第三届 eBPF开发者大会

www.ebpftravel.com

使用eBPF开发设备驱动的探索

麒麟软件 陈松

中国·西安

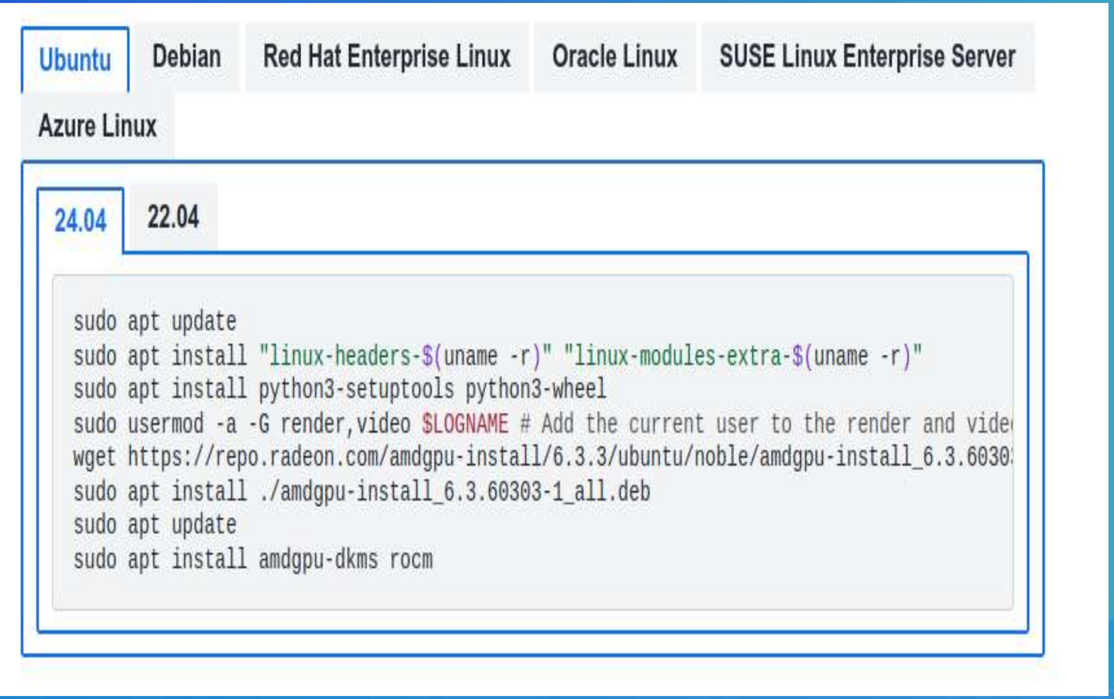
1，我遇到的问题

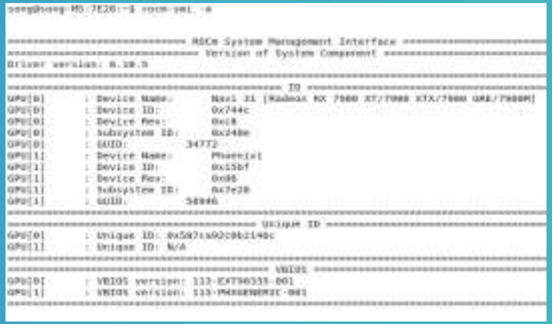
2，来自社区的灵感

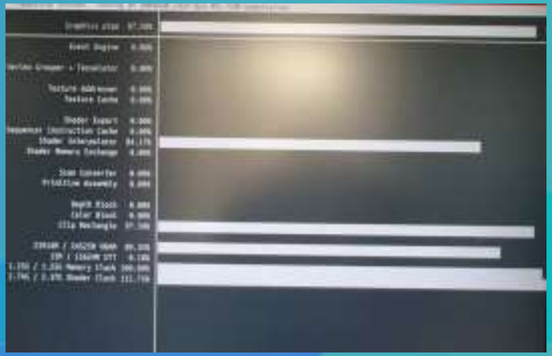
3，我的方案介绍

4，后续的思考

1，我遇到的问题







借到了一块
RX7900XTX显卡

按照AMD的官网指导安装了
rocm，其中包括amdgpu.ko

运转的不错
rocm-smi和radeontop都
显示了正确的信息

1，我遇到的问题

```
make install

Building module:
Cleaning build area...(bad exit status: 2)
. /tmp/amd.AsPn3XO1/.env && make -j16 KERNELRELEASE=6.14.0-rc1-llm-20250326+ TTM_NAME=amdttm SCHED_NAM
ERROR (dkms apport): kernel package linux-headers-6.14.0-rc1-llm-20250326+ is not supported
Error! Bad return status for module build on kernel: 6.14.0-rc1-llm-20250326+ (x86_64)
Consult /var/lib/dkms/amdgpu/6.10.5-2119913.24.04/build/make.log for more information.
dkms autoinstall on 6.14.0-rc1-llm-20250326+/x86_64 failed for amdgpu(10)
Error! One or more modules failed to install during autoinstall.
Refer to previous errors for more information.
 * dkms: autoinstall for kernel 6.14.0-rc1-llm-20250326+
run-parts: /etc/kernel/postinst.d/dkms exited with return code 11
make[1]: *** [arch/x86/Makefile:321: install] Error 11
make: *** [Makefile:251: __sub-make] Error 2


vi /var/lib/dkms/amdgpu/6.10.5-2119913.24.04/build/make.log

                 from ./amd/backport/backport.h:7,
                 from <command-line>:
/home/song/workspace/kernel/staging/include/linux/alloc_tag.h:239:2: error: expected identifier or
  239 | ({                                                                    \
      |  ^
/home/song/workspace/kernel/staging/include/linux/slab.h:1076:49: note: in expansion of macro 'all
 1076 | #define kvrealloc(...)                  alloc_hooks(kvrealloc_noprof(__VA_ARGS__))
      |                                         ^~~~~~~~~~~
./include/kcl/kcl_slab.h:42:14: note: in expansion of macro 'kvrealloc'
