`

西安郵電大學

1. 项目简介与基本情况

1.1 概述

本次的项目名称是Linux 内核实时性瓶颈分析工具，由于在工业控制、机器人控制领域中越来越多使用Linux操作系统，但Linux系统在实时性方面天然不具备优势。为了改善Linux实时性，参考eBPF或perf等相关技术原理，研发出一个探针型的工具用于分析造成中断较高的原因，便于内核程序员对症下药。通过解决有限高延迟路径，从而达到让Linux在多种高负载场景下仍然能够长期保持可接受范围内的延迟。

1.2 项目成员

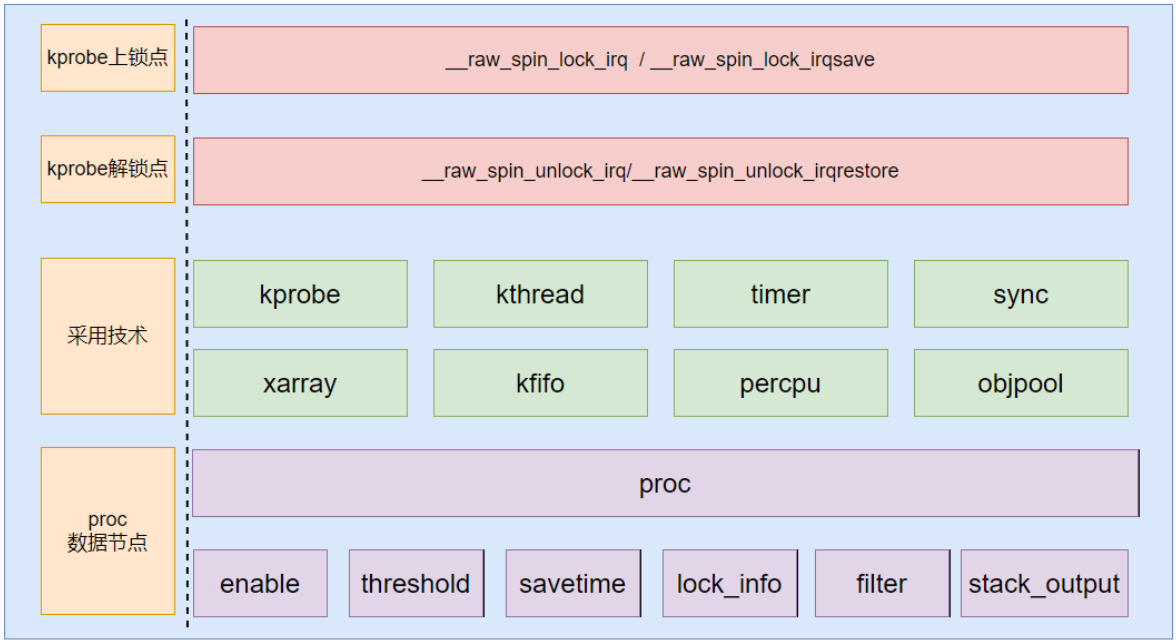
指导老师：郭浩

项目成员：

姓名	年级	专业
张玉哲	研二	软件工程
杨骏青	研二	电子信息
石泉	研一	电子信息

1.3 项目架构

项目的整体架构图如下所示：



1.4 仓库目录结构

```
project788067-126085/
./
├── docs 文档
|   ├── design_docs
|   ├── irq
|   └── env.md
├── include 头文件
|   ├── data.h
|   ├── kfifo.h
|   ├── kprobe.h
|   ├── kthread.h
|   ├── lib.h
|   ├── objpool.h
|   ├── percpu.h
|   ├── proc.h
|   ├── workqueue.h
|   └── xarray.h
├── src 子模块实现
|   ├── data.c
|   ├── kfifo.c
|   ├── kprobe.c
|   ├── kthread.c
|   ├── objpool.c
|   ├── percpu.c
|   ├── proc.c
|   ├── workqueue.c
|   └── xarray.c
├── main.c
├── Makefile
└── README.md
```

1.5 项目开发进展

题目编号	基本完成情况	说明
第一题：内核模块基础框架实现	已实现100%	已全部完成
第二题：探针工具实际内容实现	已实现100%	已全部完成
第三题：工具稳定性验证和实际效果测试	已实现100%	已全部完成并在树莓派上成功测试

第一题：内核模块基础框架实现 (已实现100%)

- [x] shell脚本能够使用自研内核模块的procfs机制与自研内核进行字符串读写
- [x] 自研内核模块内部需要对用户态输入的数据进行分析，并使用链表对数据进行格式化存储
- [x] 自研内核模块内部使用cache机制对格式化存储数据进行存储
- [x] 该工具需要支持开机自启动，读取指定配置文件下发到自研内核模块中

第二题：探针工具实际内容实现 (已实现100%)

以题目一为基础，工具需要追加下列功能：

- [x] 自研内核模块增加硬件中断号参数、阈值参数、模块开关，可通过shell工具进行配置和修改
- [x] 自研模块能够根据shell配置的硬件中断号对指定中断或全部中断的关闭中断时长进行检测，精度需要达到纳秒级别
- [x] 自研内核模块能够根据shell配置的阈值参数进行数据过滤，只有关闭时长大于该阈值时才会触发数据抓取操作，抓取内容包括使用该中断的进程相关信息，如调用栈、持有锁、文件、socket等敏感信息
- [x] 抓取后的数据需要使用cache机制存储到内核链表中。shell工具可以通过procfs读取所有抓取到的数据，也可以清空所有抓取到的数据。

第三题：工具稳定性验证和实际效果测试 (已实现100%)

以题目二为基础，工具需要追加下列功能：

- [x] 能够在内核态正常长时间运行，不会造成内存泄漏和系统卡顿
- [x] 需要在高负载场景进行工具的功能性和稳定性测试，包括CPU型高负载、内存型高负载、IO型高负载、中断型高负载、综合型高负载

2. 系统需求分析与设计

2.1 procfs分析与设计

2.1.1 proc概述

题目中需要完成对profs的交互，proc是Linux 内核提供的一种虚拟文件系统，具有在运行时访问内核内部数据结构、改变内核设置的机制。

它只存在内存当中，而不占用外存即磁盘空间。它以文件系统的方式为访问系统内核数据的操作提供接口。

用户和应用程序可以通过`proc`得到系统的信息，并可以改变内核的某些参数。

由于系统的信息，如进程，是动态改变的，所以用户或应用程序读取`proc`文件时，`proc`文件系统是动态从系统内核读出所需信息并提交的。

PS：但是proc下的文件或子文件夹，并不是全部都在你的系统中存在，这取决于你的内核配置和装载的模块。

另外，在`/proc`下还有三个很重要的目录：`net`，`scsi`和`sys`。

- `sys`目录是可写的，可以通过它来访问或修改内核的参数
- `net`和`scsi`则依赖于内核配置。例如，如果系统不支持`scsi`，则`scsi`目录不存在。

除了以上介绍的这些，还有的是一些以数字命名的目录，它们是进程目录。

系统中当前运行的每一个进程都有对应的一个目录在`/proc`下，以进程的`PID`号为目录名，它们是读取进程信息的接口。

而self目录则是读取进程本身的信息接口，是一个`link`。

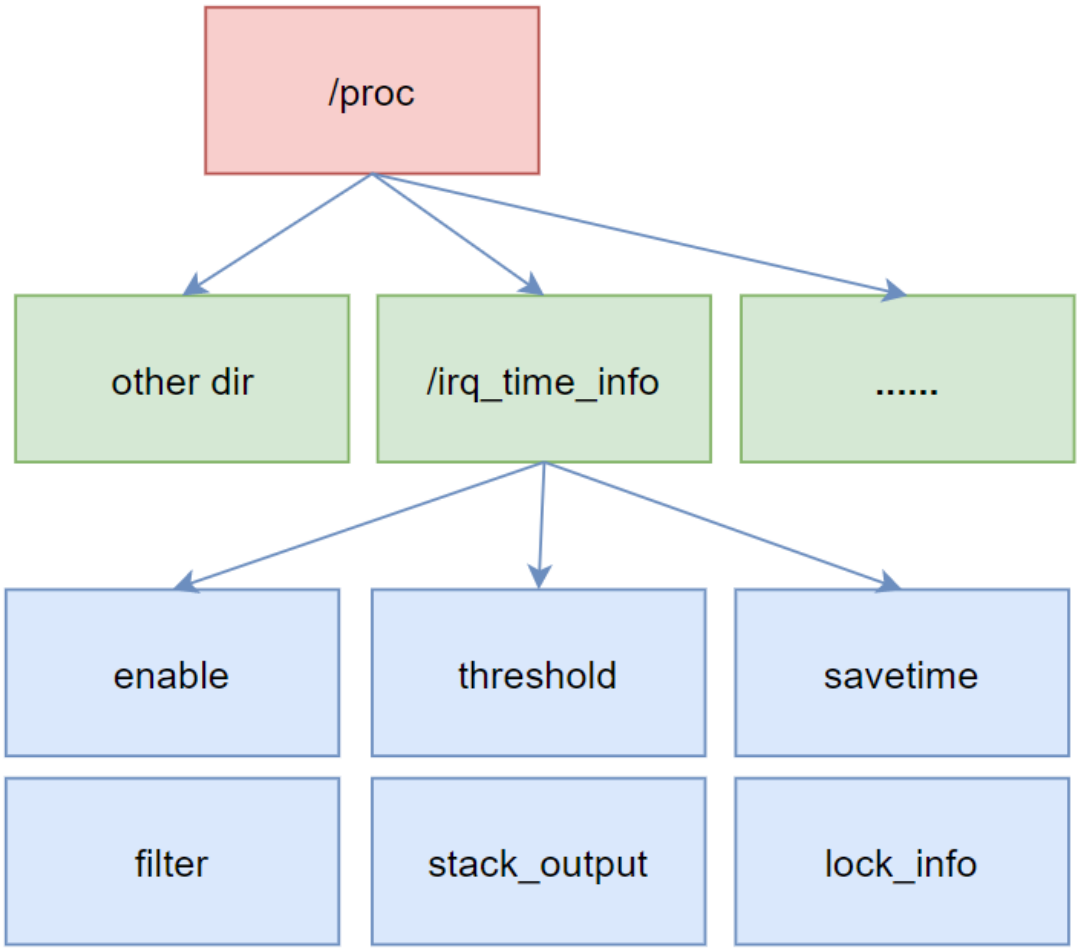
事实上，`ps`、`top`等命令就是直接从proc文件系统中读取信息的。

2.1.2 节点设计

根据题目要求，我们本次所要实现的大概是以下四个方面：

- shell脚本能够使用自研内核模块的procfs机制与自研内核进行字符串读写
- 中断号参数、阈值参数、模块开关
- 能够根据shell配置的阈值参数进行数据过滤
- 通过procfs读取所有抓取到的数据

所以在procfs下的结构如图所示：



其中:

- enable 为模块全局开关，0代表关闭监测，1代表开启监测；
- threshold 为阈值参数，只有关中断的时间大于这个参数才会被记录；

- savetime 为采集到的信息被保存的时间，可以通过指定该参数来调整信息的保存时间长度；
- lock_info 为抓取到的数据的大概展示，可以通过 cat 该节点来得到所有监测的结果；
- filter为进展示锁的调用栈信息的参数，可以通过指定该参数来调整要展示的具体进程争用具体锁的调用栈信息；
- stack_output：为filter被设置后具体进程争用具体锁的调用栈信息，可以通过cat该节点来获得结果。

2.2 kprobes 机制

`Kprobes` 是内核开发者们专门为了便于跟踪内核函数执行状态所设计的一种轻量级内核调试技术。

开发人员在内核或者模块的调试过程中，往往会需要要知道其中的一些函数有无被调用、何时被调用、执行是否正确以及函数的入参和返回值是什么等等。

内核开发人员利用`kprobes`技术，用户可以定义自己的回调函数，然后在内核或者模块中几乎所有的函数中动态的插入探测点，当内核执行流程执行到指定的探测函数时，会调用该回调函数，用户即可收集所需的信息了，同时内核最后还会回到原本的正常执行流程。如果用户已经收集足够的信息，不再需要继续探测，则同样可以动态的移除探测点。

> PS：有些函数是不可探测的，例如kprobes自身的相关实现函数

因此kprobes技术具有对内核执行流程影响小和操作方便的优点。

kprobes技术现在主要包括的2种探测手段分别：

- kprobe
- kretprobe

2.2.1 kprobe

首先我们先来看一下`kprobe`的结构体：

```
1  struct kprobe {
2      struct hlist_node hlist;
3      struct list_head list; /* list of kprobes for multi-handler support */
4      unsigned long nmissed; /*count the number of times this probe was
   temporarily disarmed */
5      kprobe_opcode_t *addr; /* location of the probe point */
6      const char *symbol_name; /* Allow user to indicate symbol name of the
   probe point */
7      unsigned int offset; /* Offset into the symbol */
8      kprobe_pre_handler_t pre_handler; /* Called before addr is executed. */
9      kprobe_post_handler_t post_handler; /* Called after addr is executed,
   unless... */
10     kprobe_fault_handler_t fault_handler; /*called if executing addr causes
   a fault*/
11     kprobe_break_handler_t break_handler; /*called if breakpoint trap occurs
   in probe handler.*/
12     kprobe_opcode_t opcode; /* Saved opcode (which has been replaced with
   breakpoint) */
13     struct arch_specific_insn ainsn; /* copy of the original instruction */
14     u32 flags; /*Indicates various status flags.*/
15 }
```

其实在内核模块中注册探针后，Kprobes 会复制一份被探测的指令，并用断点指令（例如 i386 和 x86_64 上的`int3`）替换被探测指令的第一个字节。

> 会把所设断点地址处的第一个字节改为0xCC，并把原字节保存

当 CPU 遇到断点指令时，会发生陷阱，保存 CPU 的寄存器，并通过 `notifier_call_chain` 机制将控制权传递给 Kprobes。

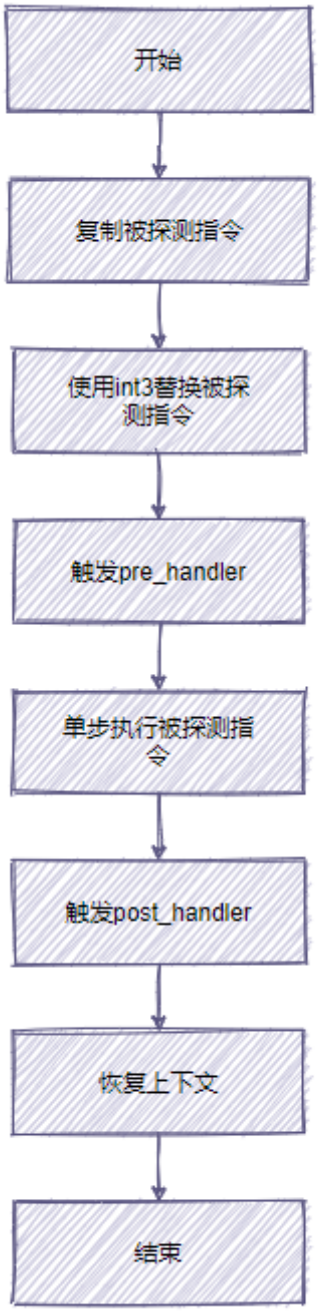
> Linux内核中各个子系统相互依赖，当其中某个子系统状态发生改变时，就必须使用一定的机制告知使用其服务的其他子系统，以便其他子系统采取相应的措施。为满足这样的需求，内核实现了事件通知链机制（notifier_call_chain）

Kprobes 执行与探针关联的`pre_handler`，向处理程序传递 kprobe 结构体的地址addr和保存的寄存器信息。

接下来，Kprobes 单步执行刚才复制的被探测指令。其实单步执行实际指令会更简单，但这样的话 Kprobes 将不得不暂时删除断点指令。

由于 kprobes 可以探测正在运行的内核代码，它可以更改寄存器组，包括指令指针。所以进行此操作时需要非常小心，例如保留堆栈帧、恢复执行路径等。

在指令单步执行后，Kprobes 执行与探针相关联的`post_handler`（如果有的话）。然后继续执行探测点之后的其他指令，大致流程如下：



2.2.2 kretprobe

下面是kretprobe的结构体：

```
struct kretprobe {
    struct kprobe kp;
    kretprobe_handler_t handler;
    kretprobe_handler_t entry_handler;
    int maxactive;
    int nmissed;
    size_t data_size;
    struct hlist_head free_instances;
    raw_spinlock_t lock;
};
```

当您调用 `register_kretprobe()` 时，Kprobes 会在函数的入口处建立一个探针。

当被探测的函数被调用并且这个探针被命中时，Kprobes 会保存一份返回地址的副本，并将返回地址替换为`trampoline`的地址。

> trampoline 可以理解为跳板，经过trampoline，最终可以跳转到目标代码。trampoline 是一段任意代码，通常只是一条 nop 指令。

在启动时，Kprobes 在`trampoline`上注册一个探针。当被探测的函数执行return时，不是返回到原来的地址而是将控制权传递给`trampoline`。

Kprobes 的 `trampoline` 处理程序会调用与 `kretprobe` 关联的返回处理程序，然后将指令指针改为之前保存的返回地址，这就是从陷阱返回后恢复执行的地方。

> kretprobe的目标是在一个函数返回的时候，执行用户指定的指令。从上文中可以得到，kretprobe在函数开头插入的探针会将函数的原返回地址保存，用一个“trampoline"来作为返回地址，并在"trampoline"上注册一个探针。当函数执行到返回地址时，“trampoline"得到控制权，触发了trampoline上的探针，在trampoline的探针中执行用户指定的程序，执行完成以后，恢复函数返回值至原返回地址。

当被探测函数正在执行时，它的返回地址存储在一个 `kretprobe_instance` 类型的对象中。在调用 `register_kretprobe()` 之前，用户设置 `kretprobe` 结构的 `maxactive` 字段来限制对指定函数可以同时探测多少实例。`register_kretprobe()` 预分配指定数量的 `kretprobe_instance` 对象。

2.3 中断挂载点分析

由题目要求可知，主要是根据硬件中断号对指定中断或全部中断的关闭中断时长进行检测，而关中断分为关闭`local cpu`中断和关闭中断线

2.3.1 关闭 local cpu

其中，关闭local cpu主要通过以下四个接口实现：

- local_irq_disable() 关闭本地cpu中断
- local_irq_enable() 打开本地cpu中断
- local_irq_save() 保存当前中断状态并关闭中断
- local_irq_restore() 恢复中断

上面所述的开关中断函数 `local_irq_*` 都是通过 `raw_local_irq_*` 实现的:

```
//include/linux/irqflags.h
#define local_irq_enable()  do { raw_local_irq_enable(); } while (0)
#define local_irq_disable() do { raw_local_irq_disable(); } while (0)
#define local_irq_save(flags)   do { raw_local_irq_save(flags); } while (0)
#define local_irq_restore(flags) do { raw_local_irq_restore(flags); } while
(0)
```

然后对于不同的体系架构有不同的实现方式：

```
//include/linux/irqflags.h
#define raw_local_irq_disable()     arch_local_irq_disable()
#define raw_local_irq_enable()      arch_local_irq_enable()
#define raw_local_irq_save(flags)
    do {
        typecheck(unsigned long, flags);
        flags = arch_local_irq_save();
    } while (0)
#define raw_local_irq_restore(flags)
    do {
        typecheck(unsigned long, flags);
        raw_check_bogus_irq_restore();
        arch_local_irq_restore(flags);
    } while (0)
```

由于我们的实验环境是在ARM64上，所以对于ARM64来说：

`arch_local_irq_disable` 的实现方式如下：

```
//arch/arm64/include/asm/irqflags.h
static inline void arch_local_irq_disable(void)
{
    if (system_has_prio_mask_debugging()) {
        u32 pmr = read_sysreg_s(SYS_ICC_PMR_EL1);

        WARN_ON_ONCE(pmr != GIC_PRIO_IRQON && pmr != GIC_PRIO_IRQOFF);
    }

    asm volatile(ALTERNATIVE(
        "msr    daifset, #3     // arch_local_irq_disable",
        __msr_s(SYS_ICC_PMR_EL1, "%0"),
        ARM64_HAS_IRQ_PRIO_MASKING)
        :
        : "r" ((unsigned long) GIC_PRIO_IRQOFF)
        : "memory");
}
```

`arch_local_irq_enable` 的实现方式如下：

```
//arch/arm64/include/asm/irqflags.h
static inline void arch_local_irq_enable(void)
{
    if (system_has_prio_mask_debugging()) {
        u32 pmr = read_sysreg_s(SYS_ICC_PMR_EL1);

        WARN_ON_ONCE(pmr != GIC_PRIO_IRQON && pmr != GIC_PRIO_IRQOFF);
```

```
    }

    asm volatile(ALTERNATIVE(
        "msr    daifclr, #3     // arch_local_irq_enable",
        __msr_s(SYS_ICC_PMR_EL1, "%0"),
        ARM64_HAS_IRQ_PRIO_MASKING)
        :
        : "r" ((unsigned long) GIC_PRIO_IRQON)
        : "memory");

    pmr_sync();
}
```

`arch_local_irq_save` 的实现方式如下：

```
static inline unsigned long arch_local_irq_save(void)
{
    unsigned long flags;

    flags = arch_local_save_flags();

    /*
     * There are too many states with IRQs disabled, just keep the current
     * state if interrupts are already disabled/masked.
     */
    if (!arch_irqs_disabled_flags(flags))
        arch_local_irq_disable();

    return flags;
}
```

`arch_local_irq_restore` 的实现方式如下：

```
static inline void arch_local_irq_restore(unsigned long flags)
{
    asm volatile(ALTERNATIVE(
        "msr    daif, %0",
        __msr_s(SYS_ICC_PMR_EL1, "%0"),
        ARM64_HAS_IRQ_PRIO_MASKING)
        :
        : "r" (flags)
        : "memory");

    pmr_sync();
}
```

