一个合法的 BPF 对象名称，而其余所有的 operation 均是可选的.

函数定义

- 具体到本例子中，dummy_example 实现了其中的 3 个函数：

```
s32 BPF_STRUCT_OPS(dummy_init)
void BPF_STRUCT_OPS(dummy_enqueue, struct task_struct *p, u64 enq_flags)
void BPF_STRUCT_OPS(dummy_exit, struct scx_exit_info *ei)
```

函数定义

- 具体到本例子中，dummy_example 实现了其中的 3 个函数:

```
s32 BPF_STRUCT_OPS(dummy_init)
void BPF_STRUCT_OPS(dummy_enqueue, struct task_struct *p, u64 enq_flags)
void BPF_STRUCT_OPS(dummy_exit, struct scx_exit_info *ei)
```

- dummy_ops 结构体:

```
SEC(".struct_ops")
struct sched_ext_ops dummy_ops = {
        .enqueue                = (void *)dummy_enqueue,
        .init                   = (void *)dummy_init,
        .exit                   = (void *)dummy_exit,
        .name                   = "dummy",
};
```

函数定义

- BPF_STRUCT_OPS 其实是这样一个宏:

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

函数定义

- BPF_STRUCT_OPS 其实是这样一个宏:

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

- BPF_PROG 又实质地声明与定义了该 BPF 函数

```
#define BPF_PROG(name, args...)                                          \
name(unsigned long long *ctx);                                           \
static __always_inline typeof(name(0))                                   \
____##name(unsigned long long *ctx, ##args);                             \
typeof(name(0)) name(unsigned long long *ctx)                            \
{                                                                        \
        _Pragma("GCC diagnostic push")                                   \
        _Pragma("GCC diagnostic ignored \"-Wint-conversion\"")           \
        return ____##name(___bpf_ctx_cast(args));                        \
        _Pragma("GCC diagnostic pop")                                    \
}                                                                        \
static __always_inline typeof(name(0))                                   \
____##name(unsigned long long *ctx, ##args)
```

dummy_init

BPF 程序关于 dummy_init 的定义如下很简单，根据装载器程序中是否传入了-a 参数来决定是否要开启调度.

```
// scx_example_dummy.bpf.c
const volatile bool switch_all;
s32 BPF_STRUCT_OPS(dummy_init)
{
        if (switch_all)
                scx_bpf_switch_all();
        return 0;
}
```

相应的可以看到，在装载器程序中使用了 BPF 的rodate节来共享变量.

```
// scx_example_dummy.c
while ((opt = getopt(argc, argv, "ah")) != -1) {
        switch (opt) {
        case 'a':
                skel->rodata->switch_all = true;
                break;
        default:
                fprintf(stderr, help_fmt, basename(argv[0]));
                return opt != 'h';
        }
}
```

dummy_enqueue

- 根据 enqueue 标志来决定该调度任务将被分配到全局队列还是每 CPU 队列中，同时调用stat_inc在 bpf Map 中记录一些统计数据.

```
void BPF_STRUCT_OPS(dummy_enqueue, struct task_struct *p, u64 enq_flags)
{
        if (enq_flags & SCX_ENQ_LOCAL) {
                stat_inc(0);
                scx_bpf_dispatch(p, SCX_DSQ_LOCAL, SCX_SLICE_DFL, enq_flags);
        } else {
                stat_inc(1);
                scx_bpf_dispatch(p, SCX_DSQ_GLOBAL, SCX_SLICE_DFL, enq_flags);
        }
}
```

dummy_enqueue

- 根据 enqueue 标志来决定该调度任务将被分配到全局队列还是每 CPU 队列中，同时调用stat_inc在 bpf Map 中记录一些统计数据.

```
void BPF_STRUCT_OPS(dummy_enqueue, struct task_struct *p, u64 enq_flags)
{
        if (enq_flags & SCX_ENQ_LOCAL) {
                stat_inc(0);
                scx_bpf_dispatch(p, SCX_DSQ_LOCAL, SCX_SLICE_DFL, enq_flags);
        } else {
                stat_inc(1);
                scx_bpf_dispatch(p, SCX_DSQ_GLOBAL, SCX_SLICE_DFL, enq_flags);
        }
}
```

- 读取 BPF Map 结构体，更新调度数据.

```
static void stat_inc(u32 idx)
{
        u64 *cnt_p = bpf_map_lookup_elem(&stats, &idx);
        if (cnt_p)
                (*cnt_p)++;
}
```

dummy_enqueue

- 根据 enqueue 标志来决定该调度任务将被分配到全局队列还是每 CPU 队列中，同时调用stat_inc在 bpf Map 中记录一些统计数据.

```
void BPF_STRUCT_OPS(dummy_enqueue, struct task_struct *p, u64 enq_flags)
{
        if (enq_flags & SCX_ENQ_LOCAL) {
                stat_inc(0);
                scx_bpf_dispatch(p, SCX_DSQ_LOCAL, SCX_SLICE_DFL, enq_flags);
        } else {
                stat_inc(1);
                scx_bpf_dispatch(p, SCX_DSQ_GLOBAL, SCX_SLICE_DFL, enq_flags);
        }
}
```

- 读取 BPF Map 结构体，更新调度数据.

```
static void stat_inc(u32 idx)
{
        u64 *cnt_p = bpf_map_lookup_elem(&stats, &idx);
        if (cnt_p)
                (*cnt_p)++;
}
```

- scx_bpf_dispatch是 scx 机制实现的内核函数，用于将任务实际分发给每 CPU/全局的调度队列中. 有关这类调度函数将在后续章节进行源码级分析.

dummy_exit

记录 exit_info, 通过 BPF 在内核和用户空间中共享该信息.

```
void BPF_STRUCT_OPS(dummy_exit, struct scx_exit_info *ei)
{
        uei_record(&uei, ei);
}
```

调度器中的相应使用为:

```
while (!exit_req && !uei_exited(&skel->bss->uei)) {
        u64 stats[2];
        read_stats(skel, stats);
        printf("local=%lu global=%lu\n", stats[0], stats[1]);
        fflush(stdout);
        sleep(1);
}
```

BPF 装载与内核支持

当在 BPF 程序中实现了 struct_ops 结构，并成功通过SEC将其装入了 .struct_ops节，内核如何实际地能够访问到用户实现的这些函数呢？

BPF 用户态装载器

- 回到装载器中可以看到几个实际装载 BPF 程序的函数功能:

```
skel = scx_example_dummy__open();
assert(!scx_example_dummy__load(skel));
link = bpf_map__attach_struct_ops(skel->maps.dummy_ops);
bpf_link__destroy(link);
scx_example_dummy__destroy(skel);
```

[3] https://stackoverflow.com/questions/68278120/ebpf-difference-between-loading-attaching-and-linking

BPF 用户态装载器

- 回到装载器中可以看到几个实际装载 BPF 程序的函数功能:

```
skel = scx_example_dummy__open();
assert(!scx_example_dummy__load(skel));
link = bpf_map__attach_struct_ops(skel->maps.dummy_ops);
bpf_link__destroy(link);
scx_example_dummy__destroy(skel);
```

- 加载 (load)? 使用 libbpf 库将 BPF 程序对象编译成字节码, 通过 `bpf(BPF_PROG_LOAD, ...)`系统调用将 BPF 程序指令注入内核; 附加 (attach)? 这个函数会将先前加载的 BPF 程序附加到一个 BPF 对象上. BPF 对象可以是各种类型, 例如网络接口、套接字和映射等. 附加 BPF 程序后, 内核将开始在 BPF 对象上执行 BPF 程序. [3]

[3] https://stackoverflow.com/questions/68278120/ebpf-difference-between-loading-attaching-and-linking

BPF 用户态装载器

- 回到装载器中可以看到几个实际装载 BPF 程序的函数功能:

```
skel = scx_example_dummy__open();
assert(!scx_example_dummy__load(skel));
link = bpf_map__attach_struct_ops(skel->maps.dummy_ops);
bpf_link__destroy(link);
scx_example_dummy__destroy(skel);
```

- 加载 (load)? 使用 libbpf 库将 BPF 程序对象编译成字节码, 通过 `bpf(BPF_PROG_LOAD, ...)`系统调用将 BPF 程序指令注入内核; 附加 (attach)? 这个函数会将先前加载的 BPF 程序附加到一个 BPF 对象上. BPF 对象可以是各种类型, 例如网络接口、套接字和映射等. 附加 BPF 程序后, 内核将开始在 BPF 对象上执行 BPF 程序. [3]
- BPF 程序必须经过严格的安全检查才能被加载和执行. 在加载 BPF 程序之前, 内核会验证程序的指令是否安全, 以确保它们不会破坏系统的稳定性或安全性.

[3] https://stackoverflow.com/questions/68278120/ebpf-difference-between-loading-attaching-and-linking

内核支持

- 而为了实现针对`struct_ops`的装载功能，内核也必须要提供相应的支持框架. 这其中又分为两部分:

内核支持

- 而为了实现针对struct_ops的装载功能，内核也必须要提供相应的支持框架. 这其中又分为两部分:
  - BPF_STRUCT_OPS_TYPE宏.

内核支持

- 而为了实现针对struct_ops的装载功能，内核也必须要提供相应的支持框架. 这其中又分为两部分:
  - BPF_STRUCT_OPS_TYPE宏.
  - `struct` bpf_struct_ops具体实现.

内核支持

- 而为了实现针对struct_ops的装载功能，内核也必须要提供相应的支持框架. 这其中又分为两部分:
  - BPF_STRUCT_OPS_TYPE宏.
  - `struct` bpf_struct_ops具体实现.
- 这两个结构是紧密相关的. 使用BPF_STRUCT_OPS_TYPE宏时需要传入一个结构体名称作为参数，该宏会生成一个 struct bpf_struct_ops 结构体的实例; 而`struct` bpf_struct_ops 结构体则用于实现该结构体内部的具体操作

内核支持

./kernel/bpf/bpf_struct_ops_types.h

```
#ifdef CONFIG_SCHED_CLASS_EXT
#include <linux/sched/ext.h>
BPF_STRUCT_OPS_TYPE(sched_ext_ops)
#endif
```

内核支持

内核中对应的操作对象为bpf_sched_ext_ops，定义在 ./kernel/sched/ext.c 中

```
/* "extern" to avoid sparse warning, only used in this file */
extern struct bpf_struct_ops bpf_sched_ext_ops;

struct bpf_struct_ops bpf_sched_ext_ops = {
        .verifier_ops = &bpf_scx_verifier_ops,
        .reg = bpf_scx_reg,
        .unreg = bpf_scx_unreg,
        .check_member = bpf_scx_check_member,
        .init_member = bpf_scx_init_member,
        .init = bpf_scx_init,
        .name = "sched_ext_ops",
};
```

内核支持

- .name指明了其操作了 BPF 结构体对象为 sched_ext_ops，这与用户态的结构体对象保持一致.