   42 | extern void *kvrealloc(const void *p, size_t oldsize, size_t newsize, gfp_t flags);
      |              ^~~~~~~~~
```

```
root@song-MS-7E28:/home/song# modprobe amdgpu
modprobe: ERROR: could not insert 'amdgpu': Invalid argument

[   82.167931] amdkcl: disagrees about version of symbol trace_raw_output_prep
[   82.167937] amdkcl: Unknown symbol trace_raw_output_prep (err -22)
[   82.167943] amdkcl: disagrees about version of symbol __trace_trigger_soft_d
[   82.167944] amdkcl: Unknown symbol __trace_trigger_soft_disabled (err -22)
[   82.167946] amdkcl: disagrees about version of symbol trace_event_printf
[   82.167947] amdkcl: Unknown symbol trace_event_printf (err -22)
[   82.167955] amdkcl: disagrees about version of symbol trace_event_raw_init
[   82.167955] amdkcl: Unknown symbol trace_event_raw_init (err -22)
[   82.167967] amdkcl: disagrees about version of symbol trace_event_buffer_com
[   82.167968] amdkcl: Unknown symbol trace_event_buffer_commit (err -22)
[   82.167979] amdkcl: disagrees about version of symbol dma_fence_describe
[   82.167980] amdkcl: Unknown symbol dma_fence_describe (err -22)
[   82.167988] amdkcl: disagrees about version of symbol perf_trace_run_bpf_sub
[   82.167989] amdkcl: Unknown symbol perf_trace_run_bpf_submit (err -22)
[   82.167997] amdkcl: disagrees about version of symbol trace_event_reg
[   82.167998] amdkcl: Unknown symbol trace_event_reg (err -22)
[   82.168006] amdkcl: disagrees about version of symbol bpf_trace_run1
[   82.168007] amdkcl: Unknown symbol bpf_trace_run1 (err -22)
[   82.168017] amdkcl: disagrees about version of symbol trace_event_buffer_res
[   82.168018] amdkcl: Unknown symbol trace_event_buffer_reserve (err -22)
[   82.168031] amdkcl: disagrees about version of symbol vmf_insert_mixed
[   82.168032] amdkcl: Unknown symbol vmf_insert_mixed (err -22)
```