所以综上所述，我们可以通过挂载 `raw_local_irq_*` 函数，来捕捉到其关中断和开中断的行为，其中的差值即为关中断的时间。

2.3.2 关闭中断线

而对于只关闭中断线来说也有若干个接口可供使用，一般是用在驱动开发程序中：

- disable_irq 关闭中断并等待中断处理完后返回, 在非中断处理函数中使用，会阻塞
- disable_irq_nosync 立即返回,在中断处理函数中使用，不会阻塞；用于屏蔽相应中断
- enable_irq 激活中断线

- synchronize_irq 等待一个特定的中断处理程序执行完毕

disable_irq

其中 `disable_irq` 的调用关系如下，其实就是调用了 `synchronize_irq()`：

```
disable_irq
    -->synchronize_irq
```

主要函数实现如下：

```
//kernel/irq/manage.c
void disable_irq(unsigned int irq)
{
    if (!__disable_irq_nosync(irq))
        synchronize_irq(irq);
}
EXPORT_SYMBOL(disable_irq);
```

disable_irq_nosync

其中 `disable_irq_nosync` 是比较常用的关闭中断线的方法，函数调用关系如下：

```
disable_irq_nosync
    -->__disable_irq_nosync
        -->__disable_irq
            -->irq_disable
                -->__irq_disable
                    -->mask_irq
```

主要函数实现如下：

```
//kernel/irq/manage.c
void disable_irq_nosync(unsigned int irq)
{
    __disable_irq_nosync(irq);
}
EXPORT_SYMBOL(disable_irq_nosync);
-----------------------------------------------------
static int __disable_irq_nosync(unsigned int irq)
{
    unsigned long flags;
    struct irq_desc *desc = irq_get_desc_buslock(irq, &flags,
IRQ_GET_DESC_CHECK_GLOBAL);

    if (!desc)
        return -EINVAL;
    __disable_irq(desc);
    irq_put_desc_busunlock(desc, flags);
    return 0;
}
```

enable_irq

其中 `enable_irq` 主要是在关闭中断线，做完相关操作之后用来激活中断线的，函数调用关系如下：

```
enable_irq
    -->__enable_irq
        -->irq_startup
            -->irq_enable
                -->unmask_irq
```

在这里调用 `irq_startup()` 而不是 `irq_enable()` 因为中断可能被标记为 `NOAUTOEN`。

所以`irq_startup()` 需要在第一次启用时被调用。如果它已经启动，那么`irq_startup()` 将在后台调用 `irq_enable()`。

具体函数实现如下：

```
//kernel/irq/manage.c
void enable_irq(unsigned int irq)
{
    unsigned long flags;
    struct irq_desc *desc = irq_get_desc_buslock(irq, &flags,
IRQ_GET_DESC_CHECK_GLOBAL);

    if (!desc)
        return;
    if (WARN(!desc->irq_data.chip,
         KERN_ERR "enable_irq before setup/request_irq: irq %u\n", irq))
        goto out;

    __enable_irq(desc);
out:
    irq_put_desc_busunlock(desc, flags);
}
EXPORT_SYMBOL(enable_irq);
```

synchronize_irq

`synchronize_irq` 主要是用来等待一个特定的中断处理程序执行完毕，会阻塞，其函数调用关系如下：

```
synchronize_irq
    -->__synchronize_hardirq
        -->raw_spin_lock_irqsave
          |irqd_irq_inprogress(&desc->irq_data);|
        -->raw_spin_unlock_irqrestore
```

具体函数实现如下：

```
//kernel/irq/manage.c
void synchronize_irq(unsigned int irq)
{
    struct irq_desc *desc = irq_to_desc(irq);

    if (desc) {
        __synchronize_hardirq(desc, true);
        /*
         * We made sure that no hardirq handler is
         * running. Now verify that no threaded handlers are
         * active.
```

```
12          */
13         wait_event(desc->wait_for_threads,
14                !atomic_read(&desc->threads_active));
15     }
16 }
17 EXPORT_SYMBOL(synchronize_irq);
```

所以综上所述，我们可以通过监测三对接口函数来实对系统关中断时间的获取，分别是：

序号	关中断	开中断
1	raw_spin_lock_irq()	raw_spin_unlock_irq()
2	raw_spin_lock_irqsave()	raw_spin_lock_irqrestore()
3	mask_irq()	unmask_irq()

2.4 调用栈获取

2.4.1 dump_stack

一般情况下，当内核出现比较严重的错误时，例如发生oops错误或者内核认为系统运行状态异常，内核就会打印出当前进程的栈回溯信息，其中包含当前执行代

码的位置以及相邻的指令、产生错误的原因、关键寄存器的值以及函数调用关系等信息，这些信息对于调试内核错误非常有用。

打印函数调用关系的函数就是 `dump_stack()`， 该函数不仅可以用在系统出问题的时候，我们在调试内核的时候，可以通过dump_stack()函数的打印信息更方便

的了解内核代码执行流程。

接下来我们分析一下 `dump_stack` 的调用过程：

```
1 dump_stack
2     -->dump_stack_lvl
3         -->__dump_stack
4             -->dump_stack_print_info
5             -->show_stack
```

而 `__dump_stack` 的调用关系如下所示：

```
1 static void __dump_stack(const char *log_lvl)
2 {
3     dump_stack_print_info(log_lvl);//打印头消息
4     show_stack(NULL, NULL, log_lvl);//打印堆栈
```

其中由`dump_stack_print_info` 和 `show_stack` 两个函数组成，第一个函数负责打印头消息，第二个函数 `show_stack` 负责打印具体的栈：

```c
//arch/arm64/kernel/stacktrace.c
void show_stack(struct task_struct *tsk, unsigned long *sp, const char
*loglvl)
{
    dump_backtrace(NULL, tsk, loglvl);
    barrier();
}
```

而 `show_stack` 主要调用的是 `dump_backtrace()` 和 `barrier()`

```c
void dump_backtrace(struct pt_regs *regs, struct task_struct *tsk,const char
*loglvl)
{
    struct stackframe frame;
    int skip = 0;

    pr_debug("%s(regs = %p tsk = %p)\n", __func__, regs, tsk);

    if (regs) {
        if (user_mode(regs))
            return;
        skip = 1;
    }

    if (!tsk)
        tsk = current;

    if (!try_get_task_stack(tsk)) //如果获取失败
        return;

    if (tsk == current) {
        start_backtrace(&frame,
                (unsigned long)__builtin_frame_address(0),
                (unsigned long)dump_backtrace);
    } else {
        /*
         * task blocked in __switch_to
         */
        start_backtrace(&frame,
                thread_saved_fp(tsk),
                thread_saved_pc(tsk));
    }

    printk("%sCall trace:\n", loglvl);
    do {
        /* skip until specified stack frame */
        if (!skip) {
            dump_backtrace_entry(frame.pc, loglvl);
        } else if (frame.fp == regs->regs[29]) {
            skip = 0;
            /*
             * Mostly, this is the case where this function is
             * called in panic/abort. As exception handler's
             * stack frame does not contain the corresponding pc
             * at which an exception has taken place, use regs->pc
             * instead.
```

```
46                 */
47                dump_backtrace_entry(regs->pc, loglvl);
48            }
49        } while (!unwind_frame(tsk, &frame));//判断是否到达栈底
50
51        put_task_stack(tsk);
52    }
```

可以看到，核心部分是do{} while循环下的栈回溯，以及使用%pS实现的输出地址和符号名。

当该输出是直接将函数的调用直接输出到dmesg中去，并不能将堆栈信息保存下来。

如果要将堆栈信息保存下来，需要使用另外一个函数stack_trace_save()，其中这个`dump_backtrace_entry`主要是负责将指针地址转换为可读的函数名，其实就是利用了`printk()`提供的一个格式化接口：

```
1  static void dump_backtrace_entry(unsigned long where, const char *loglvl)
2  {
3      printk("%s %pSb\n", loglvl, (void *)where);
4  }
```

由此可以看出，最难的从地址到函数名到地址转换这部分内容，内核已经帮我们做好了，`dump_stack`只需要去用就行了：

```
1   * %pS output the name of a text symbol with offset
2   * %ps output the name of a text symbol without offset
3   * %pF output the name of a function pointer with its offset
4   * %pf output the name of a function pointer without its offset
5   * %pB output the name of a backtrace symbol with its offset
6   * %pR output the address range in a struct resource with decoded flags
7   * %pr output the address range in a struct resource with raw flags
8   * %pM output a 6-byte MAC address with colons
9   * %pm output a 6-byte MAC address without colons
10  * %pI4 print an IPv4 address without leading zeros
11  * %pi4 print an IPv4 address with leading zeros
12  * %pI6 print an IPv6 address with colons
13  * %pi6 print an IPv6 address without colons
14  * %pI6c print an IPv6 address as specified by RFC 5952
15  * %pU[bBlL] print a UUID/GUID in big or little endian using lower or upper
16  * case.
17  * %n is ignored
```

2.4.2 stack_trace_save

但是如果我们直接使用`dump_stack`获取进程调用栈的话，会直接将调用栈等信息直接打印到`dmesg`当中，这并不是我们所要的。

所以我们需要使用另外的接口，将调用栈等信息输出到我们的缓冲区中：

- stack_trace_save
- stack_trace_snprint

在对stack_trace_save()进行分析之后，可以发现其核心函数walk_stackframe()与dump_stack的核心函数dump_backtrace()都是基于函数unwind_frame()进行栈回溯，但不同的是dump_stack直接使用dump_backtrace_entry()进行输出，而stack_trace_save()则是进行了保存。

stack_trace_save()保存了堆栈信息，并且返回了保存的条目大小，因此只需要对保存的堆栈进行遍历，便可输出申请自旋锁并且关闭中断的函数调用关系。

其 `stack_trace_save` 的具体实现如下：

```
//#ifdef CONFIG_HAVE_RELIABLE_STACKTRACE
/**
 * stack_trace_save - Save a stack trace into a storage array
 * @store:  Pointer to storage array
 * @size:   Size of the storage array
 * @skipnr: Number of entries to skip at the start of the stack trace
 *
 * Return: Number of trace entries stored.
 */
unsigned int stack_trace_save(unsigned long *store, unsigned int size,
                  unsigned int skipnr)
{
    stack_trace_consume_fn consume_entry = stack_trace_consume_entry;
    struct stacktrace_cookie c = {
        .store  = store,
        .size   = size,
        .skip   = skipnr + 1,
    };

    arch_stack_walk(consume_entry, &c, current, NULL);
    return c.len;
}
EXPORT_SYMBOL_GPL(stack_trace_save);
```

其中这个 `stacktrace_cookie` 结构体是这样定义的：

```
struct stacktrace_cookie {
    unsigned long   *store;
    unsigned int    size;
    unsigned int    skip;
    unsigned int    len;
};
```