内核支持

- `.name`指明了其操作了 BPF 结构体对象为 `sched_ext_ops`，这与用户态的结构体对象保持一致.
- `init()` 被首先调用以进行任何需要的全局设置;

内核支持

- `.name`指明了其操作了 BPF 结构体对象为 `sched_ext_ops`，这与用户态的结构体对象保持一致.
- `init()` 被首先调用以进行任何需要的全局设置;
- `init_member()` 则验证该结构中任何字段的确切值.

内核支持

- .name指明了其操作了 BPF 结构体对象为 sched_ext_ops，这与用户态的结构体对象保持一致.
- init() 被首先调用以进行任何需要的全局设置;
- init_member() 则验证该结构中任何字段的确切值.
- check_member() 确定目标结构的特定成员是否允许在 BPF 中实现;

内核支持

- `.name`指明了其操作了 BPF 结构体对象为 `sched_ext_ops`，这与用户态的结构体对象保持一致.
- `init()` 被首先调用以进行任何需要的全局设置;
- `init_member()` 则验证该结构中任何字段的确切值.
- `check_member()` 确定目标结构的特定成员是否允许在 BPF 中实现;
- `reg()` 函数在检查通过后实际替换了注册结构. 该函数将实际完成将用户态函数注入内核的工作; 有关该函数将在后续的章节进行源码级分析

内核支持

- `.name`指明了其操作了 BPF 结构体对象为 `sched_ext_ops`，这与用户态的结构体对象保持一致.
- `init()` 被首先调用以进行任何需要的全局设置;
- `init_member()` 则验证该结构中任何字段的确切值.
- `check_member()` 确定目标结构的特定成员是否允许在 BPF 中实现;
- `reg()` 函数在检查通过后实际替换了注册结构. 该函数将实际完成将用户态函数注入内核的工作; 有关该函数将在后续的章节进行源码级分析
- `unreg()` 撤销注册;

内核支持

- `.name`指明了其操作了 BPF 结构体对象为 `sched_ext_ops`，这与用户态的结构体对象保持一致.
- `init()` 被首先调用以进行任何需要的全局设置;
- `init_member()` 则验证该结构中任何字段的确切值.
- `check_member()` 确定目标结构的特定成员是否允许在 BPF 中实现;
- `reg()` 函数在检查通过后实际替换了注册结构. 该函数将实际完成将用户态函数注入内核的工作; 有关该函数将在后续的章节进行源码级分析
- `unreg()` 撤销注册;
- `verifier_ops` 结构有一些函数，用于验证各个替换函数是否可以安全执行;

运行结果

编译并实际运行该程序，结果如下：

```
# make -j16 -C tools/sched_ext
# tools/sched_ext/scx_example_dummy -a
local=0 global=3
local=5 global=24
local=9 global=44
local=13 global=56
local=17 global=72
^CEXIT: BPF scheduler unregistered
```

第 III 部分

部署与实验

部署与实验

本部分将说明如何在 QEMU 虚拟机中实际使用 sched_ext 拓展内核以及自定义的 BPF 调度器.

如何使用

- 根据文档所说，只需要打开CONFIG_SCHED_CLASS_EXT选项即可.

如何使用

- 根据文档所说，只需要打开CONFIG_SCHED_CLASS_EXT选项即可.
- 遗憾的是仅此而已并不足够，还需要为内核启用 BTF(BPF 类型格式) 功能并打开一些 Debugging 功能.

如何使用

- 根据文档所说，只需要打开CONFIG_SCHED_CLASS_EXT选项即可.
- 遗憾的是仅此而已并不足够，还需要为内核启用 BTF(BPF 类型格式) 功能并打开一些 Debugging 功能.
  - BTF (BPF Type Format) 是编码 BPF 程序、映射相关的 debug 信息的元数据格式. BTF 这个名字最初是用来描述数据类型. 后来，BTF 被扩展到包括已定义的子程序的函数信息和行信息.

工具链准备

- 内核选项配置完成后尝试编译，rustc 编译器将会报错，主要涉及到这个 Makefile Object:

```
atropos: export RUSTFLAGS = -C link-args=-lzstd
atropos: export ATROPOS_CLANG = $(CLANG)
atropos: export ATROPOS_BPF_CFLAGS = $(BPF_CFLAGS)
atropos: $(INCLUDE_DIR)/vmlinux.h
        cargo build --manifest-path=atropos/Cargo.toml $(CARGOFLAGS)
```

工具链准备

- 内核选项配置完成后尝试编译，rustc 编译器将会报错，主要涉及到这个 Makefile Object:

```
atropos: export RUSTFLAGS = -C link-args=-lzstd
atropos: export ATROPOS_CLANG = $(CLANG)
atropos: export ATROPOS_BPF_CFLAGS = $(BPF_CFLAGS)
atropos: $(INCLUDE_DIR)/vmlinux.h
        cargo build --manifest-path=atropos/Cargo.toml $(CARGOFLAGS)
```

  - Atropos Rust 是一个用于编写基于 BPF(Berkeley Packet Filter) 的网络应用程序的 Rust 库. Atropos Rust 使用 eBPF 技术，在内核空间中运行 BPF 程序，并将其与用户空间的 Rust 代码集成在一起.

工具链准备

- 内核选项配置完成后尝试编译，rustc 编译器将会报错，主要涉及到这个 Makefile Object:

```
atropos: export RUSTFLAGS = -C link-args=-lzstd
atropos: export ATROPOS_CLANG = $(CLANG)
atropos: export ATROPOS_BPF_CFLAGS = $(BPF_CFLAGS)
atropos: $(INCLUDE_DIR)/vmlinux.h
	cargo build --manifest-path=atropos/Cargo.toml $(CARGOFLAGS)
```

  - Atropos Rust 是一个用于编写基于 BPF(Berkeley Packet Filter) 的网络应用程序的 Rust 库. Atropos Rust 使用 eBPF 技术，在内核空间中运行 BPF 程序，并将其与用户空间的 Rust 代码集成在一起.
- 为了编译 Atropos，rust 需要用到`libclang.a`静态链接库:

```
/// Finds a directory containing LLVM and Clang static libraries and returns the
/// path to that directory.
fn find() -> PathBuf {
    let name = if target_os!("windows") {
        "libclang.lib"
    } else {
        "libclang.a"
    };

    let files = common::search_libclang_directories(&[name.into()], "LIBCLANG_STATIC_PATH");
    if let Some((directory, _)) = files.into_iter().next() {
        directory
    } else {
        panic!("could not find any static libraries");
    }
}
```

工具链准备

- 首要想法还是看看能不能关闭静态链接 (事实上这个想法很蠢，因为静态链接就是为了移植到 qemu 下系统库不完备的文件系统中，关了那肯定运行不了). 在 sched_ext/tools/sched_ext/atropos/Cargo.toml L27 删除 feature 中的 static 参数即可:

```
bindgen = { version = "0.61.0", features = ["logging"], default-features = false }
```

工具链准备

- 首要想法还是看看能不能关闭静态链接 (事实上这个想法很蠢，因为静态链接就是为了移植到 qemu 下系统库不完备的文件系统中，关了那肯定运行不了). 在 sched_ext/tools/sched_ext/atropos/Cargo.toml L27 删除 feature 中的 static 参数即可:

```
bindgen = { version = "0.61.0", features = ["logging"], default-features = false }
```

  - bindkey 这个 rust 库是用于构建绑定 C/C++ 标准库的 Rust EFI 程序用的.

工具链准备

- 首要想法还是看看能不能关闭静态链接 (事实上这个想法很蠢，因为静态链接就是为了移植到 qemu 下系统库不完备的文件系统中，关了那肯定运行不了). 在 sched_ext/tools/sched_ext/atropos/Cargo.toml L27 删除 feature 中的 static 参数即可:

```
bindgen = { version = "0.61.0", features = ["logging"], default-features = false }
```

  - bindkey 这个 rust 库是用于构建绑定 C/C++ 标准库的 Rust EFI 程序用的.
- 不过可以料想在 Busybox 环境中，关闭静态链接方式的 bpf 程序肯定是无法运行的.

工具链准备

- 考虑在本机配置 clang 的静态链接环境.

工具链准备

- 考虑在本机配置 clang 的静态链接环境.
- 坏消息是, 大部分常见的发行版 (Debian, Ubunut, Fedora, Arch 等) 的官方软件源中都没有提供静态链接的 clang 库文件, 这意味着我们必须从官网自行编译`libclang.a`文件.

工具链准备

- 从 Github 仓库获取 llvm-project 套件的源码包. 由于 llvm 套件的更新较快, 网上查阅到的资料可能已经过旧. 这里直接下载与本机版本 (15.0.7) 一致的源码包.

工具链准备

- 从 Github 仓库获取 llvm-project 套件的源码包. 由于 llvm 套件的更新较快, 网上查阅到的资料可能已经过旧. 这里直接下载与本机版本 (15.0.7) 一致的源码包.
- 较新的 llvm 套件有子项目的概念, 比如 llvm 和 clang 就是其两个子项目. 用户可以自行选择编译哪些子项目.