编译安装一个自己的内核，dkms报错

调整了内核版本，modprobe仍然报错

1，我遇到的问题

驱动的兼容性问题：

- 结构的定义在不同版本中，会有元素的增加，删除或者重命名
- 函数的定义也会有参数的变化，重命名，或者消失
- 模块安装时由于vermagic不同导致的安装失败

```
[   39.516164] kylinfifo_array: disagrees about version of symbol module_layout
song@raspberrypi:~/workspace$ sudo insmod kylinfifo_array.ko
insmod: ERROR: could not insert module kylinfifo_array.ko: Invalid module format
```

```
song@raspberrypi:~/workspace$ modinfo kylinfifo_array.ko
filename:       /home/song/workspace/kylinfifo_array.ko
license:        GPL
srcversion:     B3D60F5F14169CF4E63B43E
depends:
name:           kylinfifo_array
vermagic:       6.1.21-v8-20241223+ SMP preempt mod_unload modversions aarch64
song@raspberrypi:~/workspace$ uname -r
6.1.21-v8-20240815+
```

driver/gpio/TODO

```
This is a place for planning the ongoing long-term work in the GPIO
subsystem.


GPIO descriptors

Starting with commit 79a9becda894 the GPIO subsystem embarked on a journey
to move away from the global GPIO numberspace and toward a descriptor-based
approach. This means that GPIO consumers, drivers and machine descriptions
ideally have no use or idea of the global GPIO numberspace that has/was
used in the inception of the GPIO subsystem.

The numberspace issue is the same as to why irq is moving away from irq
numbers to IRQ descriptors.

The underlying motivation for this is that the GPIO numberspace has become
unmanageable: machine board files tend to become full of macros trying to
establish the numberspace at compile-time, making it hard to add any numbers
in the middle (such as if you missed a pin on a chip) without the numberspace
breaking.

Machine descriptions such as device tree or ACPI does not have a concept of the
Linux GPIO number as those descriptions are external to the Linux kernel
and treat GPIO lines as abstract entities.

The runtime-assigned GPIO numberspace (what you get if you assign the GPIO
base as -1 in struct gpio_chip) has also became unpredictable due to factors
such as probe ordering and the introduction of -EPROBE_DEFER making probe
ordering of independent GPIO chips essentially unpredictable, as their base
number will be assigned on a first come first serve basis.

The best way to get out of the problem is to make the global GPIO numbers
unimportant by simply not using them. GPIO descriptors deal with this.

Work items:

- Convert all GPIO device drivers to only #include <linux/gpio/driver.h>

- Convert all consumer drivers to only #include <linux/gpio/consumer.h>

- Convert all machine descriptors in "boardfiles" to only
  #include <linux/gpio/machine.h>, the other option being to convert it
  to a machine description such as device tree, ACPI or fwnode that
  implicitly does not use global GPIO numbers.

- When this work is complete (will require some of the items in the
  following ongoing work as well) we can delete the old global
  numberspace accessors from <linux/gpio.h> and eventually delete
```

2，来自社区的灵感

https://www.kernel.org/doc/html/next/scheduler/sched-ext.html

https://www.phoronix.com/news/cpufreq_ext-RFC

Extensible Scheduler Class

sched_ext is a scheduler class whose behavior can be defined by a set of BPF programs - the BPF scheduler.