可以看到往下是调用了 `arch_stack_walk` ，返回的是结构体c的长度， `arch_stack_walk` 的实现方式如下：

```
noinline notrace void arch_stack_walk(stack_trace_consume_fn consume_entry,
                  void *cookie, struct task_struct *task,
                  struct pt_regs *regs)
{
    struct stackframe frame;

    if (regs)
        start_backtrace(&frame, regs->regs[29], regs->pc);
    else if (task == current)
        start_backtrace(&frame,
                (unsigned long)__builtin_frame_address(1),
                (unsigned long)__builtin_return_address(0));
    else
        start_backtrace(&frame, thread_saved_fp(task),
                thread_saved_pc(task));
```

```
16
17      walk_stackframe(task, &frame, consume_entry, cookie);
18  }
```

2.5 持有锁分析

当挂载点为申请自旋锁且关闭中断时，可以获得的锁信息主要有两类：

- 什么路径申请自旋锁并关闭中断
- 具体申请的是哪一个自旋锁

2.5.1 持有锁具体路径

对于第一类信息，可以在在申请自旋锁并关闭中断时，保存下当前进程的堆栈信息，并进行栈回溯从而得到函数的调用关系。

目前，在Linux中最常使用的此类函数是dump_stack()。

2.5.2 持有锁类别

而当我们先知道是哪一个锁被申请的时候，只能输出该锁的地址了。因为自旋锁是基本的一个锁，其常被嵌入到其他各种各样的共享资源当中，而我们通常依赖的container_of()宏是必须得预先知道什么结构的资源中嵌入了一个自旋锁，这显然是不太现实的。而我们只需要知道当前申请的锁的地址，便是可以作为锁的一个key，用于标识唯一的锁。

我们分析_raw_spin_lock_irq()函数的参数，可以知道其参数是一个raw_spinlock_t的指针。

```
1  void __lockfunc _raw_spin_lock_irq(raw_spinlock_t *lock)
2  {
3      __raw_spin_lock_irq(lock);
4  }
5  EXPORT_SYMBOL(_raw_spin_lock_irq);
```

此时，可以使用函数regs_get_register()来获取其第一个参数值。

```
1  static inline u64 regs_get_register(struct pt_regs *regs, unsigned int
   offset)
2  {
3      u64 val = 0;
4
5      WARN_ON(offset & 7);
6
7      offset >>= 3;
8      switch (offset) {
9      case 0 ... 30:
10         val = regs->regs[offset];
11         break;
12     case offsetof(struct pt_regs, sp) >> 3:
13         val = regs->sp;
14         break;
15     case offsetof(struct pt_regs, pc) >> 3:
16         val = regs->pc;
17         break;
18     case offsetof(struct pt_regs, pstate) >> 3:
19         val = regs->pstate;
```

```
        break;
    default:
        val = 0;
    }

    return val;
}
```

至此，已经可以得到申请锁并关闭中断的调用路径以及具体申请的是哪一个锁了。

2.6 文件和socket

在Linux中，一个进程都有一个PCB（struct task_struct）。进程的文件以及文件系统的相关信息在其task_struct中都有体现，其中成员files标识该进程的打开文件信息：

```
    /* Filesystem information: */
    struct fs_struct        *fs;

    /* Open file information: */
    struct files_struct     *files;
```

其中 `files_struct` 的相关结构体实现如下：

```
struct files_struct {
  /*
   * read mostly part
   */
    atomic_t count;
    bool resize_in_progress;
    wait_queue_head_t resize_wait;

    struct fdtable __rcu *fdt;
    struct fdtable fdtab;
  /*
   * written part on a separate cache line in SMP
   */
    spinlock_t file_lock ____cacheline_aligned_in_smp;
    unsigned int next_fd;
    unsigned long close_on_exec_init[1];
    unsigned long open_fds_init[1];
    unsigned long full_fds_bits_init[1];
    struct file __rcu * fd_array[NR_OPEN_DEFAULT];
};

struct fdtable {
    unsigned int max_fds;
    struct file __rcu **fd;      /* current fd array */
    unsigned long *close_on_exec;
    unsigned long *open_fds;
    unsigned long *full_fds_bits;
    struct rcu_head rcu;
};
```

其中，具体的打开文件信息交由struct fdatble进行管理，fdtable中的成员max_fds标识当前支持的最大可打开文件数目，而具体的打开文件则由一个file代为表示，其中一个fd表示的文件是否打开需要经过fdtable的成员open_fds通过函数fd_is_open()进行一次确认。

```
struct file {
    union {
        struct llist_node   fu_llist;
        struct rcu_head     fu_rcuhead;
    } f_u;
    struct path     f_path;
    struct inode        *f_inode;   /* cached value */
    const struct file_operations    *f_op;
    ......
}

static inline bool fd_is_open(unsigned int fd, const struct fdtable *fdt)
{
    return test_bit(fd, fdt->open_fds);
}
```

而每一个打开文件对象file中有一个成员f_path，这是一个struct dentry对象。

```
struct path {
    struct vfsmount *mnt;
    struct dentry *dentry;
} __randomize_layout;

struct dentry {
    /* RCU lookup touched fields */
    unsigned int d_flags;       /* protected by d_lock */
    seqcount_spinlock_t d_seq;  /* per dentry seqlock */
    struct hlist_bl_node d_hash;    /* lookup hash list */
    struct dentry *d_parent;    /* parent directory */
    struct qstr d_name;
    struct inode *d_inode;      /* Where the name belongs to - NULL is
                     * negative */
    unsigned char d_iname[DNAME_INLINE_LEN];    /* small names */
    ......
}
```

可以看到struct dentry中有两个成员，一个是struct qstr d_name，另外一个是unsigned char d_iname[DNAME_INLINE_LEN]。

其中，struct qstr的定义如下，可以看到其中有一个成员const unsigned char *name。

```
struct qstr {
    union {
        struct {
            HASH_LEN_DECLARE;
        };
        u64 hash_len;
    };
    const unsigned char *name;
};
```

其实当文件名称size < DNAME_INLINE_LEN时，d_name.name指向d_iname；而当文件名称size ≥ DNAME_INLINE_LEN时，d_name.name的空间需要单独通过kmalloc申请。

以上，便可以拿到进程的所有打开文件信息。

而确认一个文件是不是网络相关的方法有很多，这里简单举几个例子。

第一个，文件file最终会指向一个struct inode。inode用于标识唯一的文件结构体。而在socket创建的时候，会调用到sock_alloc()函数

```
struct socket *sock_alloc(void)
{
    struct inode *inode;
    struct socket *sock;

    inode = new_inode_pseudo(sock_mnt->mnt_sb);
    if (!inode)
        return NULL;

    sock = SOCKET_I(inode);

    inode->i_ino = get_next_ino();
    inode->i_mode = S_IFSOCK | S_IRWXUGO;
    inode->i_uid = current_fsuid();
    inode->i_gid = current_fsgid();
    inode->i_op = &sockfs_inode_ops;

    return sock;
}
EXPORT_SYMBOL(sock_alloc);
```

在这里可以看到，我们可以通过inode→i_mode是否标记的S_IFSOCK来判断是否是一个socket相关的文件。同时我们也可以通过inode→i_op是否是sockfs_inode_ops来判断。

第二个，我们来看申请完socket，之后socket与file的绑定函数sock_map_fd()。

```
static int sock_map_fd(struct socket *sock, int flags)
{
    struct file *newfile;
    int fd = get_unused_fd_flags(flags);
    if (unlikely(fd < 0)) {
        sock_release(sock);
        return fd;
    }

    newfile = sock_alloc_file(sock, flags, NULL);
    if (!IS_ERR(newfile)) {
        fd_install(fd, newfile);
        return fd;
    }

    put_unused_fd(fd);
    return PTR_ERR(newfile);
}

struct file *sock_alloc_file(struct socket *sock, int flags, const char
*dname)
{
    struct file *file;

    if (!dname)
        dname = sock->sk ? sock->sk->sk_prot_creator->name : "";
```

```
26
27      file = alloc_file_pseudo(SOCK_INODE(sock), sock_mnt, dname,
28                  O_RDWR | (flags & O_NONBLOCK),
29                  &socket_file_ops);
30      if (IS_ERR(file)) {
31          sock_release(sock);
32          return file;
33      }
34
35      sock->file = file;
36      file->private_data = sock;
37      stream_open(SOCK_INODE(sock), file);
38      return file;
39  }
40  EXPORT_SYMBOL(sock_alloc_file);
```

在sock_alloc_file中将file的f_ops指向soket_file_ops，因此我们也可以用该成员的指向来判断一个打开文件是否是一个socket。如果是一个socket，则file的private_data指针则指向一个sock。

这部分在socket_alloc_file()函数的以下语句中可以看出，socket与file之间的对应关系：

```
1  sock->file = file;
2  file->private_data = sock;
```

socket中的一些成员定义了网络的最基本信息，如通信双方的IP地址和端口号。

```
1  #define sk_num          __sk_common.skc_num
2  #define sk_dport        __sk_common.skc_dport
3
4  #define sk_daddr        __sk_common.skc_daddr
5  #define sk_rcv_saddr        __sk_common.skc_rcv_saddr
```

至此，最基本的socket信息也得到了

2.7 cache存储机制

cache机制存储目前的设计思路有两种：

- 基于ring buffer思想的机制；
- 基于定时清除思想的机制；

2.7.1 基于ring buffer思想的机制

对于在确定关中断的时间大于阈值之后，应该申请内存并保存当前的关键数据。

因为此时处于kprobe执行的上下文中，如果频繁申请内存的过程可能会影响系统性能，并且容易造成内存泄漏。

所以，在kprobe上下文中关键数据的保存使用了ring buffer的思想。

首先，定义一个obj_pool对象，用于充当ring buffer。

```
struct obj_pool {
    unsigned long       pg_addr;
    unsigned int        pg_order;
    unsigned int        obj_size;
    unsigned int        max_idx;
    atomic_t            next_idx;
};
```

obj_pool的思想即为一次申请2 << pg_order大小的页，页起始地址为pg_addr，其中存储的对象大小为obj_size，最多可以存放max_idx个，而下一次可以申请到obj对象的下标为next_idx。

因为这个obj_pool对象可能同时被几个CPU同时用来申请obj对象，所以next_idx使用原子变量进行定义。

故在obj_pool中获得一个obj对象，并返回地址的函数是new_obj：

2.7.2 基于定时清除思想的机制

本项目将收集到的信息以进程为key进行组织，而长时间的存储对于获得系统的当前状况的用处不大，而且会严重消耗系统内存，影响系统性能。

为便于系统维护人员分析，这里采用了一种定时清除的思想。

我们可以通过proc接口savetime定义自动清除的时限，当达到时限的时候，自动释放内存。

```
struct lock_info {
    unsigned int        lock_num;
    unsigned long       lock_time;
        unsigned long       lock_time_stamp;
    unsigned long       lock_addr;
        unsigned int        num_entries;
    unsigned long         stack_entries[NUM_STACK_ENTRIES];
    struct timer_list     timer;
    struct lock_list      *lock_list;
    struct list_head      node;
};
```

在需要动态清除的数据结构中嵌入了一个timer_list类型的定时器，当该类型的对象被初始化以及被修改之后，设置定时器的到期时间。

```
// 修改
lock_info->timer.expires = jiffies + HZ * getNodeParm(savetime);
mod_timer(&(lock_info->timer), lock_info->timer.expires);

// 初始化
timer_setup(&(lock_info->timer), timer_handler, 0);
lock_info->timer.expires = jiffies + HZ * getNodeParm(savetime);
add_timer(&(lock_info->timer));
```