工具链准备

- 从 Github 仓库获取 llvm-project 套件的源码包. 由于 llvm 套件的更新较快, 网上查阅到的资料可能已经过旧. 这里直接下载与本机版本 (15.0.7) 一致的源码包.
- 较新的 llvm 套件有子项目的概念, 比如 llvm 和 clang 就是其两个子项目. 用户可以自行选择编译哪些子项目.
- 在 cmake 中指定子项目的路径, 加上一堆参数强制编译出静态链接库.

```
mkdir build; cd build
cmake ../clang -DCMAKE_INSTALL_PREFIX=$HOME/SCX/clang/install -DBUILD_SHARED_LIBS=OFF \
-DLLVM_BUILD_LLVM_DYLIB=ON -DLIBCLANG_BUILD_STATIC=ON \
-DCMAKE_BUILD_TYPE=Release -DCLANG_BUILD_EXAMPLES=OFF -DCLANG_INCLUDE_DOCS=OFF \
-DCLANG_INCLUDE_TESTS=OFF -DLLVM_APPEND_VC_REV=OFF -DLLVM_BUILD_DOCS=OFF \
-DLLVM_BUILD_EXAMPLES=OFF -DLLVM_BUILD_TESTS=OFF \
-DLLVM_BUILD_TOOLS=ON -DLLVM_ENABLE_ASSERTIONS=OFF -DLLVM_ENABLE_CXX1Y=ON \
-DLLVM_ENABLE_EH=ON -DLLVM_ENABLE_LIBCXX=OFF -DLLVM_ENABLE_PIC=ON \
-DLLVM_ENABLE_RTTI=ON -DLLVM_ENABLE_SPHINX=OFF -DLLVM_ENABLE_TERMINFO=OFF \
-DLLVM_INCLUDE_DOCS=OFF -DLLVM_INCLUDE_EXAMPLES=OFF -DLLVM_INCLUDE_GO_TESTS=OFF \
-DLLVM_INCLUDE_TESTS=OFF -DLLVM_INCLUDE_TOOLS=ON -DLLVM_INCLUDE_UTILS=OFF \
-DLLVM_PARALLEL_LINK_JOBS=1 -DLLVM_TARGETS_TO_BUILD="host;BPF"
```

工具链准备

- 从 Github 仓库获取 llvm-project 套件的源码包. 由于 llvm 套件的更新较快, 网上查阅到的资料可能已经过旧. 这里直接下载与本机版本 (15.0.7) 一致的源码包.
- 较新的 llvm 套件有子项目的概念, 比如 llvm 和 clang 就是其两个子项目. 用户可以自行选择编译哪些子项目.
- 在 cmake 中指定子项目的路径, 加上一堆参数强制编译出静态链接库.

```
mkdir build; cd build
cmake ../clang -DCMAKE_INSTALL_PREFIX=$HOME/SCX/clang/install -DBUILD_SHARED_LIBS=OFF \
-DLLVM_BUILD_LLVM_DYLIB=ON -DLIBCLANG_BUILD_STATIC=ON \
-DCMAKE_BUILD_TYPE=Release -DCLANG_BUILD_EXAMPLES=OFF -DCLANG_INCLUDE_DOCS=OFF \
-DCLANG_INCLUDE_TESTS=OFF -DLLVM_APPEND_VC_REV=OFF -DLLVM_BUILD_DOCS=OFF \
-DLLVM_BUILD_EXAMPLES=OFF -DLLVM_BUILD_TESTS=OFF \
-DLLVM_BUILD_TOOLS=ON -DLLVM_ENABLE_ASSERTIONS=OFF -DLLVM_ENABLE_CXX1Y=ON \
-DLLVM_ENABLE_EH=ON -DLLVM_ENABLE_LIBCXX=OFF -DLLVM_ENABLE_PIC=ON \
-DLLVM_ENABLE_RTTI=ON -DLLVM_ENABLE_SPHINX=OFF -DLLVM_ENABLE_TERMINFO=OFF \
-DLLVM_INCLUDE_DOCS=OFF -DLLVM_INCLUDE_EXAMPLES=OFF -DLLVM_INCLUDE_GO_TESTS=OFF \
-DLLVM_INCLUDE_TESTS=OFF -DLLVM_INCLUDE_TOOLS=ON -DLLVM_INCLUDE_UTILS=OFF \
-DLLVM_PARALLEL_LINK_JOBS=1 -DLLVM_TARGETS_TO_BUILD="host;BPF"
```

- 其中的DLIBCLANG_BUILD_STATIC=ON可能最为重要

工具链准备

编译出来的目标文件树大概为这样:

```
.
├── bin
├── include
├── lib
│   ├── clang
│   ├── cmake
│   ├── libclang.a
│   ├── libclangAnalysis.a
│   ├── libclangAnalysisFlowSensitive.a
│   ├── libclangAnalysisFlowSensitiveModels.a
│   ├── libclangAPINotes.a
│   ├── libclangARCMigrate.a
│   ├── libclangAST.a
│   ├── ...
│   ├── libclang.so -> libclang.so.15
│   ├── libclang.so.15 -> libclang.so.15.0.7
│   ├── libclang.so.15.0.7
│   ├── ...
│   ├── libear
│   └── libscanbuild
├── libexec
└── share
```

图 3: clang 编译产生的目标文件树

工具链准备

- `lib/`里面貌似已经有了我们想要的内容!

工具链准备

- lib/里面貌似已经有了我们想要的内容!
- 不过发现 libclang.a 居然很小, 最大的居然是 libclang.so:

```
# ls -lh | awk '{print $5, $9}' | sort -hr
73M libclang-cpp.so.15
39M libclang.so.15.0.7
27M libclangSema.a
22M libclangStaticAnalyzerCheckers.a
17M libclangAST.a
16M libclangCodeGen.a
13M libclangBasic.a
11M libclangDynamicASTMatchers.a
8.5M libclangStaticAnalyzerCore.a
7.0M libclangDriver.a
6.5M libclangARCMigrate.a
5.9M libclangTransformer.a
4.6M libclangAnalysis.a
4.3M libclangFrontend.a
...
```

工具链准备

- 先直接将`libcalng.a`放进去看一下 (放到了/usr/local/lib/里, 不要污染本机的 lib 路径). 不过并不能成功编译, 报错缺少少一些函数的实现.

工具链准备

- 先直接将`libcalng.a`放进去看一下 (放到了/usr/local/lib/里, 不要污染本机的 lib 路径). 不过并不能成功编译, 报错缺少少一些函数的实现.
- 那么感觉实际上 clang 是把不同的功能分散了多个.a 文件中了, 因而如果想要静态编译还得把所有的静态库全部塞到路径里面去, 那么直接将该 lib 文件夹整体放到路径下, 再次编译, 又报下错:

```
"kernel/bpf/btf.c:6403:2: error: #warning "btf argument scalar test nerfed" [-Werror=cpp]"
```

工具链准备

- 先直接将libcalng.a放进去看一下 (放到了/usr/local/lib/里, 不要污染本机的 lib 路径). 不过并不能成功编译, 报错缺少少一些函数的实现.
- 那么感觉实际上 clang 是把不同的功能分散了多个.a 文件中了, 因而如果想要静态编译还得把所有的静态库全部塞到路径里面去, 那么直接将该 lib 文件夹整体放到路径下, 再次编译, 又报下错:

```
"kernel/bpf/btf.c:6403:2: error: #warning "btf argument scalar test nerfed" [-Werror=cpp]"
```

- bpf 内部产生了 Warning! 并不很想直接关闭 Warning 警告选项, 因为产生该 Warning 意味着 bpf 内核代码执行到了开发者不曾设想的道路上, 可能会产生意想不到的错误. 不过目前也只能关闭该内核编译参数了:

```
# KBUILD_CFLAGS-$(CONFIG_WERROR) += -Werror
```

工具链准备

- 先直接将libcalng.a放进去看一下 (放到了/usr/local/lib/里, 不要污染本机的 lib 路径). 不过并不能成功编译, 报错缺少少一些函数的实现.
- 那么感觉实际上 clang 是把不同的功能分散了多个.a 文件中了, 因而如果想要静态编译还得把所有的静态库全部塞到路径里面去, 那么直接将该 lib 文件夹整体放到路径下, 再次编译, 又报下错:

```
"kernel/bpf/btf.c:6403:2: error: #warning "btf argument scalar test nerfed" [-Werror=cpp]"
```

- bpf 内部产生了 Warning! 并不很想直接关闭 Warning 警告选项, 因为产生该 Warning 意味着 bpf 内核代码执行到了开发者不曾设想的道路上, 可能会产生意想不到的错误. 不过目前也只能关闭该内核编译参数了:

```
# KBUILD_CFLAGS-$(CONFIG_WERROR) += -Werror
```

- 至此可以成功编译出搭载有 scx 的内核镜像!

QEMU+Busybox 启动

- 通过 Busybox 制作一个极简的文件系统

QEMU+Busybox 启动

- 通过 Busybox 制作一个极简的文件系统
- 编译 scx 提供的几个示例程序

```
make -j16 -C tools/sched_ext
```

QEMU+Busybox 启动

- 通过 Busybox 制作一个极简的文件系统
- 编译 scx 提供的几个示例程序

```
make -j16 -C tools/sched_ext
```

- 将可执行文件放到 rootfs 中，随后生成 rootfs 镜像文件.

QEMU+Busybox 启动

- 通过 Busybox 制作一个极简的文件系统
- 编译 scx 提供的几个示例程序

```
make -j16 -C tools/sched_ext
```

- 将可执行文件放到 rootfs 中，随后生成 rootfs 镜像文件.
- 启动 QMEMU，试图执行这几个 scx 可执行文件，但是很奇怪的是…

QEMU+Busybox 启动


```
/ # ls -al
total 5612
drwxr-xr-x    9 1000     1000           360 Apr 17 06:36 .
drwxr-xr-x    9 1000     1000           360 Apr 17 06:36 ..
-rw-------    1 0        0               48 Apr 17 06:37 .ash_history
drwxr-xr-x    2 1000     1000          5380 Apr 16 10:00 bin
drwxr-xr-x    2 1000     1000           180 Apr 16 10:01 dev
-rwxr-xr-x    1 1000     1000            97 Apr 16 10:01 init
-rwxr-xr-x    1 1000     1000        736056 Apr 16 10:00 main
-rw-r--r--    1 1000     1000           184 Apr 16 10:00 main.c
dr-xr-xr-x  105 0        0               0 Apr 17 06:36 proc
drwx------    2 0        0              40 Apr  9 11:51 root
drwxr-xr-x    2 1000     1000          2660 Apr 16 10:00 sbin
-rwxr-xr-x    1 1000     1000        988424 Apr 17 03:29 scx_example_central
-rwxr-xr-x    1 1000     1000        971864 Apr 17 03:29 scx_example_dummy
-rwxr-xr-x    1 1000     1000       1043720 Apr 17 03:29 scx_example_pair
-rwxr-xr-x    1 1000     1000        984568 Apr 17 03:29 scx_example_qmap
-rwxr-xr-x    1 1000     1000        995464 Apr 17 03:29 scx_example_userland
dr-xr-xr-x   12 0        0               0 Apr 17 06:36 sys
drwxr-xr-x    2 1000     1000           80 Apr 16 09:59 trusted
/ # ./main
Privilege level: 33
FS: 0
/ # ./scx_example_dummy
-/bin/sh: ./scx_example_dummy: not found
/ # ./scx_example_central
-/bin/sh: ./scx_example_central: not found
```

QEMU+Busybox 启动

- 并不知道如何能够修理 Busybox 出现的这个奇怪的问题

QEMU+Busybox 启动

- 并不知道如何能够修理 Busybox 出现的这个奇怪的问题
- 怀疑还是 Busybox 太简朴了. 在网络上搜索时发现了 buildroot 套件, 可以理解为 Busybox 的超集.