- sched_ext exports a full scheduling interface so that any scheduling algorithm can be implemented on top.
- The BPF scheduler can group CPUs however it sees fit and schedule them together, as tasks aren't tied to specific CPUs at the time of wakeup.
- The BPF scheduler can be turned on and off dynamically anytime.
- The system integrity is maintained no matter what the BPF scheduler does. The default scheduling behavior is restored anytime an error is detected, a runnable task stalls, or on invoking the SysRq key sequence *SysRq-S*.
- When the BPF scheduler triggers an error, debug information is dumped to aid debugging. The debug dump is passed to and printed out by the scheduler binary. The debug dump can also be accessed through the *sched_ext_dump* tracepoint. The SysRq key sequence *SysRq-D* triggers a debug dump. This doesn't terminate the BPF scheduler and can only be read through the tracepoint.

Switching to and from sched_ext

`CONFIG_SCHED_CLASS_EXT` is the config option to enable sched_ext and `tools/sched_ext` contains the example schedulers. The following config options should be enabled to use sched_ext:

```
CONFIG_BPF=y
CONFIG_SCHED_CLASS_EXT=y
CONFIG_BPF_SYSCALL=y
CONFIG_BPF_JIT=y
CONFIG_DEBUG_INFO_BTF=y
CONFIG_BPF_JIT_ALWAYS_ON=y
CONFIG_BPF_JIT_DEFAULT_ON=y
CONFIG_PAHOLE_HAS_SPLIT_BTF=y
CONFIG_PAHOLE_HAS_BTF_TAG=y
```

sched_ext is used only when the BPF scheduler is loaded and running.

Cpufreq_ext Being Worked On For BPF-Based CPU Frequency Scaling

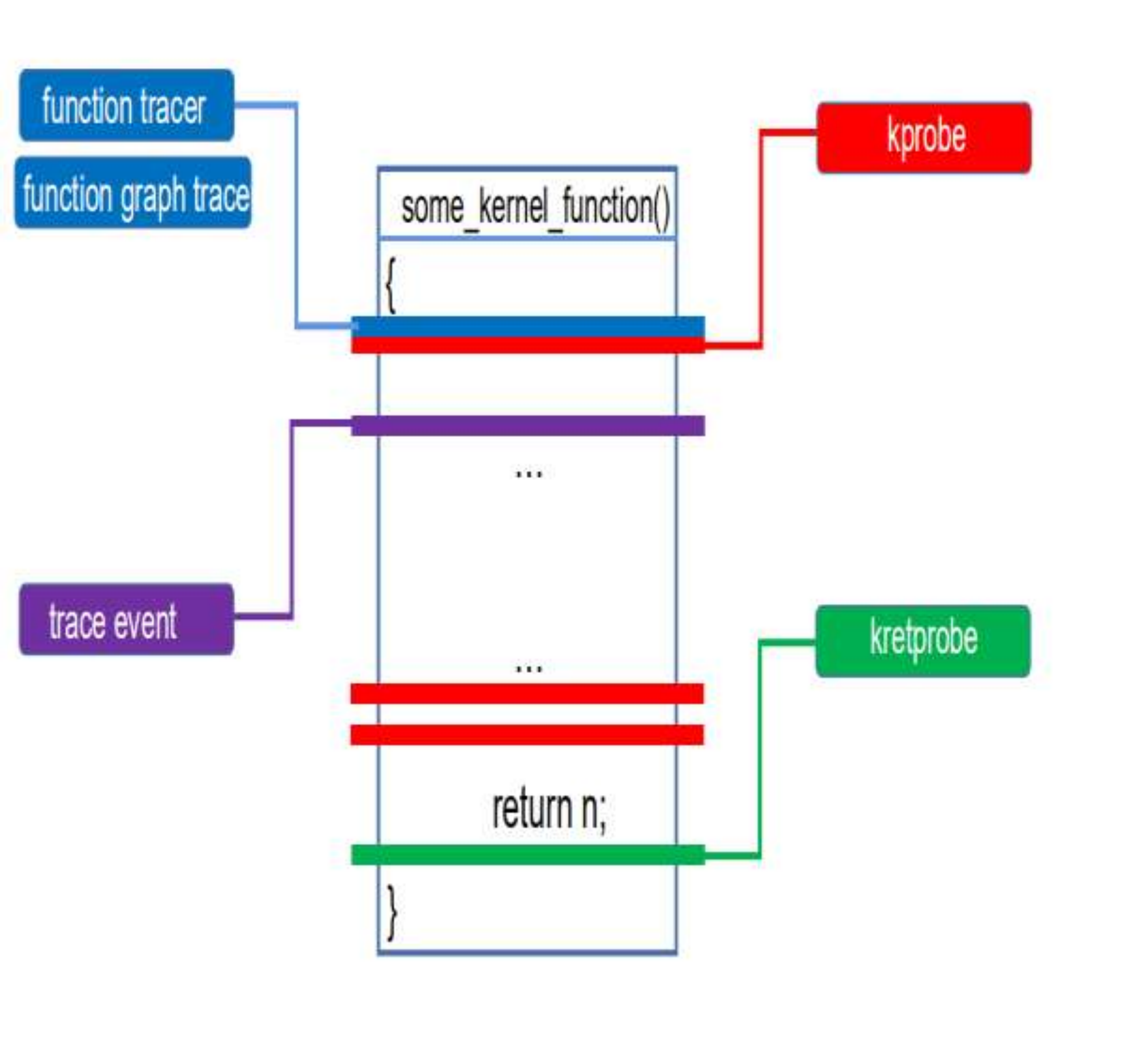
Written by Michael Larabel in Linux Kernel on 30 September 2024 at 07:49 AM EDT. 9 Comments

