



西安邮电大学

1. 项目简介与基本情况

- 1.1 概述
- 1.2 项目成员
- 1.3 项目架构
- 1.4 仓库目录结构
- 1.5 项目开发进展

2. 系统需求分析与设计

- 2.1 procfs分析与设计
 - 2.1.1 proc概述
 - 2.1.2 节点设计
- 2.2 cache存储机制
 - 2.2.1 双链表
 - 2.2.2 kfifo
- 2.3 kprobes 机制
 - 2.3.1 kprobe
 - 2.3.2 kretprobe
- 2.4 中断挂载点分析
 - 2.4.1 关闭 local cpu
 - 2.4.2 关闭中断线
- 2.5 调用栈获取
 - 2.5.1 dump_stack
 - 2.5.2 stack_trace_save
- 2.6 持有锁分析
 - 2.6.1 持有锁具体路径
 - 2.6.2 持有锁类别
- 2.7 文件和socket

3. 系统实现

- 3.1 kprobe挂载程序实现
 - 3.1.1 API接口
 - 3.1.1.1 注册kprobe探针
 - 3.1.1.2 注册kprobe探针
 - 3.1.1.3 注销探针
 - 3.1.1.4 前处理函数
 - 3.1.1.5 后处理函数
 - 3.1.1.6 错误处理函数
 - 3.1.1.7 禁用探针
 - 3.1.1.8 启用探针
 - 3.1.2 具体实现
- 3.2 procfs交互
 - 3.2.1 procfs API
 - 3.2.2 具体实现
- 3.3 双链表存储实现
- 3.4 kfifo缓存实现
 - 3.4.1 API
 - 3.4.2 具体实现
- 3.5 调用栈打印实现
- 3.6 进程其他信息获取实现

- 4. 系统测试
 - 4.1 测试环境
 - 4.2 测试指标及测试结果
 - 4.2.1 procs下各节点
 - 4.2.1 模块总开关
 - 4.2.2 系统全部关中断时长(ns级)
 - 4.2.3 指定阈值
- 5. 总结
 - 5.1 项目开发进展
 - 5.2 遇到的问题和解决办法
- 6. 项目使用说明
 - 6.1 环境配置
 - 6.2 克隆并运行