QEMU+Busybox 启动

- 并不知道如何能够修理 Busybox 出现的这个奇怪的问题
- 怀疑还是 Busybox 太简朴了. 在网络上搜索时发现了 buildroot 套件, 可以理解为 Busybox 的超集.
- 尝试使用 buildroot 做根文件系统吧!

BuildRoot

下载，配置，编译[4]

```
git clone git://git.buildroot.net/buildroot /source/buildroot
cd buildroot
make menuconfig
make -j16
```

4.

5 https://mp.weixin.qq.com/s/Bh19_embfHDvbIcEEpQ2mw

BuildRoot

- 下载，配置，编译[4]

```
git clone git://git.buildroot.net/buildroot /source/buildroot
cd buildroot
make menuconfig
make -j16
```

- 具体的配置选项可以参考 < 调试 eBPF 虚拟机 >[5]

4.

5 https://mp.weixin.qq.com/s/Bh19_embfHDvbIcEEpQ2mw

BuildRoot

- 下载，配置，编译[4]

```
git clone git://git.buildroot.net/buildroot /source/buildroot
cd buildroot
make menuconfig
make -j16
```

- 具体的配置选项可以参考 < 调试 eBPF 虚拟机 >[5]
- Buildroot 生成的文件系统拥有完整的系统库，可以按需要下载任意需要的软件并编译进入该系统. 这里我们需要安装 clang 等相关的套件.

4:

5 https://mp.weixin.qq.com/s/Bh19_embfHDvbIcEEpQ2mw

运行结果

使用 Buildroot 构建的文件系统可以成功在 QEMU 上执行，示例程序可以正常运行:

```
# uname -a
Linux buildroot 6.1.0-rc4-gc42fc855558d-dirty #3 SMP PREEMPT_DYNAMIC Mon Apr 17 11:56:40 CST 2023 x86_64 GNU/Linux
# ./scx_example_
scx_example_central    scx_example_pair      scx_example_userland
scx_example_dummy      scx_example_qmap
# ./scx_example_central -h
A central FIFO sched_ext scheduler.

See the top-level comment in .bpf.c for more details.

Usage: scx_example_central [-a] [-c CPU]

  -a            Switch all tasks
  -c CPU        Override the central CPU (default: 0)
  -h            Display this help and exit
# ./scx_example_central -a
[SEQ 0]
total:         0    local:         0   queued:         0     lost:         0
timer:         5 dispatch:         0 mismatch:         0 overflow:         0
[SEQ 1]
total:         9    local:         7   queued:         0     lost:         0
timer:       872 dispatch:       876 mismatch:         0 overflow:         0
[SEQ 2]
total:        17    local:        10   queued:         0     lost:         0
timer:      1739 dispatch:      1748 mismatch:         0 overflow:         0
[SEQ 3]
total:        25    local:        13   queued:         0     lost:         0
timer:      2611 dispatch:      2626 mismatch:         0 overflow:         0
[SEQ 4]
total:        41    local:        25   queued:         0     lost:         0
timer:      3488 dispatch:      3512 mismatch:         0 overflow:         0
[SEQ 5]
total:        49    local:        29   queued:         0     lost:         0
timer:      4358 dispatch:      4387 mismatch:         0 overflow:         0
[SEQ 6]
total:        71    local:        47   queued:         0     lost:         0
timer:      5226 dispatch:      5266 mismatch:         0 overflow:         0
[SEQ 7]
total:        79    local:        52   queued:         0     lost:         0
timer:      6092 dispatch:      6137 mismatch:         0 overflow:         0
qemu-system-x86_64: terminating on signal 2
```

5个示例程序

帮助菜单

使用eBPF装载进入的调度策略

第 IV 部分

内核基础设施

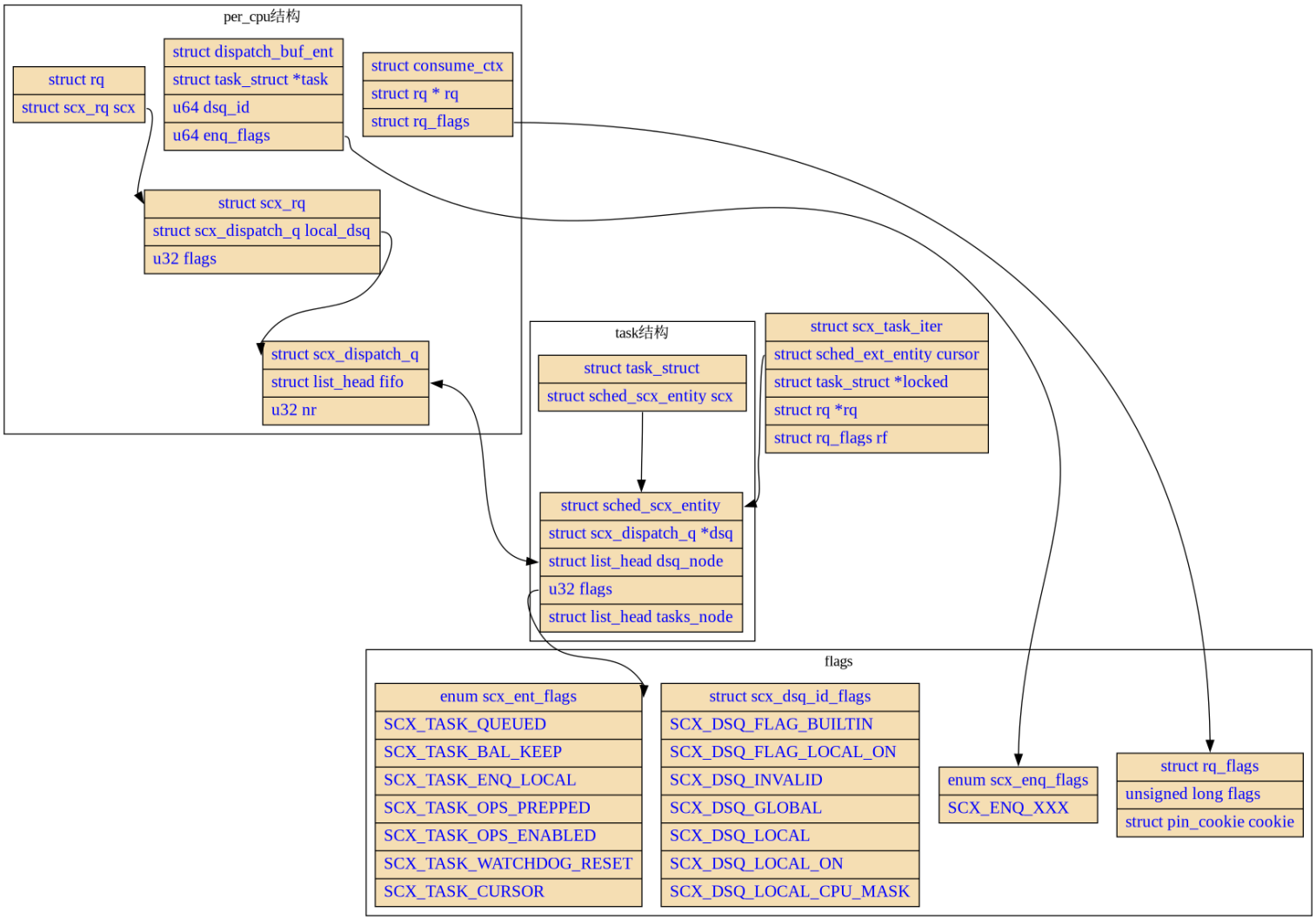
内核基础设施

sched_ext 机制使用调度类的方式装载进入原生内核结构. 本章节将简要概括说明一些内核调度系统与 sched_ext 机制使用到的内核基础设施.

内核基础设施

可拓展调度器类 sched_ext 技术

施鹏飞



内核原生结构

sched_ext 或利用或修改了一些内核的原生结构:

- `struct rq` 运行队列
- `struct task_struct` 进程描述符
- `struct sched_class` 调度类

struct rq

运行队列是一个每 CPU 结构，其用于管理操作系统内核中的进程和线程。每个 CPU 都有自己的 struct_rq 数据结构来管理本地运行队列。

struct task_struct

运行队列是 Linux 调度中最重要的数据结构，其拥有大量的字段；在 sched_ext 中，其主要增添了一个`struct scx_rq`类型、名为`scx`的成员.

```
struct rq {
        ...
        struct cfs_rq           cfs;
        struct rt_rq            rt;
        struct dl_rq            dl;
#ifdef CONFIG_SCHED_CLASS_EXT
        struct scx_rq           scx;
#endif
        ...
};
```

struct rq_flags

`struct rq_flags`是一个内核原生结构，其用于管理运行队列标志位，可以帮助调度器更好地控制和管理进程的调度.

- flags 字段是一个无符号长整型数，用于存储运行队列的各种状态和属性. flags 中的每一位都对应着一个特定的标志位，可以通过位运算等方式进行设置和读取.
- cookie 字段是一个结构体，用于存储固定进程的相关信息. 在 Linux 内核中，固定进程指的是不会被调度器迁移的进程，例如 init 进程和 kthreadd 进程等. cookie 信息可以帮助调度器识别和处理固定进程.
- clock_update_flags 字段只在 CONFIG_SCHED_DEBUG 配置选项开启时存在. itms

```
struct rq_flags {
        unsigned long flags;
        struct pin_cookie cookie;
#ifdef CONFIG_SCHED_DEBUG
        /*
         * A copy of (rq::clock_update_flags & RQCF_UPDATED) for the
         * current pin context is stashed here in case it needs to be
         * restored in rq_repin_lock().
         */
        unsigned int clock_update_flags;
#endif
};
```

struct task_struct

在任务描述符中也插入了一个新的字段，sched_ext_entity类型的字段scx用于存储当前任务有关 scx 调度的一些信息，比如其目前处在哪个 rq 上，scx 分发标志，list_head 成员等；而原本的sched_class则也会被 sched_ext 机制所使用到.

```
struct task_struct {
  ...
#ifdef CONFIG_SCHED_CLASS_EXT
        struct sched_ext_entity         scx;
#endif
        const struct sched_class        *sched_class;
  ...
```