The newly-merged sched_ext allows for the Linux kernel scheduler to be made more extensible by allowing BPF programs to be loaded to affect the kernel's scheduling behavior. There's now a similar take on CPU frequency scaling: cpufreq_ext. There's a "request for comments" patch series on cpufreq_ext for making extensible CPU frequency scaling algorithm adaptations with BPF.

Yipeng Zou of Huawei has proposed cpufreq_ext as a CPU frequency governor based on BPF to allow for customizing and implementing different CPU frequency scaling strategies. Cpufreq_ext wants to provide a customizable framework for different systems and applications by providing greater CPUFreq control than other kernel-based options or the userspace governor that is less flexible. Cpufeq_ext can also integrate with sched_ext.

```
        --------------------------
        |      BPF governor      |
        --------------------------
                    |
                    v
               BPF Register
                    |
                    v
 ---------------------------------------
 |               CPUFreq ext           |
 ---------------------------------------
     ^              ^              ^
     |              |              |
 ---------      ---------      ---------
 | policy0 | ... | policy1 | ... | policyn |
 ---------      ---------      ---------
```

2，来自社区的灵感



挂载在函数的kprobe上

```c
SEC("kprobe/futex_wake")
int on_futex_wake(struct pt_regs *ctx)
{
    u32 pid = bpf_get_current_pid_tgid();
    u64 cur_ts = bpf_ktime_get_ns();
```