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

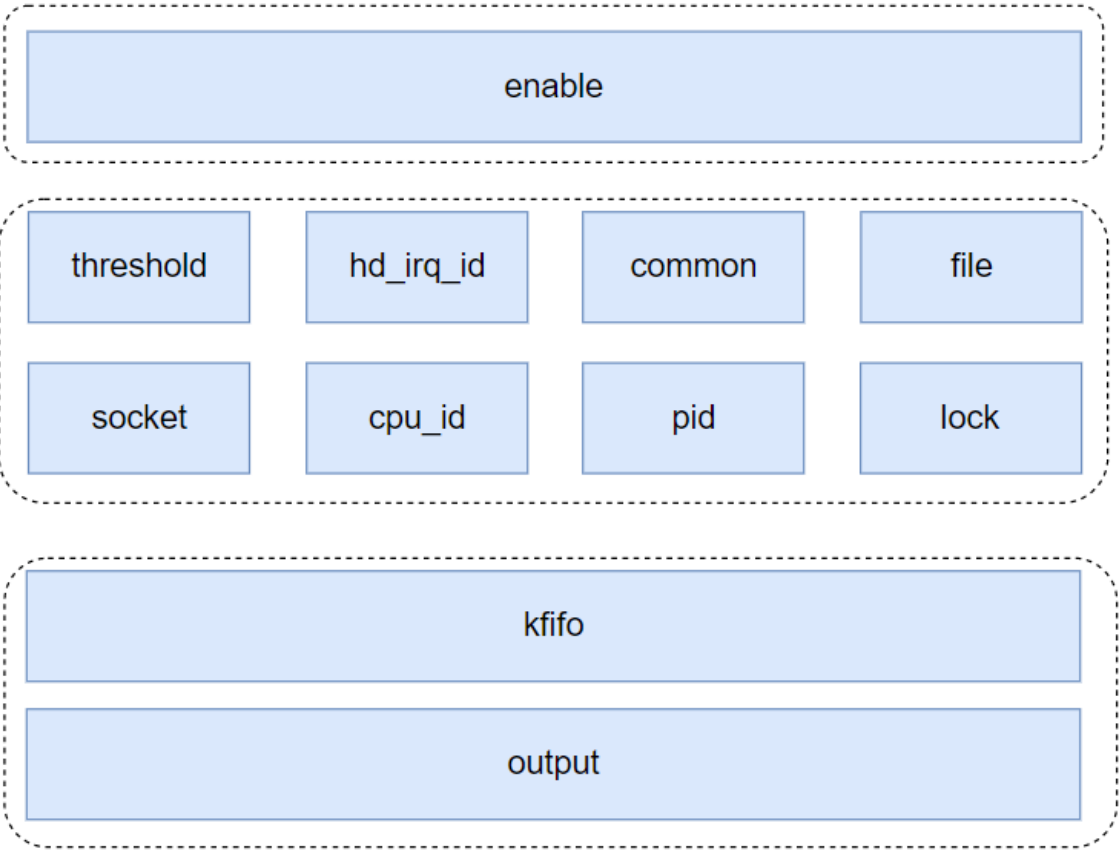
指导老师：郭浩

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

```
1 project788067-126085/
2 |— docs           文档
3 |— submod         子模块
4 |— tmp            临时文件
5 |— irq.c          挂载点
6 |— irqsave.c
7 |— hd_irq_id.c
8 |— enable.c
9 |— threshold.c
10 |— output.c
11 |— README.md
12 |— trace.h
13 |— main.c
14 |— Makefile
```

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90%	开机启动目前还有一点点的小bug
第二题：探针工具实际内容实现	已实现85%	
第三题：工具稳定性验证和实际效果测试	已实现50%	目前暂未进行在高负载场景下的稳定性测试

第一题：内核模块基础框架实现 (已实现90%)

- ☒ 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
- ☒ 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
- ☒ 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
- ☐ 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85%)

以题目一为基础，工具需要追加下列功能：

- ☒ 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
- ☒ 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
- ☒ 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
- ☒ 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50%)

以题目二为基础，工具需要追加下列功能：

- ☒ 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
- ☐ 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procfs的交互，proc是Linux 内核提供了一种虚拟文件系统，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。

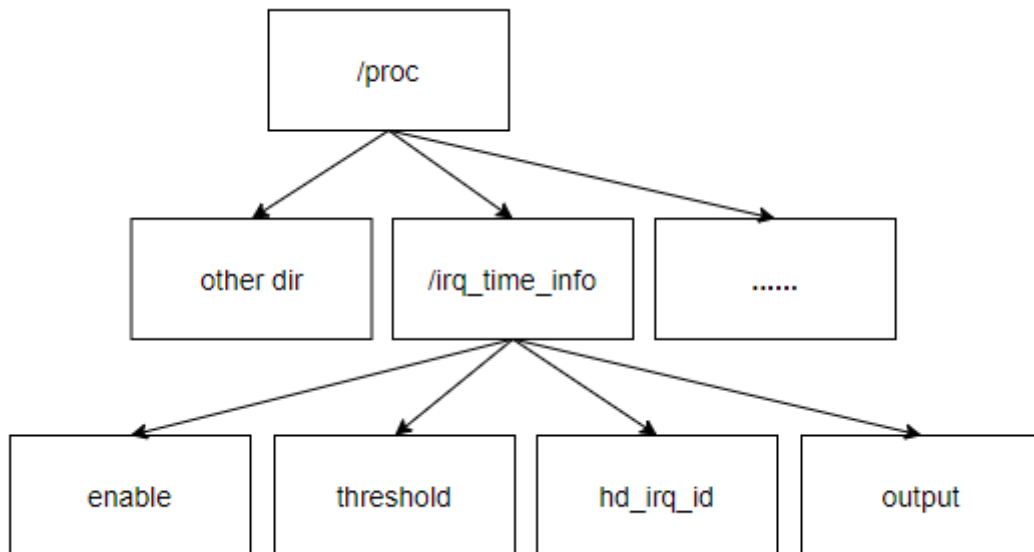
事实上，ps、top等命令就是直接从proc文件系统中读取信息的。

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
- hd_irq_id 为硬件中断号参数，可以通过指定该参数来获取特定的硬件中断
- output 为抓取到的所有数据，可以通过 cat 该节点来得到所有监测的结果

2.2 cache存储机制

cache机制存储目前的设计思路有两种：

- 基于双链表的cache机制存储
- 基于kfifo的cache机制存储。

2.2.1 双链表

第一种思路使用内核提供的最基本数据结构中包括双链表struct list_head。

```
1 struct list_head {
2     struct list_head *next, *prev;
3 };
```