sched_class

调度类结构. sched_ext 未做修改, 但是利用了该字段用来装载自己进入调度系统.

```
/* file: kernel/sched/sched.h */
struct sched_class {
        void (*enqueue_task)(struct rq *rq, struct task_struct *p, int flags);
        void (*dequeue_task)(struct rq *rq, struct task_struct *p, int flags);
        void (*yield_task)(struct rq *rq);
        bool (*yield_to_task)(struct rq *rq, struct task_struct *p);
        struct task_struct *(*pick_next_task)(struct rq *rq);
        void (*put_prev_task)(struct rq *rq, struct task_struct *p);
        void (*set_next_task)(struct rq *rq, struct task_struct *p, bool first);
        int (*balance)(struct rq *rq, struct task_struct *prev, struct rq_flags *rf);
        ...
};
```

sched_class

sched_ext 调度器类实现若干接口函数:

```
DEFINE_SCHED_CLASS(ext) = {
        .enqueue_task           = enqueue_task_scx,
        .dequeue_task           = dequeue_task_scx,
        .yield_task             = yield_task_scx,
        .yield_to_task          = yield_to_task_scx,
        .check_preempt_curr     = check_preempt_curr_scx,
        .pick_next_task         = pick_next_task_scx,
        .put_prev_task          = put_prev_task_scx,
        .set_next_task          = set_next_task_scx,
#ifdef CONFIG_SMP
        .balance                = balance_scx,
        .select_task_rq         = select_task_rq_scx,
        .pick_task              = pick_task_scx,
        .set_cpus_allowed       = set_cpus_allowed_scx,
        .rq_online              = rq_online_scx,
        .rq_offline             = rq_offline_scx,
#endif
        .task_tick              = task_tick_scx,
        .switching_to           = switching_to_scx,
        .switched_to            = switched_to_scx,
        .reweight_task          = reweight_task_scx,
        .prio_changed           = prio_changed_scx,
        .update_curr            = update_curr_scx,
};
```

SCX 数据结构

为了实现拓展调度功能，sched_ext 在定义了多个新数据结构. 这里选择较为核心的数据结构进行说明.

struct scx_rq

每个运行队列均有一个scx_rq结构. 该结构中进一步拥有scx_dispatch_q结构的local_dsq, 以及其他若干标志等字段.

```
#ifdef CONFIG_SCHED_CLASS_EXT
/* scx_rq->flags, protected by the rq lock */
66 enum scx_rq_flags {
        SCX_RQ_CAN_STOP_TICK    = 1 << 0,
};

struct scx_rq {
        struct scx_dispatch_q   local_dsq;
        struct list_head        watchdog_list;
        u64                     ops_qseq;
        u32                     nr_running;
        u32                     flags;
        bool                    cpu_released;
#ifdef CONFIG_SMP
        cpumask_var_t           cpus_to_kick;
        cpumask_var_t           cpus_to_preempt;
        cpumask_var_t           cpus_to_wait;
        u64                     pnt_seq;
        struct irq_work         kick_cpus_irq_work;
#endif
};
#endif /* CONFIG_SCHED_CLASS_EXT */
```

scx_dispatch_q

- scxc_dispatch_q拥有一个锁结构; 由于其他的 CPU 可能会来本 CPU 的 dsq 中取任务, 因而锁是必要的
- fifo字段将处于本 dsq 的所有的任务串在一起.

```
/*
 * Dispatch queue (dsq) is a simple FIFO which is used to buffer between the
 * scheduler core and the BPF scheduler. See the documentation for more details.
 */
struct scx_dispatch_q {
        raw_spinlock_t          lock;
        struct list_head        fifo;
        u64                     id;
        u32                     nr;
        struct rhash_head       hash_node;
        struct list_head        all_node;
        struct llist_node       free_node;
        struct rcu_head         rcu;
};
```

struct sched_ext_entity

每个任务都在描述符中有一个该类型的实体，其中包含了一个任务能够被 SCX 所调用需要的所有的字段内容.

- dsq指向其所在的运行队列
- dsq_node则用来串接任务链表.
- flags指向的是一个enum scx_ent_flags的结构
- tasks_node用于在 BPF 函数中对所有的可执行任务进行链表索引.

```
struct sched_ext_entity {
        struct scx_dispatch_q   *dsq;
        struct list_head        dsq_node;
        struct list_head        watchdog_node;
        u32                     flags;          /* protected by rq lock */
        u32                     weight;
        s32                     sticky_cpu;
        s32                     holding_cpu;
        atomic64_t              ops_state;
        unsigned long           runnable_at;
        u64                     slice;
        bool                    disallow;       /* reject switching into SCX */
        struct list_head        tasks_node;
#ifdef CONFIG_EXT_GROUP_SCHED
        struct cgroup           *cgrp_moving_from;
#endif
};
```

struct dispatch_buf_ent

该结构是一个每 CPU 结构，用于辅助完成分发工作.

- 向本地 dsq 中分发任务可能需要等待排队完成或请求锁. 为了避免在 ops.dispatch()内部执行额外的编码工作以避免锁的顺序倒置问题，因此其将调度分为两个部分.
- scx_bpf_dispatch()由ops.dispatch()调用，其会将任务以及 dsq 等信息存放在该 buffer 中. 一旦 ops.dispatch()返回，再调用 finish_dispatch利用 buffer 中的数据来完成整个过程.

```
/* dispatch buf */
struct dispatch_buf_ent {
        struct task_struct      *task;
        u64                     qseq;
        u64                     dsq_id;
        u64                     enq_flags;
};

static u32 dispatch_max_batch;
static struct dispatch_buf_ent __percpu *dispatch_buf;
static DEFINE_PER_CPU(u32, dispatch_buf_cursor);
```

struct dispatch_buf_ent

dispatch_buf_ent的指针数组结构的初始化如下:

```
dispatch_buf = __alloc_percpu(sizeof(dispatch_buf[0]) * dispatch_max_batch,
                              __alignof__(dispatch_buf[0]));
```

dispatch_buf[0] 实际是一个 dispatch_buf_ent 结构体的类型, 尽管还没有初始化, 但是可以用于计算 sizeof, 这一点值得学习. 使用一个结构体指针即可统一的实现 malloc, 而不需要在 malloc 内部关注指针指向的类型

struct consume_ctx

该结构仅在balance_scx方法中被设置，并在scx_bpf_consume方法中被使用. balance_scx方法绑定到了调度类的blance接口上，该接口是内核中用于负载均衡的调度器接口. 因而实际上consume_ctx就指定了当前 CPU 应该从哪里取得下一个任务，当下一次负载均衡发生时该结构的内容也会相应的调整.

```
/* consume context */
struct consume_ctx {
        struct rq               *rq;
        struct rq_flags         *rf;
};

static DEFINE_PER_CPU(struct consume_ctx, consume_ctx);
```

标志类型

sched_ext 技术中定义了许多枚举/结构体类型的标志结构，用于说明各种操作细节.

enum scx_dsq_id_flags

该枚举类型通过 64bit 整数来标识一个 dsq.

```
/*
 * DSQ (dispatch queue) IDs are 64bit of the format:
 *
 *   Bits: [63] [62 ..  0]
 *         [ B] [   ID   ]
 *
 *    B: 1 for IDs for built-in DSQs, 0 for ops-created user DSQs
 *   ID: 63 bit ID
 *
 * Built-in IDs:
 *
 *   Bits: [63] [62] [61..32] [31 ..  0]
 *         [ 1] [ L] [   R  ] [    V   ]
 *
 *    1: 1 for built-in DSQs.
 *    L: 1 for LOCAL_ON DSQ IDs, 0 for others
 *    V: For LOCAL_ON DSQ IDs, a CPU number. For others, a pre-defined value.
 */
enum scx_dsq_id_flags {
        SCX_DSQ_FLAG_BUILTIN    = 1LLU << 63,
        SCX_DSQ_FLAG_LOCAL_ON   = 1LLU << 62,

        SCX_DSQ_INVALID         = SCX_DSQ_FLAG_BUILTIN | 0,
        SCX_DSQ_GLOBAL          = SCX_DSQ_FLAG_BUILTIN | 1,
        SCX_DSQ_LOCAL           = SCX_DSQ_FLAG_BUILTIN | 2,
        SCX_DSQ_LOCAL_ON        = SCX_DSQ_FLAG_BUILTIN | SCX_DSQ_FLAG_LOCAL_ON,
        SCX_DSQ_LOCAL_CPU_MASK  = 0xffffffffLLU,
};
```

enum scx_enq_flags

该字段用于在分发系列的函数（如finish_dispatch, scx_bpf_dispatch等）中使用，也存在于dispatch_buf_ent的enq_flags字段中.

- 前两个字段是将内核原生的两个标志位暴露出去
- PREEMPT使得任务分发完成后立刻执行一次抢占.
- LAST标志当前分发的任务是当前 CPU 唯一可用的任务.
- LOCAL标志建议将当前任务优先分发到本地 dsq 中.

```
enum scx_enq_flags {
        SCX_ENQ_WAKEUP          = ENQUEUE_WAKEUP,
        SCX_ENQ_HEAD            = ENQUEUE_HEAD,

        SCX_ENQ_PREEMPT         = 1LLU << 32,
        SCX_ENQ_REENQ           = 1LLU << 40,
        SCX_ENQ_LAST            = 1LLU << 41,
        SCX_ENQ_LOCAL           = 1LLU << 42,
        __SCX_ENQ_INTERNAL_MASK = 0xffLLU << 56,
        SCX_ENQ_CLEAR_OPSS      = 1LLU << 56,
};
```

enum scx_ent_flags

说明了当前的 scx 实体的状态如何.

```
/* scx_entity.flags */
enum scx_ent_flags {
        SCX_TASK_QUEUED         = 1 << 0, /* on ext runqueue */
        SCX_TASK_BAL_KEEP       = 1 << 1, /* balance decided to keep current */
        SCX_TASK_ENQ_LOCAL      = 1 << 2, /* used by scx_select_cpu_dfl() to set SCX_ENQ_LOCAL */

        SCX_TASK_OPS_PREPPED    = 1 << 3, /* prepared for BPF scheduler enable */
        SCX_TASK_OPS_ENABLED    = 1 << 4, /* task has BPF scheduler enabled */

        SCX_TASK_WATCHDOG_RESET = 1 << 5, /* task watchdog counter should be reset */

        SCX_TASK_CURSOR         = 1 << 6, /* iteration cursor, not a task */
};
```

m scx_ops_enable_state

定义了 ops 结构体中的函数状态.

```
enum scx_ops_enable_state {
        SCX_OPS_PREPPING,
        SCX_OPS_ENABLING,
        SCX_OPS_ENABLED,
        SCX_OPS_DISABLING,
        SCX_OPS_DISABLED,
};
```

第 V 部分

sched_ext 内核函数

sched_ext 内核函数

本章节我们将深入内核源码观察 sched_ext 都实现了哪些内核函数供 BPF 程序使用.