如果该定时器被触发，则执行timer_handler函数进行清除工作。

```
static void timer_handler(struct timer_list *timer)
{
    struct lock_info *lock_info;
    lock_info = container_of(timer, struct lock_info, timer);

```

```
    spin_lock(&(lock_info->lock_list->lock));

    lock_info->lock_list->total_num -= lock_info->lock_num;
    lock_info->lock_list->total_time -= lock_info->lock_time;
    list_del(&(lock_info->node));

    spin_unlock(&(lock_info->lock_list->lock));

    kfree(lock_info);
}
```

2.8 数据收集与组织

2.8.1 数据收集方式

使用ring buffer机制可以减少系统在kprobe上下文中停留的时间，但如果把要获取的所有信息都在kprobe上下文中进行收集，无疑会影响系统的性能。

又因为大量的信息其实是和进程相关的，而这些信息都可以通过struct task_struct进行获取。

故只需要保存一个task_struct的指针即可，真正后续信息的获取可以留给内核线程去执行。

```
struct kp_info {
    atomic_t            kp_state;
    struct task_struct  *task;
    unsigned long       lock_addr;
    unsigned int        cpu;
    unsigned long       time_stamp;
    unsigned long       delta;
    unsigned int        num_entries;
    unsigned long       stack_entries[NUM_STACK_ENTRIES];
};
```

但为避免此时指向的task_struct在后续内核线程的处理过程中失效，需要在获取task_struct指针之后，增加引用计数；而当该task_struct被内核线程处理完成之后，需要减去对应的引用计数。

```
task = (struct task_struct *)current;
refcount_inc(&(task->usage));
```

除了task_struct的指针需要注意，为了在进程上下文与此时的kprobe上下文中保持读写同步，并尽量减少kprobe上下文的处理时间，这里使用了atomic_t类型的成员kp_state。

kp_state算上0，共有4种状态。

```
#define KP_STATE_WAITING        0x1
#define KP_STATE_READING        0x1 << 1
#define KP_STATE_WRITING        0x1 << 2
```

- 0代表未被处理或已经被读取完成；
- KP_STATE_WAITING代表已被写入，等待读取；
- KP_STATE_READING代表当前正在读；
- KP_STATE_WRITING代表当前正在写。

以kprobe上下文中的锁获取代码为例：

```
if (!(kp_state = atomic_cmpxchg(&(kp_info->kp_state), 0, KP_STATE_WRITING))
||
            (kp_state = atomic_cmpxchg(&(kp_info->kp_state),
                        KP_STATE_WAITING, KP_STATE_WRITING)) &
KP_STATE_WAITING) {

            if (kp_state & KP_STATE_WAITING)
                refcount_dec(&(kp_info->task->usage));
```

如果当前状态是未被处理或已经被读取完成，则直接置为正在写状态；

而如果当前状态是一个等待被读取的状态，则直接覆盖，同样置为正在写状态，但需要注意的是，释放之前的引用计数。

2.8.2 数据组织方式

获取到数据后，我们以pid为索引进行组织，方便后续的分析。

常规的组织方式可能会选择内核的哈希表或者红黑树进行组织。但我们在对比之后，选择了xarray这种数据结构。

哈希表需要我们自己定义散列函数，如果没有一个比较合理的散列算法，造成的冲突会很大，会严重影响索引效率。

而红黑树在频繁读写的情况，需要一个全局性的锁来保护临界资源，且争用相当频繁。

而xarray可以视为一种哈希+变长的指针数组，且利用RCU的优势保证器执行查找不需要锁定。

故，最终我们选择xarray作为数据组织的基础数据结构。

xarray中存放的是struct task_info类型的指针。

```
struct task_info {
    unsigned int                pid;
    unsigned int                cpu;
    char                        comm[TASK_COMM_LEN];
    char                        exe[MAX_FILE_LEN];
    struct rw_semaphore         sem;
    struct file_list            files;
    struct lock_list            locks;
};
```

task_info中存储的都是公有数据，不需要反复更新。一旦创建，只需要检查是否打开了新的文件，如果是，则更新文件信息；以及更新本次持有锁信息。

task_info中的公有信息通过rw_semaphore读写信号量来进行保护。因为对这部分公有信息的读写都是在进程上下文中执行，故读写信号量可以满足要求。

文件信息是通过struct file_list以及struct file_node两个结构体进行组织的。

```
struct file_list {
    unsigned long       open_fds;
    struct list_head    head;
};

struct file_node {
    struct list_head    node;
    char                file_name[MAX_FILE_LEN];
    unsigned int        f_flags;
    void *              private_data;
};
```

其中open_fds可以视为进程打开过的文件的位图。

```
fdt = rcu_dereference_raw(task->files->fdt);
if (fdt == NULL || fdt->open_fds == NULL) {
    goto rcu_unlock;
}

fds = *(fdt->open_fds);
fds ^= task_info->files.open_fds;
fds &= ~(task_info->files.open_fds);

task_info->files.open_fds |= fds;
```

进程持有的fdt→open_fds是进程现在打开的文件位图。

先通过一个异或操作可以得到进程前后变化的文件，再通过与之前打开的文件的非取反可以得到上次没有打开而这次已经打开的文件位图。

之后，通过按位查找的方式可以获取要更新的文件信息，无疑大大提升了速度。

```
if (fds) {
        while ((fd_bit = ffs(fds))) {
            fd = fd_bit - 1;

            f = rcu_dereference_raw(fdt->fd[fd]);
            if (f) {
                file_node = kmalloc(sizeof(*file_node), GFP_KERNEL);
                        ...
```

而锁的信息则是通过struct lock_list和lock_info进行组织。

```
struct lock_list {
    spinlock_t      lock;
    unsigned int        total_num;
    unsigned long       total_time;
    struct list_head    head;
};

struct lock_info {
    unsigned int        lock_num;
    unsigned long       lock_time;
    unsigned long       lock_addr;
    unsigned int    num_entries;
    unsigned long   stack_entries[NUM_STACK_ENTRIES];
```

```
    struct timer_list   timer;
    struct lock_list    *lock_list;
    struct list_head    node;
};
```

其中lock_list需要维护该进程的持有锁的总次数以及总时间。而lock_info则具体到某一个具体的锁地址。

之前提到我们采用了定时清除的机制，当一段时间没有持有锁某个具体的锁之后，该锁信息会被销毁。而销毁的工作是在timer中断中执行的，因此这部分临界资源应该采用spin_lock。

以下是初始化一个lock_info的代码：

```
lock_info = kmalloc(sizeof(*lock_info), GFP_KERNEL);

lock_info->lock_num = 1;
lock_info->lock_time = delta;
lock_info->lock_addr = lock_addr;
lock_info->num_entries = kp_info->num_entries;
memcpy(lock_info->stack_entries,
            kp_info->stack_entries, sizeof(unsigned long) *
NUM_STACK_ENTRIES);
INIT_LIST_HEAD(&(lock_info->node));
lock_info->lock_list = &(task_info->locks);
timer_setup(&(lock_info->timer), timer_handler, 0);
lock_info->timer.expires = jiffies + HZ * getNodeParm(savetime);

spin_lock_irq(&(task_info->locks.lock));

add_timer(&(lock_info->timer));
list_add_tail(&(lock_info->node), &(task_info->locks.head));
task_info->locks.total_num++;
task_info->locks.total_time += delta;

spin_unlock_irq(&(task_info->locks.lock));
```

3. 系统实现

3.1 kprobe挂载程序实现

首先我们先来看一下，kprobe的接口：

3.1.1 API接口

3.1.1.1 注册kprobe探针

Kprobes 对每种类型的探针包括一个“注册”函数和一个“注销”函数。

还包括 `register_*probes` 和 `unregister_*probes` 函数，用于批量注册或者批量取消注册探针。

```
#include <linux/kprobes.h>
int register_kprobe(struct kprobe *kp);
```

在地址 `kp->addr` 处设置断点。当断点被命中时，Kprobes 调用 `kp->pre_handler`。被探测的指令单步执行后，Kprobe 调用 `kp->post_handler`

如果在 `kp->pre_handler` 或 `kp->post_handler` 的执行过程中，或被探测指令的单步执行过程中发生故障，Kprobes 将调用 `kp->fault_handler`

如果 `kp->flags` 设置为 `KPROBE_FLAG_DISABLED`，则该 kp 将被注册但禁用，因此，在调用 `enable_kprobe(kp)` 之前不会触发其处理程序。

3.1.1.2 注册kprobe探针

```
#include <linux/kprobes.h>
int register_kretprobe(struct kretprobe *rp);
```

为地址为 `rp->kp.addr` 的函数建立返回探针。当该函数返回时，Kprobes 调用 `rp->handler`。

`register_kretprobe()` 成功时返回 `0`，否则返回负数。

3.1.1.3 注销探针

```
#include <linux/kprobes.h>
void unregister_kprobe(struct kprobe *kp);
void unregister_kretprobe(struct kretprobe *rp);
```

移除指定的探针。 注册探针后，可以随时调用取消注册函数

> 如果找到不正确的探针（例如未注册的探针），它们会清除探针的 addr 字段。

3.1.1.4 前处理函数

```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
int pre_handler(struct kprobe *p, struct pt_regs *regs);
```

调用时 `p` 指向与断点关联的 kprobe，而 `regs` 指向包含在断点被击中时保存的寄存器的结构。 除非您是 Kprobes 极客，否则请在此处返回 0。

3.1.1.5 后处理函数

```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
void post_handler(struct kprobe *p, struct pt_regs *regs,unsigned long
flags);
```

`p` 和 `regs` 与 pre_handler 中的含义相同。 flags 似乎总是为`0`

3.1.1.6 错误处理函数

```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
int fault_handler(struct kprobe *p, struct pt_regs *regs, int trapnr);
```

`p` 和 `regs` 也是与 pre_handler 中的含义相同。

`trapnr` 是与故障相关的特定于体系结构的陷阱编号（例如，在 i386 上，13 表示一般保护故障，14 表示页面故障）。

如果成功处理了异常，则返回 1。

3.1.1.7 禁用探针

```
#include <linux/kprobes.h>
int disable_kprobe(struct kprobe *kp);
int disable_kretprobe(struct kretprobe *rp);
```

暂时禁用指定的 `*probe`。 您可以使用 `enable_*probe()` 再次启用它。 但是您必须指定已注册的探针。

3.1.1.8 启用探针

```
#include <linux/kprobes.h>
int enable_kprobe(struct kprobe *kp);
int enable_kretprobe(struct kretprobe *rp);
```

启用已被 `disable_*probe()` 禁用的探针， 您必须指定已注册的探针。

3.1.2 具体实现

由中断挂载点分析可知，kprobe挂载可以分为三对：

序号	关中断	开中断
1	raw_spin_lock_irq()	raw_spin_unlock_irq()
2	raw_spin_lock_irqsave()	raw_spin_lock_irqrestore()
3	mask_irq()	unmask_irq()