双链表通常是作为其余数据结构的成员，用于组织数据与数据之间的关系。

在限定了cache大小的情况下，我们将所有的数据使用链表进行存储。

当数据量大小要超出cache大小的情况下，可以对当前最“旧”的数据进行替换，以达到cache存储的思想。

在进行数据输出和数据替换时，主要用到的接口函数有container_of以及基于container_of延申的链表遍历函数list_for_each_entry

```
1 #define container_of(ptr, type, member) ({  
2     void *__mptr = (void *)(ptr);  
3     BUILD_BUG_ON_MSG(!__same_type(*(ptr), ((type *)0)->member) &&  
4         !__same_type(*(ptr), void),  
5         "pointer type mismatch in container_of()");  
6     ((type *)__mptr - offsetof(type, member))); }
```

container_of(ptr, type, member)中的ptr标识结构体中member的地址，type标识结构体类型，member表示结构体成员。container_of通过结构体中成员member的指针获得该结构体的指针。

2.2.2 kfifo

第二种思路使用内核提供的一种First In First Out数据结构kfifo，特点如下：

- 采用环形缓冲区来实现，提供一个无边界的字节流服务。采用环形缓冲区的好处为，当一个数据元素被用掉后，其余数据元素不需要移动其存储位置，从而减少拷贝提高效率。
- 保证缓冲区大小为2的次幂，不是的向上取整为2的次幂（很重要）。
- 使用无符号整数保存输入(in)和输出(out)的位置，在输入输出时不对in和out的值进行模运算，而让其自然溢出，并能够保证in-out的结果为缓冲区中已存放的数据长度。
- 将需要取模的运算用 & 操作代替 ($a \% size = (a \& (size - 1))$), 这需要size保证为2的次幂。
- 使用内存屏障(Memory Barrier)技术，实现单消费者和单生产者对kfifo的无锁并发访问（包括多CPU的情况），多个消费者、生产者的并发访问还是需要加锁的。

其结构体实现如下所示：

```
1 struct __kfifo {  
2     unsigned int    in;  
3     unsigned int    out;  
4     unsigned int    mask;  
5     unsigned int    esize;  
6     void            *data;  
7 };
```

in和out分别为入队列和出队列，in为下次入队列的位置，out为下次出队列的位置，当in值等于out值时表示队列为空，esize为空间大小。

2.3 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

2.3.1 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 nmissd; /*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。

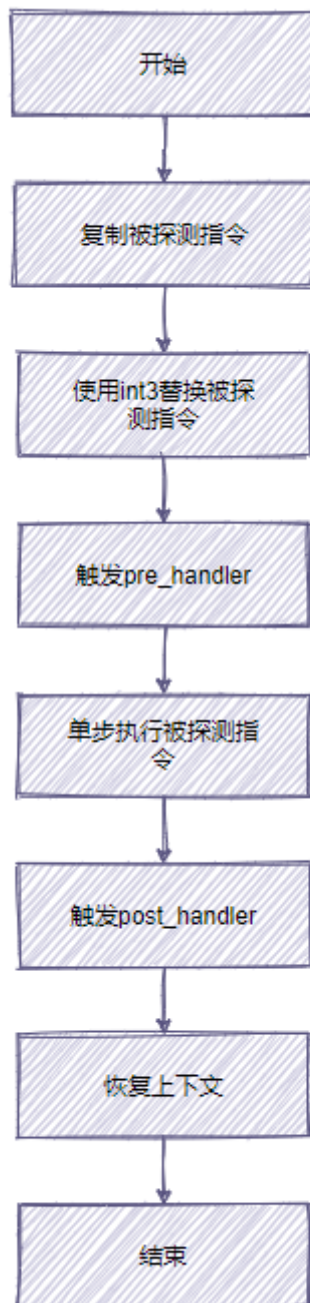
Linux内核中各个子系统相互依赖，当其中某个子系统状态发生改变时，就必须使用一定的机制告知使用其服务的其他子系统，以便其他子系统采取相应的措施。为满足这样的需求，内核实现了事件通知链机制（`notifier_call_chain`）