大部分的 SCX 内核函数均被定义在./kernel/sched/ext.c 中

scx_bpf_dispatch

`scx_bpf_dispatch`用于任务的分发，该函数在之前的 dummy 例子中第一个出现，因而我们从这里开始.

scx_bpf_dispatch

- 该函数的作用是将一个 task 分发到某个 dsq 上
- 其接受 4 个参数，其中`slice`表示任务可以执行的时间.
- 总的来说，该函数将指定的任务分发到指定的每 CPU 的 dsq 队列中，同时维护每 CPU 结构和每任务结构的双向连接关系，当任务开启了 preempt 标志后还会手动地触发一次 rq 的重调度.

scx_bpf_dispatch

下面对其具体的细节进行说明:

- 内核有死锁检测的功能 lockdep, 该功能由 CONFIG_PROVE_LOCKING内核选项定义, 具体的细节不予深究. lockdep_assert_irqs_disabled 是一个与此有关的宏函数. 当开启了内核死锁检测后, 该函数将执行:

```
#define lockdep_assert_irqs_disabled()                                  \
do {                                                                    \
  WARN_ON_ONCE(__lockdep_enabled && this_cpu_read(hardirqs_enabled));   \
} while (0)
```

可以认为该函数其实就是个 assert, 保证此时中断是关闭的.

scx_bpf_dispatch

- 随后做一些基本的检查，比如 p 是否为空，以及是否设置了非法的 enq_flags
- 如果传入的 slice 为 0，则其原本的 slice 状态将被保持. 不过由于结合 sched_ext_entity 的 slice 字段的内容可知，该字段如果为 0 则意味着其被带有%SCX_KICK_PREEMPT 标志的任务抢占，此时该字段不能用于决定当前任务能够执行多久；因而将其修改为一个非零结果.

scx_bpf_dispatch

- 随后读一个当前 CPU 的每 CPU 变量 direct_dispatch_task，如果不空，则说明要进行直接分发，进而执行 direct_dispatch 函数.

```
/*
 * Direct dispatch marker.
 *
 * Non-NULL values are used for direct dispatch from enqueue path. A valid
 * pointer points to the task currently being enqueued. An ERR_PTR value is used
 * to indicate that direct dispatch has already happened.
 */
static DEFINE_PER_CPU(struct task_struct *, direct_dispatch_task);
```

- 该变量其实只是一个标志，一个非空的结果用于 enqueue 内核路径上的直接分发过程. 如果指针指向的内容合法，则其指向的是当前正在被分发的任务；如果指针内容为ERR_PTR则说明直接分发已经发生过了

direct_dispatch

- 该函数首先做检查，ddsp 必须和 p 一致
- 不允许带有%SCX_DSQ_LOCAL_ON 标志的任务发生直接分发，原因是在 enqueue 内核路径上不允许解锁 rq，因而无法向其他 CPU 的本地 dsq 分发任务
- task_rq 将获取 p 任务所在的 CPU 的 rq 队列，将 rq 队列传入给 find_dsq_for_dispatch 获取 dsq 队列指针.

```
#define task_rq(p)              cpu_rq(task_cpu(p))
```

find_dsq_for_dispatch

- 本地 DSQ 则直接返回，不然根据 dsq_id 调用函数 find_non_local_dsq 获取对应的 dsq 指针
- 如果 dsq_id 对应的 dsq 烂了，则缺省返回全局 dsq

find_non_local_dsq

- 全局 DSQ 则直接返回，不然查 Hash 表返回 dsq_id 对应的 dsq.

rhashtable_lookup_fast

快速哈希表查询函数，不需要上 RCU 读锁.

dispatch_enqueue

这是一个 static 内部函数，做的工作主要是将 task_struct 对应的进程插入到 dsq 中：

- 如果该 dsq 不是一个 CPU 本地的 dsq，则需要上锁避免多 CPU 的竞争；同时检查一下其是否还有效，因为可能被其他的 CPU 进程销毁了，如果已经失效则缺省变为全局 dsq.
- 检查入队标志，据此决定是否插入到队头或队尾，并维护 dsq 和 task_struct 数据结构中的一些字段内容，使得每 CPU 变量 struct scx_dispatch_q 中的 fifo 链表可以与任务描述符中的 scx.dsq_node 相关联
- 如果 dsq 是全局 dsq 释放锁后即可返回

dispatch_enqueue

- 如果是 CPU 本地 dsq，则根据 container_of 宏获取包含该 dsq 字段的 rq 结构体的指针. 如果设置了 SCX_ENQ_PREEMPT 标志且 p 不是 rq 正在运行的进程且调度类选中了 sched_ext 类，则设置 slice 为 0，同时设置 preempt 标志为 True. sched_class_above 用于比较调度类的优先级，如果第一个拥有更高的优先级则返回 1，这种情况下也将启用 preempt 标志.
- preempt 标志被设置时，则将执行 resched_curr 原生内核函数. resched_curr 内核函数不会立刻进行进程切换，其只是简单的在当前 rq 的任务上设置 TIF_NEED_RESCHED 标志，调度器后续才会进一步调用. [6]

[6] https://stackoverflow.com/questions/75577661/resched-curr-in-does-not-reschedule-the-current-task-instantly

direct_dispatch

- 调用 dispatch_enqueue 将 p 插入到对应的 dsq 中
- 最后写 direct_dispatch_task 指针，写入一个非法的位置，表示直接分发已经发生过了. 与该函数在开头校验该指针是否合法相对应.

scx_bpf_dispatch

回到`scx_bpf_dispatch`中:

- 如果`ddsp_task`为空, 则先读一个 dispatch_buf_cursor 变量, 这是一个每 CPU 的 u32 类型, 指向当前 cpu 对应的 buffer idx. 相关的定义参考 dispatch_buf_ent.
- 对该 idx 做一些检查后, 访问对应的 buffer 内容, 并将当前任务和 dsq 写入该 buffer. 在写完 buffer 后会对 idx 自增, 当 buffer 表项用光后则报错, 该 buffer 即将用在非直接分发过程中.

scx_bpf_consume

该函数将从指定 dsq_id 的 dsq 中取任务到本地 dsq:

- cctx 是一个每 CPU 变量 struct consume_ctx, 其中有一个 struct rq 和一个 struct rq_flags
- 通过 dsq_id 以及 find_non_local_dsq 获取 dsq 队列

find_non_local_dsq

全局 DSQ 则直接返回，不然查 Hash 表返回 dsq_id 对应的 dsq.

scx_bpf_consume

- 通过 dsq_id 以及 find_non_local_dsq 获取 dsq 队列，如果 hash 查表失败则报错，不然进一步执行 consume_dispatch_q

consume_dispatch_q

该函数用于实际将任务插入到相应的运行队列中:

- 队列为空, 无法 consume, 返回 false
- 所有的 dsq_node 都是用一个双向链表连接起来的, dsq->fifo 应该存储了该链表的表头, 因而 list_for_each_entry 宏遍历 task_struct 的 scx.dsq_node 成员, scx 实际就是 sched_ext_entity 结构体类型; 随后一直循环访问到了 &dsq->fifo 为止结束, 暴露出 p 指针为每次遍历的结果, 此时的 p 正是该 dsq 中的某个任务的指针.
- 通过 task_rq 获取当前任务所对应的 rq, 如果该 rq 与 CPU 本地 dsq(传入的 rq 和 rf 都是 CPU 本地 dsq 结构内的内容) 所对应的 rq 一致则跳转到 this_rq 代码段进一步执行; 不然则做一些检查后进入 remote_rq 代码段.

consume_dispatch_q

- 对于 this_rq 而言，将 p 对应的节点移动到当前运行队列的 dsq 队列中，同时维护一些数据字段即可.
- 对于 remote_rq 而言，(显然一个现代的计算机必须考虑带有 SMP 配置的时候) 由于需要把一个非本 CPU 的任务 p 移动到本地 dsq 中，且 p 被 dsq 的锁所保护，因而争取该锁可能会导致死锁. SCX 使用了 move_task_to_local_dsq 函数进行处理，能够解决 SMP 结构下任务的分发，不过就不再继续深入了.

scx_bpf_consume

- 最后，检查一下moved布尔值，如果不真的话 jump 回 retry 代码段，重新执行上述流程.

scx_bpf_kick_cpu

该函数会唤醒一个闲置的 CPU 或者触发一个忙碌 CPU 重新进行调度. 其可以被任意执行中的 scx_ops 调用, 不过实际上 kick 并不立刻执行, 而是通过 irq 延迟执行:

- 函数内部首先检查 cpu 号是否合法, 随后关闭内核抢占, 获取当前 CPU 的 rq, 并将该 CPU 打上 rq 内部的 cpus_to_kick 标志.
- 再根据 flags 的内容为 CPU 打上几个其余的标志后, 将 cpumask_var_t 传给 irq 工作队列, 最后打开内核抢占即可.
  - 关于 kick_cpus_irq_work, 这是一个 struct irq_work 类型的成员, 大概长这样, 有关 irq_work 的也不多去关注了.

```
struct irq_work {
        struct __call_single_node node;
        void (*func)(struct irq_work *);
        struct rcuwait irqwait;
};
```

第 VI 部分

eBPF 内核支持

eBPF 内核支持

本章节将介绍一些用于 BPF 支持的相关代码.

bpf_struct_ops

99

```
struct bpf_struct_ops {
        const struct bpf_verifier_ops *verifier_ops;
        int (*init)(struct btf *btf);
        int (*check_member)(const struct btf_type *t,
                            const struct btf_member *member,
                            struct bpf_prog *prog);
        int (*init_member)(const struct btf_type *t,
                           const struct btf_member *member,
                           void *kdata, const void *udata);
        int (*reg)(void *kdata);
        void (*unreg)(void *kdata);
        const struct btf_type *type;
        const struct btf_type *value_type;
        const char *name;
        struct btf_func_model func_models[BPF_STRUCT_OPS_MAX_NR_MEMBERS];
        u32 type_id;
        u32 value_id;
};
```

bpf_scx_reg

该代码本质上调用了scx_ops_enable函数，这个函数才是真正做事的函数．下面我们进入该函数看一下．

```
static int bpf_scx_reg(void *kdata)
{
        return scx_ops_enable(kdata);
}
```

scx_ops_enable

- 首先检查一下 scx_ops_helper 结构是否为空，这是一个 struct kthread_worker 类型的东西，kthread_worker 是 Linux 内核中的一个机制，用于实现内核线程的创建和管理. 它是一个轻量级的工作队列，可以在内核中创建多个线程，以异步方式执行一些需要长时间运行的任务，而不会阻塞其他进程或线程.

```
struct kthread_worker {
        unsigned int            flags;
        raw_spinlock_t          lock;
        struct list_head        work_list;
        struct list_head        delayed_work_list;
        struct task_struct      *task;
        struct kthread_work     *current_work;
};

static struct kthread_worker *scx_ops_helper;
```