其中，挂载点表示为：

```
static char symbol_lock_irq[MAX_SYMBOL_LEN] = "_raw_spin_lock_irq";
static char symbol_unlock_irq[MAX_SYMBOL_LEN] = "_raw_spin_unlock_irq";

static char symbol_lock_irqsave[MAX_SYMBOL_LEN] = "_raw_spin_lock_irqsave";
static char symbol_unlock_irqrestore[MAX_SYMBOL_LEN] =
"_raw_spin_unlock_irqrestore";

static char symbol_mask_irq[MAX_SYMBOL_LEN] = "mask_irq";
static char symbol_unmask_irq[MAX_SYMBOL_LEN] = "unmask_irq";
```

将挂载点函数分别赋值给特定的kprobe结构体：

```
static struct kprobe kp_lock_irq = {
    .symbol_name = symbol_lock_irq,
};

static struct kprobe kp_unlock_irq = {
    .symbol_name = symbol_unlock_irq,
};

static struct kprobe kp_lock_irqsave = {
    .symbol_name = symbol_lock_irqsave,
};

```

```
static struct kprobe kp_unlock_irqrestore = {
    .symbol_name = symbol_unlock_irqrestore,
};

static struct kprobe kp_mask_irq = {
    .symbol_name = symbol_mask_irq,
};

static struct kprobe kp_unmask_irq = {
    .symbol_name = symbol_unmask_irq,
};
```

在每一对开关中断的处理过程中，因为需要在关中断时记录:

- 下关中断的时间
- 关闭的中断号
  - 对于mask_irq()来说，是特定中断号；
  - 对于*raw_spin_lock_irq()*和raw_spin_lock_irqsave()来说，是全体中断

所以，设计出了一个per_cpu的数据结构struct disable_irq_entry，用于记录不同的关中断的执行时间以及对应的中断号。

```
struct disable_irq_entry {
    int flag;
    u64 entry_time;
    int irq;
};

static struct disable_irq_entry __percpu *pdie_lock_irq;
static struct disable_irq_entry __percpu *pdie_lock_irqsave;
static struct disable_irq_entry __percpu *pdie_mask_irq;
```

在模块插入时，对三个per_cpu变量的flag设置为0，代表还没有开始记录信息；而当屏蔽中断的事件发生后，则将flag设置1，并记录其entry_time。

```
static int __init kprobe_init(void)
{
    struct disable_irq_entry *ptrdie_lock_irq, ptrdie_lock_irqsave, ptrdie_mask_irq;

    pdie_lock_irq = alloc_percpu(struct disable_irq_entry);
    pdie_lock_irqsave = alloc_percpu(struct disable_irq_entry);
    pdie_mask_irq = alloc_percpu(struct disable_irq_entry);

    if (!pdie_lock_irq || !pdie_lock_irqsave || !pdie_mask_irq) {
        pr_err("alloc_percpu failed\n");
        return 0;
    }

    for_each_online_cpu(i) {
        ptrdie_lock_irq = per_cpu_ptr(pdie_lock_irq, i);
        ptrdie_lock_irq->flag = 0;

        ptrdie_lock_irqsave = per_cpu_ptr(pdie_lock_irqsave, i);
        ptrdie_lock_irqsave->flag = 0;

```

```
        ptrdie_mask_irq = per_cpu_ptr(pdie_mask_irq, i);
        ptrdie_mask_irq->flag = 0;
    }

    kp_lock_irq.pre_handler = handler_lock_irq;
    kp_unlock_irq.pre_handler = handler_unlock_irq;

    kp_lock_irqsave.pre_handler = handler_lock_irqsave;
    kp_unlock_irqrestore.pre_handler = handler_unlock_irqrestore;

    kp_mask_irq.pre_handler = handler_mask_irq;
    kp_unmask_irq.pre_handler = handler_unmask_irq;

    ......
}
```

这里以handler_lock_irq()和handler_unlock_irq()的实现为例：

```
static int __kprobes handler_lock_irq(struct kprobe *p, struct pt_regs
*regs)
{
    u32 cpu = smp_processor_id();
    u64 now = ktime_get_ns();
    struct disable_irq_entry *entry_pos;

    entry_pos = per_cpu_ptr(pdie, cpu);
    entry_pos->entry_time = now;
    entry_pos->flag = 1;
    entry_pos->irq = -1;

    return 0;
}

static int __kprobes handler_unlock_irq(struct kprobe *p, struct pt_regs
*regs)
{
    u32 cpu = smp_processor_id();
  u64 now = ktime_get_ns();
    u64 delta = 0;

    struct disable_irq_entry *entry_pos;

    entry_pos = per_cpu_ptr(pdie, cpu);
    entry_pos->flag = 0;
    delta = now - entry_pos->entry_time;

    .....
}
```

3.2 procfs交互

3.2.1 procfs API

procfs是进程文件系统的缩写，包含一个伪文件系统（启动时动态生成的文件系统），用于通过内核访问进程信息（Linux将此概念扩展到了非进程相关数据）。

这个文件系统通常被挂载到/proc目录。由于/proc不是一个真正的文件系统，它也就不占用存储空间，只是占用有限的内存。

要在procfs下创建自定义的目录及文件，需要用到接口函数proc_mkdir及proc_create。

```
static inline struct proc_dir_entry *proc_mkdir(const char *name,
    struct proc_dir_entry *parent) {return NULL;}
#define proc_create(name, mode, parent, proc_ops) ({NULL;})
```

在创建具体文件时，需要实现自定义的struct proc_ops结构体，用于支持其读写等特性。

```
struct proc_ops {
    unsigned int proc_flags;
    int (*proc_open)(struct inode *, struct file *);
    ssize_t (*proc_read)(struct file *, char __user *, size_t, loff_t *);
    ssize_t (*proc_read_iter)(struct kiocb *, struct iov_iter *);
    ssize_t (*proc_write)(struct file *, const char __user *, size_t, loff_t
*);
    loff_t  (*proc_lseek)(struct file *, loff_t, int);
    int (*proc_release)(struct inode *, struct file *);
    __poll_t (*proc_poll)(struct file *, struct poll_table_struct *);
    long    (*proc_ioctl)(struct file *, unsigned int, unsigned long);
#ifdef CONFIG_COMPAT
    long    (*proc_compat_ioctl)(struct file *, unsigned int, unsigned
long);
#endif
    int (*proc_mmap)(struct file *, struct vm_area_struct *);
    unsigned long (*proc_get_unmapped_area)(struct file *, unsigned long,
unsigned long, unsigned long, unsigned long);
} __randomize_layout;
```

由于procfs的默认操作函数只使用一页的缓存，如果自定义的数据较大就有点麻烦，并且在输出一系列结构体中的数据也比较不灵活，需要自己在proc_read函数中实现迭代，容易出现Bug。

出于灵活性和未来可扩展性的考虑，我们使用seq_file接口，此接口可以用于创建一个由一系列数据顺序组合而成的虚拟文件。seq_file的定义如下：

```
struct seq_file {
    char *buf; /* seq_file接口使用的缓存页指针 */
    size_t size; /* seq_file接口使用的缓存页大小 */
    size_t from; /* 从seq_file中向用户态缓冲区拷贝时相对于buf的偏移地址 */
    size_t count; /* buf中可以拷贝到用户态的字符数目 */
    size_t pad_until;
    loff_t index; /* start，next的处理的下标pos数值 */
    loff_t read_pos; /* 当前已拷贝到用户态的数据量大小 */
    struct mutex lock; /* 针对此seq_file操作的互斥锁 */
    const struct seq_operations *op; /* 操作实际底层数据的函数 */
    int poll_event;
    const struct file *file;
    void *private;
};
```

seq_file中的op定义了操作实际底层数据的函数，seq_file内部机制使用这些函数访问底层的实际数据结构体，并不断沿数据序列向前，同时逐个输出序列里的数据到seq_file自建的缓存（大小为一页）中。也就是说seq_file内部机制实现了对序列数据的读取和放入缓存的机制，因此我们需要实现底层的迭代函数接口。

```
struct seq_operations {
    void * (*start) (struct seq_file *m, loff_t *pos);
    void (*stop) (struct seq_file *m, void *v);
    void * (*next) (struct seq_file *m, void *v, loff_t *pos);
    int (*show) (struct seq_file *m, void *v);
};
```

start方法会首先被调用，它的作用是设置访问的起始点。

设置好访问起始点，seq_file内部机制可能会使用show方法获取start返回值指向的结构体中的数据到内部缓存，并适时送往用户空间。

show方法就是负责将v指向的元素中的数据输出到seq_file的内部缓存，当时其中必须要借助seq_file提供的一些类似于printf的接口函数：

```
void seq_printf(struct seq_file *m, const char *fmt, ...);
void seq_putc(struct seq_file *m, char c);
void seq_puts(struct seq_file *m, const char *s);
void seq_put_decimal_ull_width(struct seq_file *m, const char *delimiter,
                    unsigned long long num, unsigned int width);
void seq_put_decimal_ull(struct seq_file *m, const char *delimiter,
                unsigned long long num);
void seq_put_decimal_ll(struct seq_file *m, const char *delimiter, long long
num);
void seq_put_hex_ll(struct seq_file *m, const char *delimiter,
                unsigned long long v, unsigned int width);
```

在show函数返回之后，seq_file机制可能需要移动到下一个数据元素，那就必须使用next方法。

如果next的返回值是非NULL，则是下一个需要输出到缓存的元素指针；否则表明已经输出结束，将会调用stop方法做清理。

而在实际应用中，将proc中的文件与一个seq_file结合起来，最简单的函数是single_open和single_open_size：

```
int single_open(struct file *, int (*)(struct seq_file *, void *), void *);
int single_open_size(struct file *, int (*)(struct seq_file *, void *), void
*, size_t);
```

以single_open为例：

```
int single_open(struct file *file, int (*show)(struct seq_file *, void *),
        void *data)
{
    struct seq_operations *op = kmalloc(sizeof(*op), GFP_KERNEL_ACCOUNT);
    int res = -ENOMEM;

    if (op) {
        op->start = single_start;
        op->next = single_next;
        op->stop = single_stop;
```

```
        op->show = show;
        res = seq_open(file, op);
        if (!res)
            ((struct seq_file *)file->private_data)->private = data;
        else
            kfree(op);
    }
    return res;
}
EXPORT_SYMBOL(single_open);
```

可以看到，single_open主要实现了自定义的show方法，实现了将自定义的数据输入到seq_file的缓冲区中，再由seq_file输出的用户缓冲区。

3.2.2 具体实现

首先定义节点名称：

```
#define NODE "enable"
```

其次，定义每次读写的字符串大小以及 `proc_dir_entry` 结构体：

```
#define KS 32
static char kstring[KS];
static struct proc_dir_entry *enable_proc;
```