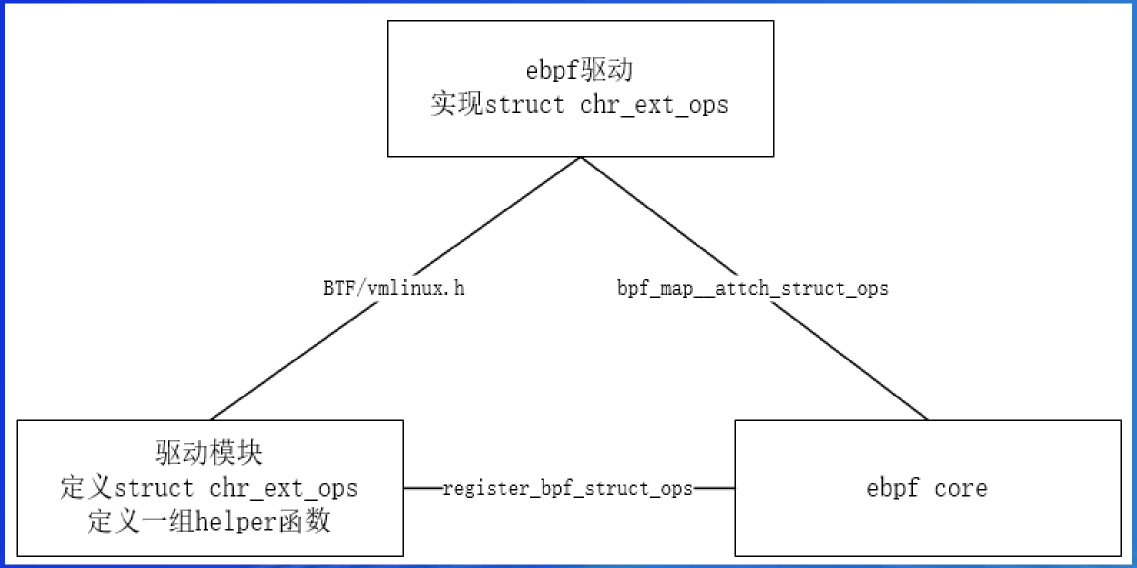
挂载在trace event上

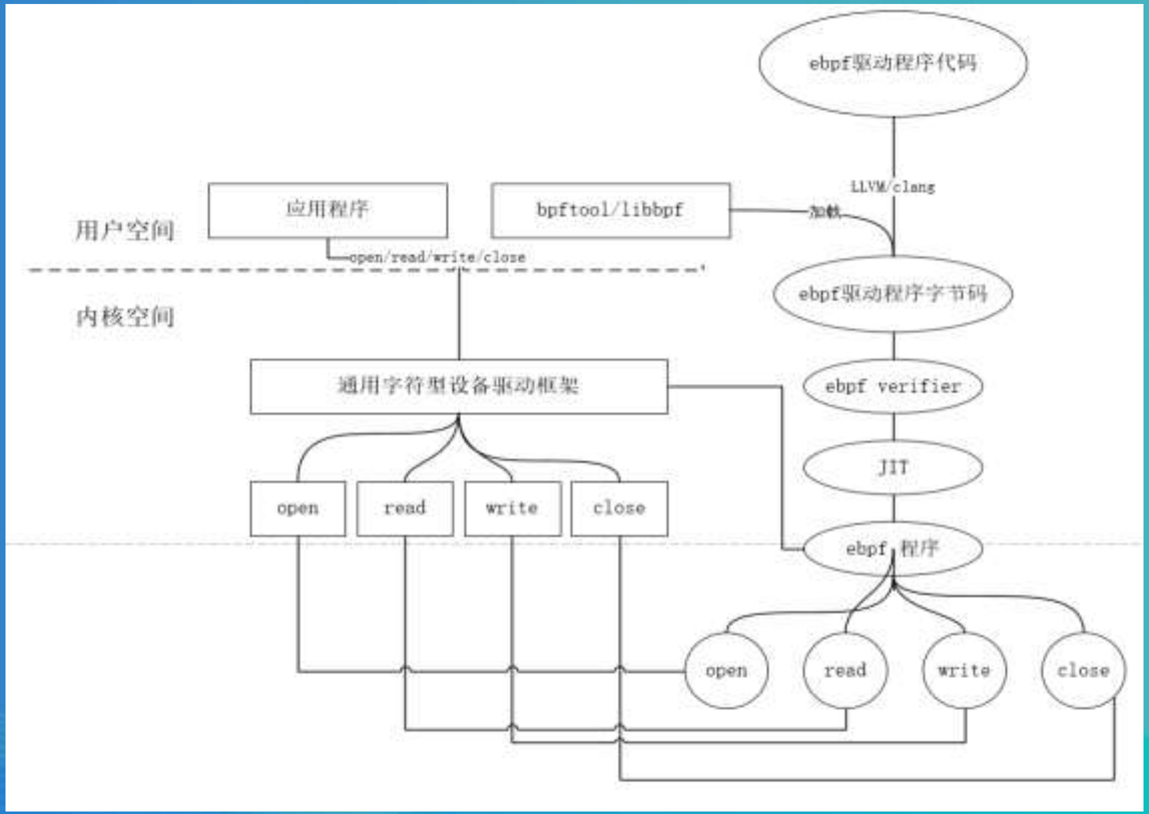
```c
SEC("tracepoint/sched/sched_waking")
int sched_waking(struct sched_waking_args *args)
{
    u32 pid = bpf_get_current_pid_tgid();
    u64 cur_ts = bpf_ktime_get_ns();
    u32 next_pid = args->next_pid;
```