如果这是空的话，则调用 scx_create_rt_helper 函数创建一个该类型的元素，将对应的返回指针写进去.

scx_ops_enable

- 随后通过scx_ops_enable_state()读目前的运行状态，该枚举变量如果不是被禁用的状态则继续推进.
- 初始化哈希表，绑定 scx_ops 结构，该元素定义为:

```
static struct sched_ext_ops scx_ops;
```

- 初始化一些字段以及统计值.
- 随后调用用户自定义的 init 函数函数，判断一些返回值，并进行可能的错误处理.

scx_ops_enable

- 初始化 dispatch_buf 的 CPU 空间，如果用户规定了大小则使用不然则使用缺省的最大大小.
- 初始化 task_runnable_timeout_ms

```
/*
 * The maximum amount of time that a task may be runnable without being
 * scheduled on a CPU. If this timeout is exceeded, it will trigger
 * scx_ops_error().
 */
unsigned long task_runnable_timeout_ms;
```

- 初始化 last_timeout_check，其用于解决一些 irq 的问题

```
/*
 * The last time the delayed work was run. This delayed work relies on
 * ksoftirqd being able to run to service timer interrupts, so it's possible
 * that this work itself could get wedged. To account for this, we check that
 * it's not stalled in the timer tick, and trigger an error if it is.
 */
unsigned long last_timeout_check = INITIAL_JIFFIES;
```

scx_ops_enable

- 使用 percpu_down_write 操作每 CPU 信号量，该信号量是一个每 CPU 的读写锁，该函数的作用是获取一个该信号量的写锁. 下面这种语法不知道是定义在哪里的，但是根据对注释的分析我认为这种语法 (以下面的为例) 可能就是为 fork 系统调用加了个读写锁.

```
DEFINE_STATIC_PERCPU_RWSEM(scx_fork_rwsem);
```

通过获取这个写锁，就不会有新的进程被创建，后续的 prepare 等阶段就可以顺利进行

- scx_cgroup_lock 用于保护进程控制组 (cgroup)，cgroup(Control Group) 是 Linux 内核提供的一种机制，用于将进程分组并为它们分配资源限制. cgroup 可以控制一个或多个进程的资源使用情况，如 CPU 时间、内存、IO 等，从而实现对系统资源的管理和控制.

scx_ops_enable

遍历所有可能的函数指针位置，并且更新 scx_has_op 的内容，这是一个数组，其类型如下所示：

```
struct static_key_false scx_has_op[SCX_NR_ONLINE_OPS] =
        { [0 ... SCX_NR_ONLINE_OPS-1] = STATIC_KEY_FALSE_INIT };
```

有关该数组的类型，其实涉及到内核同步的相关知识. static_branch_enable 是一个宏，它用于在编译时确定是否创建静态分支. 静态分支是一种在运行时根据条件跳转到不同代码路径的技术. 启用静态分支可以提高代码执行效率，因为它可以避免在运行时进行条件判断，而带上 lock 则意味着要还要锁定该 CPU.

scx_ops_enable

- 随后根据 flags 标志位再额外启用/禁用一些函数指针.
- scx_tasks_lock 是一个自旋锁，用于保护 scx_tasks 队列的操作，该队列在 scx_task_iter_* 系列函数中被使用. 据注释所说，这是由于在退出阶段的任务可能会在失去自己 PID 的情况下被调度，所以当禁用 bpf 调度器时应该保证所有状态下的进程均被正常退出，所以用一个 task list 链起来.

```
/*
 * During exit, a task may schedule after losing its PIDs. When disabling the
 * BPF scheduler, we need to be able to iterate tasks in every state to
 * guarantee system safety. Maintain a dedicated task list which contains every
 * task between its fork and eventual free.
 */
static DEFINE_SPINLOCK(scx_tasks_lock);
static LIST_HEAD(scx_tasks);
```

scx_task_iter_*

下面介绍一下scx_task_iter_*系列的几个函数.

scx_task_iter

迭代器实体的基本结构如下:

```
struct scx_task_iter {
        struct sched_ext_entity         cursor;
        struct task_struct              *locked;
        struct rq                       *rq;
        struct rq_flags                 rf;
};
```

scx_task_iter_init

初始化函数:

- 调用时必须持有指定的锁, 可以通过 lockdep 机制检查
- cursor 指向一个 sched_ext_entity 类型的元素, 其 flags 字段为 SCX_TASK_CURSOR, 根据注释可知这表示其是一个遍历指针, 而不是一个任务.
- 添加链表元素, 链头是 ext.c 定义的一个元素 scx_tasks, 这是 `ext.c`文件定义的一个 static 的变量, 应该可以理解成一个 per-cpu 变量.
- 把 locked 指针置空, 初始化完成

scx_task_iter_exit

销毁一个之前初始化过的迭代器:

- 确保持有锁
- 如果持有运行队列的锁也需要释放一下，并将 locked 置空，可能是为了防止后续有人继续使用 iter 访问到该任务造成破坏.
- 判断链表是否空，如果不空则调用 list_del_init 从链表中删除该元素

```
/**
 * list_del_init - deletes entry from list and reinitialize it.
 * @entry: the element to delete from the list.
 */
static inline void list_del_init(struct list_head *entry)
```

scx_task_iter_next

遍历 task 链表:

- 确保持有锁
- 调用 list_for_each_entry, pos 是暴露出来的遍历指针, cursor 是链表头, tasks_node 是链表中 list_head 字段的名字.
- 如果 pos 访问到了 scx_tasks 了, 说明遍历到头了, 返回空
- 不然, 往后找第一个不是 CURSOR 标志的 task, 执行 list_move, 将 cursor 放到 &pos->tasks_node 后面
- 返回对应的 task_struct 结构
- 由于正常来说一定会终止, 所以加上 BUG()

_task_iter_next_filtered

返回后面的第一个非空闲的任务，封装了一层 scx_task_iter_next 即可.

ter_next_filtered_locked

1

113

返回后面的第一个非空闲的任务，不过要把其运行队列上锁后再返回，进一步封装 scx_task_iter_next_filtered

scx_ops_*_task

在接着看到 enable 函数之前，再看一组用于操作任务的初始化、启用、禁用的函数.

scx_ops_prepare_task

- 设置 disallow 字段为 false，允许该任务参与 scx 调度系统中
- 检查 sched_ext_ops 是否有自定义的 prep_enable 函数，有的话则调用之，判断一下返回值，并且认为失败是小概率事件
- 如果在 prep_enable 中启用了 disallow 标志，则不允许其参与 scx 调度系统，加锁 rq，将 policy 设置为 NORMAL，同时更新一些统计量
- 更新 p->scx 的 flags，表示程序已经准备好被 bpf 调度器 enable 了，同时指明 watchdog couter 应该被重置.

scx_ops_enable_task

- 使用 lockdep 验证对锁的持有，同时检查是否已经 prepare 好了
- 如果有自定义的 enable 则执行.
- 设置标志，清 SCX_TASK_OPS_PREPPED 标志，设置 SCX_TASK_OPS_ENABLED 表示已经 enabled 了

scx_ops_disable_task

- 使用 lockdep 验证对锁的持有
- 查看任务是只 prepare 过了还是已经 enable 了，分别处理：
  - 如果只 prepare 了，调用对应的关闭函数为 cancel_enable
  - 如果 enable 了，调用对应的禁止函数 disable

scx_ops_enable

下面回到scx_ops_enable函数中:

- scx_task_iter_init 初始化一个迭代器 sti, 随后不断调用. scx_task_iter_next_filtered 获取下一个非空闲的任务描述符 p. 对于每一个任务描述符来说:
  - 调用 get_task_struct, 其实该函数只是做个了引用计时器 +1, 目的是为了防止被认为无人使用而被释放掉

```
static inline struct task_struct *get_task_struct(struct task_struct *t)
{
        refcount_inc(&t->usage);
        return t;
}
```

  - 由于已经取到了任务描述符, 释放 &scx_tasks_lock 锁

scx_ops_enable

- 调用 scx_ops_prepare_task 初始化任务的一些状态标志，如果返回值不为 0，则说明装载用户自定义的函数后失败了，则：
  - put_task_struct，与 get_task_struct 对应，引用计数-1
  - 加锁后调用 scx_task_iter_exit，销毁该任务迭代器，jump 到错误处理代码中进一步收尾
- 如果成功，则释放当前进程的一些锁即可进入下一轮迭代.
- 在 prepare 阶段，所有的相关进程都已经准备就绪，但是还未开启 enable，所以当前的进程不会被调度出去. 同时由于 fork 写锁已被获取，因而保证不会有新的进程加入进来. 不过 scx 机制允许进程离开，这是因为 sched_ext_free 既可以处理 prepped 又可以处理 enable 状态的任务.

scx_ops_enable

- 第一轮循环完成后，所有符合条件的进程均完成 prepare，禁用抢占，防止当前进程由于调度的原因在还没完成工作的情况下被饿死
- 调用 scx_ops_tryset_enable_state 将所有的就绪状态转换为启用状态，随后全部认为任务的功能函数已经启用
- 使用 WRITE_ONCE 将 scx_switch_all_req 的布尔值写入到 scx_switching_all 中.

hed_deq_and_put_task

进入第二轮循环前，先看一个辅助函数sched_deq_and_put_task，该函数本质上封装了若干调度类的接口：

- 获取任务描述符的运行队列，使用 lockdep 机制进行锁的检查，将 ctx 指针的 4 个字段全部填充后传出去.
  - queue_flags 中或上 DEQUEUE_NOCLOCK 标志，其用于指示在没有锁定同步机制的情况下执行 dequeue 操作. 其在不需要同步保证的情况下允许进行快速操作，但必须确保并发线程安全性.
  - .queued 是一个布尔结果，该函数返回值表示给定的进程在运行队列中是否处于已排队状态 (即 TASK_ON_RQ_QUEUED). 如果返回值为 true，则意味着该任务已经在运行队列中，否则意味着该任务未被添加到运行队列中.

```
static inline int task_on_rq_queued(struct task_struct *p)
{
        return p->on_rq == TASK_ON_RQ_QUEUED;
}
```

sched_deq_and_put_task

- .running 也是一个布尔结果:

```
static inline int task_current(struct rq *rq, struct task_struct *p)
{
        return rq->curr == p;
}
```