接下来就是具体读写函数的实现：

```
static ssize_t param_read(struct file *file, char __user *buf, size_t lbuf,
loff_t *ppos)
{
    int nbytes = sprintf(kstring, "%d\n", enable);
    return simple_read_from_buffer(buf, lbuf, ppos, kstring, nbytes);
}

static ssize_t param_write(struct file *file, const char __user *buf, size_t
lbuf,loff_t *ppos)
{
    ssize_t rc;
    rc = simple_write_to_buffer(kstring, lbuf, ppos, buf, lbuf);
    sscanf(kstring, "%d", &enable);
    optParam(enable);
    return rc;
}
static struct proc_ops my_proc_fops = {
    .proc_write = param_write,
    .proc_read  = param_read,
};
```

最后需要注意的是，`proc_create(NODE, 0, my_root, &my_proc_fops)` ，如果第三个参数为 `NULL` 的话，则证明直接挂载在 `/proc` 下，否则在自己定义的目录下。

3.3 双链表存储实现

使用双链表存储，首先需要自己定义出数据结构struct info_entry，用于存储一次关中断的系统信息。

pid	本次开关中断的具体进程ID
cpu	本次开关中断是发生在哪个CPU上
time	本次关中断的持续时间
comm	本次开关中断的进程的名称
stack_entries	当申请自旋锁并关中断时，保存申请自旋锁的调用路径
num_stack_entries	保存的调用路径的条目个数
irq	本次具体关闭了哪个irq，如果为-1，则代表关闭了本cpu上的所有中断
lock_address	当申请自旋锁并关中断时，保存申请的自旋锁的地址

struct info_entry中有一个struct list_head的成员node，其作为链表成员被struct info_head管理。

struct info_head有一个原子变量成员num，用于记录当前链表中的成员个数；

此外，还有一个struct list_head的指针成员new，用于指向所管理的双链表中最新的成员。

```
struct info_entry {
    struct  list_head node;
    u32     pid;
    u32     cpu;
    u64     time;
    char    comm[TASK_COMM_LEN];
        unsigned long   stack_entries[NUM_STACK_ENTRIES];
        u32     num_stack_entries;
        int     irq;
        unsigned long lock_address;
};

struct info_head {
    atomic_t num;
    struct list_head list;
    struct list_head *new;
};

static struct info_head printinfo_head = {
    .num = {
        0,
    },
    .list = {
        .prev = &(printinfo_head.list),
        .next = &(printinfo_head.list),
    },
    .new = &(printinfo_head.list),
};
```

使用双链表进行存储数据时，首先对比链表个数是否超出了cache_size，如果超出了，则不增加链表的大小，替换掉当前链表中最旧的数据；否则，直接向链表中添加一个新的成员进去

```
if (atomic_read(&(printinfo_head.num)) >= cache_size) {
            lnode = printinfo_head.new->next;
            if (lnode == &(printinfo_head.list)) {
                lnode = lnode->next;
            }

            printinfo_head.new = lnode;

            pos = container_of(lnode, struct info_entry, node);

            pos->cpu = cpu;
            pos->pid = current->pid;
            pos->time = delta;
            memcpy(pos->comm, current->comm, TASK_COMM_LEN);
            pos->irq = irq;
            pos->lock_address = lock_address;

            pos->num_stack_entries = stack_trace_save(
                (unsigned long *)(pos->stack_entries),
                NUM_STACK_ENTRIES,
                1);



        } else {
            printinfo_entry = kmalloc(sizeof(*printinfo_entry), GFP_KERNEL);

            printinfo_entry->cpu = cpu;
            printinfo_entry->pid = current->pid;
            printinfo_entry->time = delta;
            memcpy(printinfo_entry->comm, current->comm, TASK_COMM_LEN);
            printinfo_entry->irq = irq;
            printinfo_entry->lock_address = lock_address;


            printinfo_entry->num_stack_entries = stack_trace_save(
                (unsigned long *)(printinfo_entry->stack_entries),
                NUM_STACK_ENTRIES,
                1);

            atomic_inc(&(printinfo_head.num));
            list_add_tail(&(printinfo_entry->node), &(printinfo_head.list));
            printinfo_head.new = &(printinfo_entry->node);
}
```

至此，实现了基于双链表的存储实现。

3.4 kfifo缓存实现

3.4.1 API

对kfifo的操作主要集中在以下几个函数：

kfifo_alloc()动态分配一个新的kfifo缓冲区。

```
#define kfifo_alloc(fifo, size, gfp_mask)
__kfifo_int_must_check_helper(
({
    typeof((fifo) + 1) __tmp = (fifo);
    struct __kfifo *__kfifo = &__tmp->kfifo;
    __is_kfifo_ptr(__tmp) ?
    __kfifo_alloc(__kfifo, size, sizeof(*__tmp->type), gfp_mask) :
    -EINVAL;
})
)
```

kfifo_put()向kfifo缓冲区中放入一个新的数据：

```
#define kfifo_put(fifo, val)
({
    typeof((fifo) + 1) __tmp = (fifo);
    typeof(*__tmp->const_type) __val = (val);
    unsigned int __ret;
    size_t __recsize = sizeof(*__tmp->rectype);
    struct __kfifo *__kfifo = &__tmp->kfifo;
    if (__recsize)
        __ret = __kfifo_in_r(__kfifo, &__val, sizeof(__val),
            __recsize);
    else {
        __ret = !kfifo_is_full(__tmp);
        if (__ret) {
            (__is_kfifo_ptr(__tmp) ?
            ((typeof(__tmp->type))__kfifo->data) :
            (__tmp->buf)
            )[__kfifo->in & __tmp->kfifo.mask] =
                *(typeof(__tmp->type))&__val;
            smp_wmb();
            __kfifo->in++;
        }
    }
    __ret;
})
```

kfifo_get()从kfifo缓冲区中取出一个最“旧”的数据：

```
#define kfifo_get(fifo, val)
__kfifo_uint_must_check_helper(
({
    typeof((fifo) + 1) __tmp = (fifo);
    typeof(__tmp->ptr) __val = (val);
    unsigned int __ret;
    const size_t __recsize = sizeof(*__tmp->rectype);
    struct __kfifo *__kfifo = &__tmp->kfifo;
    if (__recsize)
        __ret = __kfifo_out_r(__kfifo, __val, sizeof(*__val),
```

```
            __recsize);
    else {
        __ret = !kfifo_is_empty(__tmp);
        if (__ret) {
            *(typeof(__tmp->type))__val =
                (__is_kfifo_ptr(__tmp) ?
                ((typeof(__tmp->type))__kfifo->data) :
                (__tmp->buf)
                )[__kfifo->out & __tmp->kfifo.mask];
            smp_wmb();
            __kfifo->out++;
        }
    }
    __ret;
})
)
```

在使用kfifo的情况下，不再需要自己去定义替换规则和手动确定哪些数据最“旧”，kfifo已经帮我们确定好了，我们要做的只是在新数据来临的时候调用kfifo_put()，数据读取的时候调用kfifo_get()就可以了

3.4.2 具体实现

kfifo提供两种创建队列的方法：

- 动态创建
- 静态创建

动态创建

```
struct kfifo g_fifoqueue;
/*
该函数创建并初始化一个size大小的kfifo。
内核使用gfp_mask标识符分配队列的缓冲区内存。
如果成功，函数返回0，错误则返回负数的错误码。如果要自己分配缓冲区，可以调用函数：
*/
int kfifo_alloc(struct kfifo *fifo, unsigned int size, gfp_t gfp_mask);
void kfifo_init(struct kfifo *fifo, void *buffer, unsigned int size);
```

静态创建

```
DECLARE_KFIFO(name, size) ;
INIT_KFIFO(name);
```

在本模块中，采用静态创建的方式。

而关于数据的入队和出队，使用到了以下两个接口：

```
kfifo_in(&output_fifo, info, ret);
```

```
kfifo_to_user(&output_fifo, buf, lbuf, &actual_readed);
```

先将获取到的数据入队，然后再将队中的数据拷贝到用户空间进行读取。

3.5 调用栈打印实现

在申请自旋锁并且关闭本地中断后，如果检测到关中断时间超过了我们设定的阈值，则使用stack_trace_save()将其函数调用关系保存下来。

```
pos->num_stack_entries = stack_trace_save(
                (unsigned long *)(pos->stack_entries),
                NUM_STACK_ENTRIES,
                1);
```

stack_trace_save()将当前的函数调用关系保存到目标数组当中，返回保存的函数条目个数。

而在输出的时候，只需要对数组进行遍历，并采用%pS输出函数地址对应的函数名即可。

```
for (i = 0; i < pos->num_stack_entries; i++) {
            seq_printf(p, "%pS\n", (void *)pos->stack_entries[i]);
}
```

3.6 进程其他信息获取实现

进程的普通信息如pid和comm可以直接使用current指向的struct task_struct结构体的成员pid和comm进行获取。

```
pos->cpu = cpu;
pos->pid = current->pid;
```

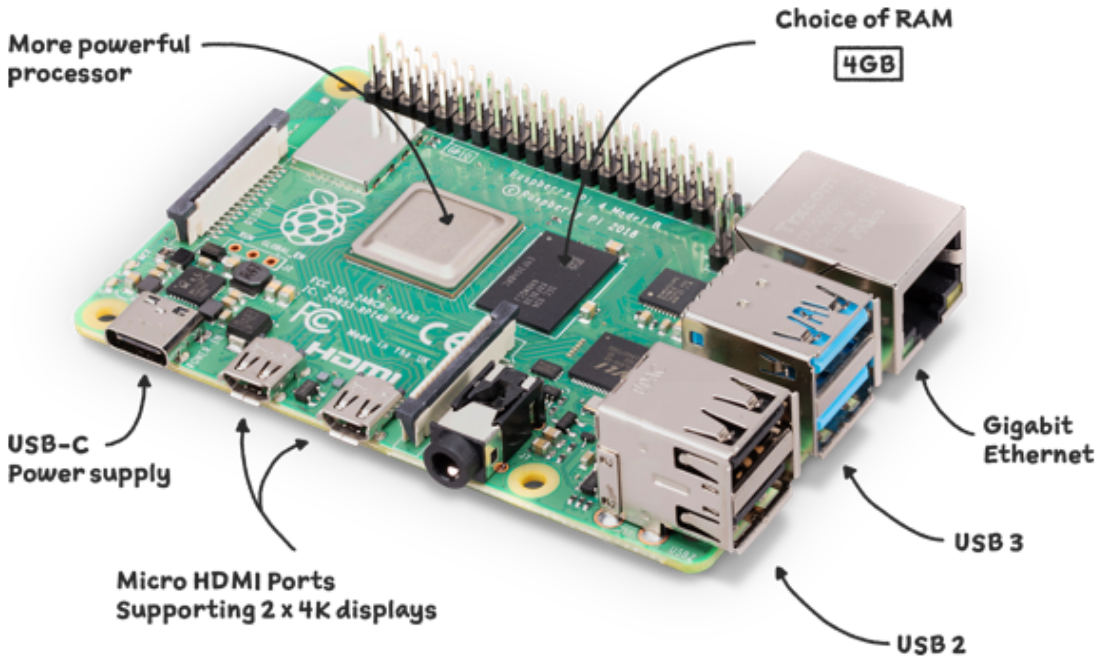
而进程所持有所有文件可以通过current→files→fdt获取。再基于fd做一个遍历，便可以得到该进程的所有打开文件。其中区别具体哪个文件是socket相关的，可以进一步通过file指向的inode的i_mode判断是否设置了S_IFSOCK标志位，来判断是否是socket相关的文件。

```
        files = current->files;

    fdt = files->fdt;
    max_fds = fdt->max_fds;

    printk("-----------\n");

    for (i = 0; i < max_fds; i++) {
        if (fd_is_open(i, fdt)) {

            f = fdt->fd[i];

            f_name = f->f_path.dentry->d_iname;
            f_inode = f->f_inode;
            if (f_inode->i_mode & S_IFSOCK) {
                is_sock = 1;
            } else {
                is_sock = 0;
            }
            printk("Pid: %d filename: %s Sock: %s\n", current->pid, f_name,
is_sock ? "YES" : "NO");

        }
    }
```