通过struct_ops

```c
SEC("struct_ops.s/cust_read")
unsigned int BPF_PROG(cust_read_demo, char *buf)
{
    char local_buf[16] = "ebpf";
    bpf_printk("%s:%s, buf:%s\n", LOG_PREFIX, __FUNCTION__, buf);
    chr_ext_helper_demo((u8 *)local_buf, 4);
    return 0;
}
```

3，我的方案介绍





- 方案包含两部分：内核驱动的框架，eBPF驱动
- 在内核框架中定义一组回调函数，在eBPF程序中实现这组回调函数
- 以struct_ops为桥梁，将eBPF的实现注入到内核驱动模块中

3，我的方案介绍

1，
```
struct chr_ext_ops {
    unsigned long (*cust_read)(char *buf);
    unsigned int (*cust_write)(const char *buf);
    unsigned int (*cust_open)(void);
    void (*cust_close)(void);
    char name[CHR_EXT_NAME_LEN];
};
```
一组函数指针，在内核中定义，在ebpf代码中实现。
static struct chr_ext_ops cust_chr_ext_ops; 保存ebpf传递过来的函数指针，也就是ebpf驱动的行为逻辑。

2，
```
static struct bpf_struct_ops bpf_chr_ext_ops = {
    .verifier_ops = &ext_verifier_ops,
    .reg = ext_reg,
    .unreg = ext_unreg,
    .check_member = ext_check_member,
    .init_member = ext_init_member,
    .init = ext_init,
...
};
```
bpf_struct_ops是内核与ebpf程序的桥梁，帮助ebpf中实现的chr_ext_ops 传递到内核中。
ret = register_bpf_struct_ops(&bpf_chr_ext_ops, chr_ext_ops);在该模块初始化时注册到ebpf的内核管理器中

3，ebpf内核程序实现具体逻辑
```
void BPF_PROG(cust_open_demo)
void BPF_PROG(cust_open_demo)
unsigned int BPF_PROG(cust_write_demo, const char *buf)
unsigned int BPF_PROG(cust_read_demo, char *buf)
...
```
同时，将该些函数指针放到一个struct chr_ext_ops结构体中：
```
SEC(".struct_ops.link")
struct chr_ext_ops chr_ext_demo_ops = {
    .cust_open      = (void *)cust_open_demo,
    .cust_close = (void *)cust_close_demo,
    .cust_read          = (void *)cust_read_demo,
    .cust_write         = (void *)cust_write_demo,
    .name           = "cs"
};
```
最后将chr_ext_demo_ops加载到内核
```
 bpf_map__set_autoattach(skel->maps.chr_ext_demo_ops...
 bpf_map__attach_struct_ops(skel->maps.chr_ext_demo_ops);
```

4，内核响应，调用ext_reg，
将ebpf的chr_ext_ops指针赋值给内核的cust_chr_ext_ops
```
  static int ext_reg(void *kdata, struct bpf_link *link)
{
    struct chr_ext_ops *ops = (struct chr_ext_ops *)kdata;
    cust_chr_ext_ops = *ops;
...
}
```