- update_rq_clock 是 Linux 操作系统中的一个函数，它被用来更新系统运行队列的时钟.
- dequeue_task 和 put_prev_task 都是 static 的函数，本质是一些调度类接口函数的封装

dequeue_task

- 根据一些标志执行一些函数
- 随后执行调度类中的 dequeue_task 函数，实际上就是对这些调度类提供的接口函数做了一层封装而已

put_prev_task

类似的，做了一些封装，用于将进程加入到执行队列中

scx_ops_enable

进行第二轮循环，初始化一个迭代器，使用 scx_task_iter_next_filtered_locked 获取下一个非空闲的任务描述符 p，同时将其所处的运行队列上锁后再开始遍历任务描述符:

- 如果该任务是 DEAD 状态的任务，则直接调用 scx_ops_disable_task 禁用所有的任务即可
- 不然，调用 sched_deq_and_put_task：初始化 sched_enq_and_set_ctx 实体内容，并根据需求间接执行调度类接口的 dequeue_task 以及 put_prev_task 函数
- 使用 scx_ops_enable_task 启用 task 功能
- 使用 ___setscheduler_prio 修改进程的调度策略
- 使用 check_class_changing 检查是否发生了调度策略的变化，如果变化了，则还要使用调度类暴露的接口函数 switching_to
- 当上述对于进程的操作完成后，使用 sched_enq_and_set_task 再把该进程塞回去，该操作就是`sched_deq_and_put_task`的反面.

scx_ops_enable

事实上. 由于 sched_deq_and_put_task 函数将任务从原先的队列中取出, 并且该函数是在任务锁定的情况下执行的, 因此在这段时间内任务不会被调度.

接下来, sched_enq_and_set_task 函数将任务放入新的队列中, 由于这些操作都是在任务未被调度的情况下完成的, 因此只有当这些操作全部完成后, 才能够使任务重新进入调度器并重新开始调度.

用这种方式能够确保任务能够正确地被取出和重新插入到队列中, 并且能够避免并发操作引起的冲突和错误.

scx_ops_enable

- 执行 check_class_changed，进一步处理一些切换工作
- 本轮迭代完成后，该函数的大部分工作均完成，将一些锁释放后即可

bpf_scx_unreg

- 执行`scx_ops_disable`禁用所有的 ops
- 执行`kthread_flush_work(&scx_ops_disable_work)`刷新指定的工作队列并等待其中的所有工作项完成

第 VII 部分

scx_example_central

scx_example_central

最后，我们考察一个更加复杂的例子．在这里例子所呈现的调度策略中：

- 由一个 CPU 来决定所有的调度策略，当其余的 CPU 完成所有任务后必须通知中心 CPU 来分发任务．在该策略下，有一个全局的 BPF 队列，中心 CPU 通过 dispatch() 分发全局队列的任务都每个 CPU 的局部 dsq.

scx_example_central

最后，我们考察一个更加复杂的例子．在这里例子所呈现的调度策略中：

- 无时钟中断技术．
  - 无时钟中断是指在系统空闲时停止 CPU 的定时器中断，从而减少不必要的上下文切换和能耗，通过查看 /proc/interrupts 文件可以观察到无时钟中断的运行情况．
  - 为了防止无时钟中断过长时间不被调度，Linux 内核会周期性地检查所有 CPU 的状态，并根据需要进行抢占．这个周期性检查由一个定时器实现，不过由于 BPF 定时器目前没有办法绑定到特定的 CPU 上，所以周期性检查的定时器也不能被绑定到中央 CPU 上
- Preemption：内核进程 Kthreads 总是无条件地被排到指定本地 dsq 的头部，这保证了内核进程总是比用户进程拥有更高的优先级．比如，如果周期性时钟运行在 ksoftirqd 上，并且由于用户进程的运行使其发生了饥饿，那么就再也没有其他的内核进程可以来将这个用户进程腾出去 CPU 了．

实现细节

下面对用户态编写的 BPF 程序的一些细节进行分析，说明其如何利用之前所述的一些内核函数，实现该调度策略.

central_ops


```
SEC(".struct_ops")
struct sched_ext_ops central_ops = {
        /*
         * We are offloading all scheduling decisions to the central CPU and
         * thus being the last task on a given CPU doesn't mean anything
         * special. Enqueue the last tasks like any other tasks.
         */
        .flags                  = SCX_OPS_ENQ_LAST,

        .select_cpu             = (void *)central_select_cpu,
        .enqueue                = (void *)central_enqueue,
        .dispatch               = (void *)central_dispatch,
        .consume                = (void *)central_consume,
        .consume_final          = (void *)central_consume_final,
        .running                = (void *)central_running,
        .stopping               = (void *)central_stopping,
        .init                   = (void *)central_init,
        .exit                   = (void *)central_exit,
        .name                   = "central",
};
```

central_select_cpu

选择中心服务器，该选择可能并非是最终运行的 CPU；当该 CPU 不可用时内核也会自动进行 fallback.

central_enqueue

尽管按照 sched_ext_ops 注释所说，sched_ext_ops.enqueue 应当是将一个任务送入 BPF 调度器中，随后再通过 dispatch 的方式从 BPF 调度器中取任务分发到 dsq 中.

不过在 central 调度策略的实现中可以看到，enqueue 直接调用内核函数进行了分发，这样也是可以的

central_dispatch

- 首先通过 scx_bpf_dsq_nr_queued，接受参数 FALLBACK_DSQ_ID，该枚举常量定义为:

```
enum {
        FALLBACK_DSQ_ID         = 0,
        MAX_CPUS                = 4096,
        MS_TO_NS                = 1000LLU * 1000,
        TIMER_INTERVAL_NS       = 1 * MS_TO_NS,
};
```

scx_bpf_dsq_nr_queued

返回指定 dsq_id 中入队任务的数量，主要的 if-else 分支是用于处理不同种类的 dsq，代码内容比较简单：

- 如果查询本地 dsq 的话，则直接通过 rq 的内部数据结构逐渐访问到 scx_dispatch_q 结构体内容，获取其中的 nr 字段即可
- 不然，则进一步判断是不是一个带有 SCX_DSQ_LOCAL_ON 标志的 dsq_id，这表示这是某个其他 CPU 的本地 dsq，根据获取的 cpu 号读取该 cpu 下 rq 的对应数据结构.
- 最后，说明这不是一个 CPU 本地的 dsq 结构，则通过 find_non_local_dsq 获取 dsq 的指针，并读取其中的 nr 内容.

central_dispatch

- 再看一个 example 中自行实现的宏 MEMBER_VPTR. 其用于获取一个检查过的指针指向的结构体或数组成员. 编译器通常可能会打乱指令执行的顺序, 而该宏强制编译器产生一段先计算字节偏移量检查是否越界, 再实际计算偏移后的地址来产生成员的实际指针.

central_dispatch

- 其传入的 cpu_gimme_task 其实就是一个 bool 数组，每个 CPU 占一个元素．如果指针非空，则设置 gimme 内容为 true.

```
/* can't use percpu map due to bad lookups */
static bool cpu_gimme_task[MAX_CPUS];
```

- 检查将要分发的 CPU 是不是中心服务器，如果是的话调用 bpf_loop.

```
static long (*bpf_loop)(__u32 nr_loops, void *callback_fn, void *callback_ctx, __u64 flags) = (
        void *) 181;
```

bpf_loop

- 有关bpf_loop助手函数，该函数有四个参数：
  - nr_loops：给定函数应该循环执行的次数
  - callback_fn：指向一个静态函数的指针，该函数将在循环中执行
  - callback_ctx：指向上下文参数的指针，该参数将传递给回调函数
  - flags：一个当前未使用的参数
- 回调函数应该具有以下签名：

```
long (*callback_fn)(u32 index, void *ctx);
```

  - index 参数表示循环中的当前索引，ctx 是传递给 bpf_loop 函数的上下文参数.
  - 如果成功，bpf_loop 函数返回执行的循环次数，如果 flags 无效则返回 -einval，如果 nr_loops 超过最大循环次数则返回 -e2big.

tch_to_one_cpu_loopfn

- 通过 bpf_loop 循环执行若干次 dispatch_to_one_cpu_loopfn 代码，传入 NULL.
  - 该代码中，idx 为 bpf_loop 时的索引，即为 CPU 号.
  - 检查对应 CPU 是否有 gimme 标志，如果没有设置的话，对于该 CPU 的 loop 即可 return 0
  - 不然，则进一步使用 bpf_loop 循环调用 dispatch_a_task_loopfn，传入当前 CPU 的指针作为参数.

dispatch_a_task_loopfn

- 该函数首先通过 bpf_map_pop_elem 从 map 中弹一个元素出来，第一个参数为 map 的指针，第二个为存储了结果的指针，指针指向了在 central_enqueue 过程中加入到 bpf 映射表中的任务的 pid
- 通过 scx_bpf_find_task_by_pid 根据 pid 获取任务描述符 p，进而获取 cpus_ptr 对 cpu 标志进行测试．如果为 0，则分发到默认的 FALLBACK_DSQ_ID 里去．
- 不然则实际分发到对应的 CPU 本地 dsq 中．同时，如果该 CPU 不是中心 CPU，则应该手动 kick 一次敦促它进行一次调度．由于分发了任务，则可以将 gimme 清为 False．
- 不过，为什么要执行 1 « 23 次这么多次呢..

central_dispatch

回到 dispatch 方法中，如果目标不是中心服务器的则使用`scx_bpf_kick_cpu`唤醒一下中心服务器即可，让它为我们统一分发任务

central_consume


```
void BPF_STRUCT_OPS(central_consume, s32 cpu)
{
        /*
         * When preempted, we want the central CPU to always run dispatch() as
         * soon as possible so that it can schedule other CPUs. Don't consume
         * the fallback dsq if central.
         */
        if (cpu != central_cpu)
                scx_bpf_consume(FALLBACK_DSQ_ID);
}
```

central_consume_final


```
void BPF_STRUCT_OPS(central_consume_final, s32 cpu)
{
        /*
         * Now that the central CPU has dispatched, we can let it consume the
         * fallback dsq.
         */
        if (cpu == central_cpu)
                scx_bpf_consume(FALLBACK_DSQ_ID);
}
```

感谢观看！