4. 系统测试

4.1 测试环境

本次实验测试环境采用的是 `Raspberry Pi 4`：



详细信息如下所示：

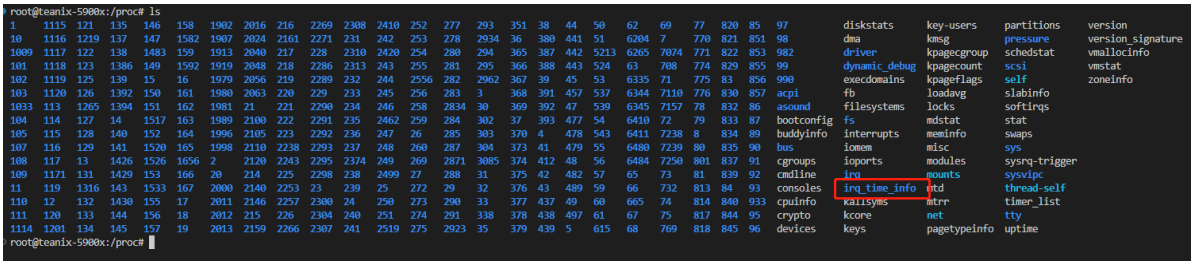
```
#kernel version
Linux shi-Rspi 5.15.0-1008-raspi #8-Ubuntu SMP PREEMPT
aarch64 aarch64 aarch64 GNU/Linux

#CPU
Architecture:            aarch64
  CPU op-mode(s):        32-bit, 64-bit
  Byte Order:            Little Endian
CPU(s):                  4
  On-line CPU(s) list:   0-3
Vendor ID:               ARM
  Model name:            Cortex-A72
    Model:               3
    Thread(s) per core:  1
    Core(s) per cluster: 4
    Socket(s):           -
    Cluster(s):          1
    Stepping:            r0p3
    CPU max MHz:         1500.0000
    CPU min MHz:         600.0000
    BogoMIPS:            108.00
    Flags:               fp asimd evtstrm crc32 cpuid
Vulnerabilities:
  Itlb multihit:         Not affected
  L1tf:                  Not affected
  Mds:                   Not affected
```

```
  Meltdown:               Not affected
  Spec store bypass:      Vulnerable
  Spectre v1:             Mitigation; __user pointer sanitization
  Spectre v2:             Vulnerable
  Srbds:                  Not affected
  Tsx async abort:        Not affected

#disk
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           379M  3.6M  376M   1% /run
/dev/mmcblk0p2   15G   11G  3.3G  77% /
tmpfs           1.9G     0  1.9G   0% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
/dev/mmcblk0p1  253M  120M  133M  48% /boot/firmware
tmpfs           379M   76K  379M   1% /run/user/127
tmpfs           379M   68K  379M   1% /run/user/1000

#mem
               total        used        free      shared  buff/cache
available
Mem:           3.7Gi       682Mi       1.3Gi       5.0Mi       1.7Gi
2.9Gi
Swap:          1.0Gi          0B       1.0Gi
```

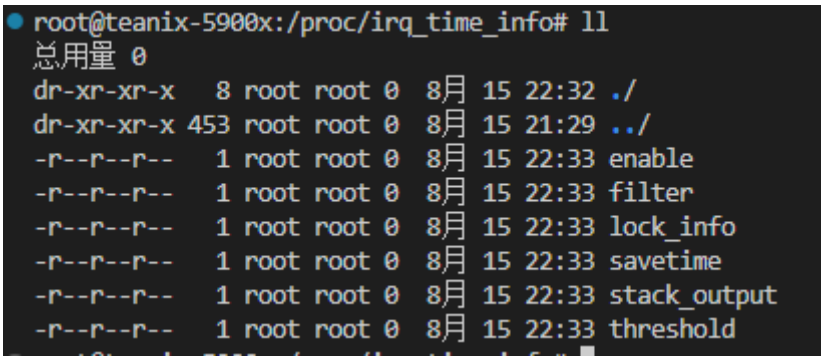
4.2 测试指标及测试结果

4.2.1 procfs下各节点

/proc 下父级目录 - irq_time_info



各子节点



4.2.1 模块总开关

模块总开关默认为 `0` 关闭状态，可以通过向其中写入 `1` 来打开模块

```
root@teanix-5900x:/proc/irq_time_info# echo 1 > enable
```

4.2.2 系统关中断信息

可以通过 `lock_info` 查看抓取到的数据的大概展示：

```
root@teanix-5900x:/proc/irq_time_info# cat lock_info
Lock Info:
pid: 769    cpu: 7    comm: systemd-oomd     exe: systemd-oomd
Files:
[00] - UNIX-STREAM
Locks:
total_nums:6  total_times:595036
addr:ffff9fe386906000    t1:4057105424402        nums:6          times:595036
-------------------------------------------------

Lock Info:
pid: 6411   cpu: 19   comm: kworker/u64:3    exe: Null
Locks:
total_nums:2  total_times:177062
addr:ffff9fe380051000    t1:4057265720758        nums:2          times:177062
-------------------------------------------------
```

然后输入相应的 `pid`、 `cpu` 、`lock_addr` 可以得到详细的调用栈信息：

```
root@teanix-5900x:/proc/irq_time_info# echo 769 7 ffff9fe386906000 > filter
root@teanix-5900x:/proc/irq_time_info#
root@teanix-5900x:/proc/irq_time_info#
root@teanix-5900x:/proc/irq_time_info# cat stack_output
pid: 769    cpu: 7    comm: systemd-oomd     exe: systemd-oomd
Call Trace:
[00]-[00000000578792aa] kprobe_ftrace_handler+0xf6/0x1c0
[01]-[00000000b683fbe8] 0xffffffffc04e90e3
[02]-[0000000089f15c51] _raw_spin_unlock_irqrestore+0x1/0x30
[03]-[0000000043ab72fc] __mem_cgroup_flush_stats+0x62/0xa0
[04]-[00000000694f6129] memory_stat_format+0x34d/0x410
[05]-[00000000eadeb788] memory_stat_show+0x21/0x50
[06]-[00000000b16887d4] cgroup_seqfile_show+0x68/0x110
[07]-[000000002b6c1ea2] kernfs_seq_show+0x27/0x30
[08]-[000000008a83a4b4] seq_read_iter+0x124/0x4b0
[09]-[0000000067c3959a] kernfs_fop_read_iter+0x30/0x40
[10]-[00000000c1881c5a] new_sync_read+0x110/0x1a0
[11]-[000000003c557a09] vfs_read+0x103/0x1a0
[12]-[0000000001124890] ksys_read+0x67/0xf0
[13]-[000000007e1adede] __x64_sys_read+0x19/0x20
[14]-[000000001bd19e4b] do_syscall_64+0x5c/0xc0
[15]-[000000008107aaae] entry_SYSCALL_64_after_hwframe+0x61/0xcb
```

4.2.3 指定阈值

阈值默认为 `1000 ns` ，可以通过向其中写入`整数ns` 的方式修改阈值

```
root@shi-Rspi:/proc/irq_time_info# cat threshold_ns
1000
root@shi-Rspi:/proc/irq_time_info# echo 27000000 > threshold_ns
root@shi-Rspi:/proc/irq_time_info#
root@shi-Rspi:/proc/irq_time_info# cat threshold_ns
27000000
root@shi-Rspi:/proc/irq_time_info#
```

5. 总结

5.1 项目开发进展

日期	进展
04.15 - 04.22	查看 proc 的实现，学习proc的接口API及其使用
04.22 - 04.29	查看kernel documents 及博客 等关于 kprobes机制的分析，尝试编写 kprobes demo
04.29 - 05.06	分析中断流程的过程，寻找挂载点
05.06 - 5.13	整合kprobes关于中断挂载及procfs交互的代码，完善相关功能，修改bug
05.13 - 05.20	实现stack_trace 及持有锁 等功能
05.20 - 05.27	实现文件&socket等信息的提取
05.27 - 6.03	整理前面学习文章，撰写初赛文档
06.03 - 07.15	修改相关项目代码，加入xarry、objpool以及锁同步等内容
07.15 - 08.15	完善代码并撰写结项文档

5.2 遇到的问题和解决办法

1. 中断挂载点问题

> 在刚开始分析中断挂载点时，主要是想借鉴`ftrace`实现的功能，因为`ftrace`在这反面算是已经比较成熟了，`ftrace`有一些`tracepoint`的固定点，但是题目要求我们使用内核模块来完成这些信息的获取，所以想到了两个解决办法：
> - 在内核模块中实现`tracepoint`的使用
> - 分析`tracepoint`挂载点的位置，查看在其函数执行流程上下文能不能找到一些`kprobes`可以挂载的点
>
> 经过分析验证，发现第二个方案还是比较可行的。

2. stack_trace输出问题

> 因为题目要输出相关进程的函数调用栈关系，一般来说在内核中打印函数调用栈的接口是 `dump_stack`但是，这个接口并没有提供入参之类的东西，它直接会将调用栈关系打印到`dmesg`中去，但是我们需要的是将其入队到`kfifo`中，本来想的办法是分析`dump_stack`的实现过程，移植修改该接口到我们的内核模块当中，后来发现这个还是比较有难度，为了避免耽搁项目的进度，后来通过和导师开会讨论，最终决定使用`stack_trace_save`这个接口

6. 项目使用说明

6.1 环境配置

确保开启以下编译选项：

```
CONFIG_KPROBES=y
CONFIG_MODULES=y
CONFIG_MODULE_UNLOAD=y
CONFIG_KALLSYMS=y
CONFIG_KALLSYMS_AL=y
CONFIG_DEBUG_INFO=y
```

6.2 克隆并运行

克隆仓库：

```
git clone https://gitlab.eduxiji.net/vegeta/project788067-126085.git
cd project788067-126085
```

编译及插入模块

```
make
insmod irq_mod.ko
```

查看实验结果

```
cd /proc/irq_info
# enable  模块开关 0-关闭 1-启用
# threshold 阈值大小
# output 信息输出

eg:
echo 1 > enable       #开启模块
echo 2000 > threshold #更改合适阈值大小
cat lock_info #查看粗粒度的锁信息
echo 769 7 ffff9fe386906000 > filter #输入需要查看的pid、CPU、锁地址等数据
cat stack_output #查看详细的调用栈信息
```