5，内核调用（以read为例）
```
static ssize_t chr_ext_read(...)
{
        if (cust_chr_ext_ops.cust_write)

cust_chr_ext_ops.cust_write(device_buffer);
    ......
}
```

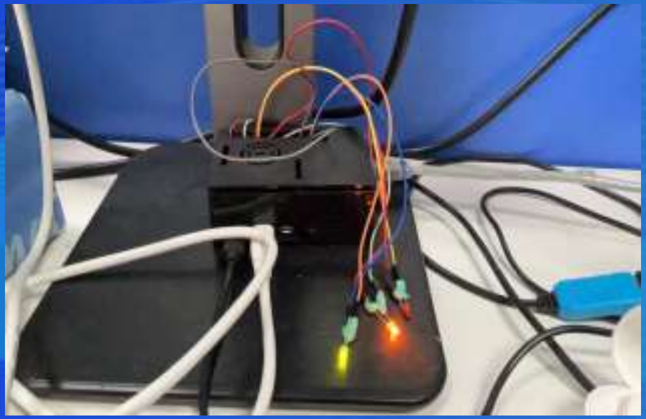
3，我的方案介绍

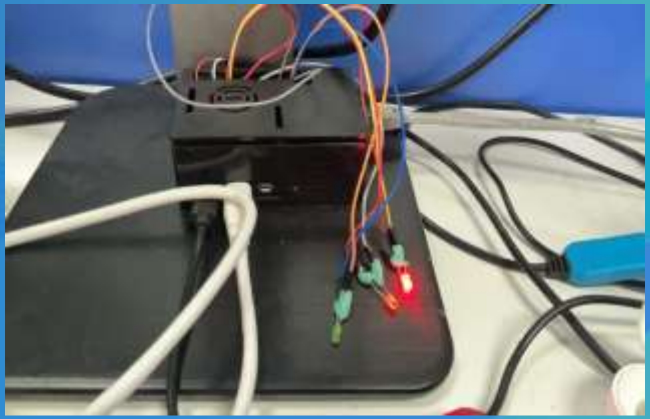
```
SEC("struct_ops.s/cust_write")
unsigned int BPF_PROG(cust_write_demo,
const char *buf)
{
    bpf_printk("%s:%s, buf:%s\n", LOG_PREFIX,
__FUNCTION__, buf);
    chr_ext_helper_set_gpio_value(0x05);

    return 0;
}
```

```
__bpf_kfunc void chr_ext_helper_set_gpio_value(unsigned long val)
{
...
    ret = gpiod_set_array_value(buck_gpios->ndescs, buck_gpios->desc,
buck_gpios->info, bitmap);
...
    return;
}

BTF_KFUNCS_START(ext_bpf_helpers)
BTF_ID_FLAGS(func, chr_ext_helper_set_gpio_value)
BTF_KFUNCS_END(ext_bpf_helpers)
```





4，后续的思考

- 可移植性的收益
  - 跨内核版本的可移植性
  - 跨平台的可移植性

- 适用场景：
  - 国产设备驱动 （jjw）
  - 第三方定制产品外设驱动（fpga）
  - 非开源代码的驱动（rocm,cuda）

- 使用eBPF编写驱动的思考：
  - 当前采用eBPF技术的多是策略型的模块，比如cpufreq，scheduler， oom killer
  - 通用性的问题，是否能实现子系统的通用框架
  - 子系统通用逻辑和定制逻辑的分离
  - 增加关于硬件操作的helper函数，如gpio，pcie，spi等
  - helper函数的符号版本化

4，后续的思考

- 兼容性治理的思考：
  - 内核模块的兼容性使用dkms，kabi检测，weak-modules等
  - xenomai使用LINUX_VERSION_CODE/KERNEL_VERSION解决内核版本间的差异
  - glibc使用符号版本化
  - eBPF的CO-RE也是着力于解决兼容性的问题
  - eBPF技术正在发挥越来越大的作用，我们能否将其延伸到驱动程序定制化的领域

谢谢大家