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：



2.3.2 kretprobe

下面是kretprobe的结构体：

```
1 struct kretprobe {  
2     struct kprobe kp;  
3     kretprobe_handler_t handler;  
4     kretprobe_handler_t entry_handler;  
5     int maxactive;  
6     int nmissed;  
7     size_t data_size;  
8     struct hlist_head free_instances;  
9     raw_spinlock_t lock;  
10 };
```

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。

2.4 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

2.4.1 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(flags)` 保存当前中断状态并关闭中断
- `local_irq_restore(flags)` 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：

```
1 //include/linux/irqflags.h
2 #define local_irq_enable() do { raw_local_irq_enable(); } while (0)
3 #define local_irq_disable() do { raw_local_irq_disable(); } while (0)
4 #define local_irq_save(flags) do { raw_local_irq_save(flags); } while (0)
5 #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：

```
1 //include/linux/irqflags.h
2 #define raw_local_irq_disable() arch_local_irq_disable()
3 #define raw_local_irq_enable() arch_local_irq_enable()
4 #define raw_local_irq_save(flags)
5     do {
6         typecheck(unsigned long, flags);
7         flags = arch_local_irq_save();
8     } while (0)
9 #define raw_local_irq_restore(flags)
10    do {
```

```

11     typecheck(unsigned long, flags);
12     raw_check_bogus_irq_restore();
13     arch_local_irq_restore(flags);
14 } 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：

```

1 //arch/arm64/include/asm/irqflags.h
2 static inline void arch_local_irq_disable(void)
3 {
4     if (system_has_prio_mask_debugging()) {
5         u32 pmr = read_sysreg_s(SYS_ICC_PMR_EL1);
6
7         WARN_ON_ONCE(pmr != GIC_PRIO_IRQON && pmr != GIC_PRIO_IRQOFF);
8     }
9
10    asm volatile(ALTERNATIVE(
11        "msr    daifset, #3    // arch_local_irq_disable",
12        __msr_s(SYS_ICC_PMR_EL1, "%0"),
13        ARM64_HAS_IRQ_PRIO_MASKING)
14        :
15        : "r" ((unsigned long) GIC_PRIO_IRQOFF)
16        : "memory");
17 }

```

`arch_local_irq_enable` 的实现方式如下：

```

1 //arch/arm64/include/asm/irqflags.h
2 static inline void arch_local_irq_enable(void)
3 {
4     if (system_has_prio_mask_debugging()) {
5         u32 pmr = read_sysreg_s(SYS_ICC_PMR_EL1);
6
7         WARN_ON_ONCE(pmr != GIC_PRIO_IRQON && pmr != GIC_PRIO_IRQOFF);
8     }
9
10    asm volatile(ALTERNATIVE(
11        "msr    daifclr, #3    // arch_local_irq_enable",
12        __msr_s(SYS_ICC_PMR_EL1, "%0"),
13        ARM64_HAS_IRQ_PRIO_MASKING)
14        :
15        : "r" ((unsigned long) GIC_PRIO_IRQON)
16        : "memory");
17
18    pmr_sync();
19 }

```

`arch_local_irq_save` 的实现方式如下：

```

1 static inline unsigned long arch_local_irq_save(void)
2 {
3     unsigned long flags;
4
5     flags = arch_local_save_flags();

```

```

6
7     /*
8     * There are too many states with IRQs disabled, just keep the current
9     * state if interrupts are already disabled/masked.
10    */
11    if (!arch_irqs_disabled_flags(flags))
12        arch_local_irq_disable();
13
14    return flags;
15 }

```

`arch_local_irq_restore` 的实现方式如下:

```

1  static inline void arch_local_irq_restore(unsigned long flags)
2  {
3      asm volatile(ALTERNATIVE(
4          "msr    daif, %0",
5          __msr_s(SYS_ICC_PMR_EL1, "%0"),
6          ARM64_HAS_IRQ_PRIO_MASKING)
7          :
8          : "r" (flags)
9          : "memory");
10
11      pmr_sync();
12  }

```

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。

2.4.2 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()`:

```

1  disable_irq
2      --> synchronize_irq

```

主要函数实现如下:

```

1  //kernel/irq/manage.c
2  void disable_irq(unsigned int irq)
3  {
4      if (!__disable_irq_nosync(irq))
5          synchronize_irq(irq);
6  }
7  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：

```
1 disable_irq_nosync
2     -->__disable_irq_nosync
3         -->__disable_irq
4             -->irq_disable
5                 -->__irq_disable
6                     -->mask_irq
```

主要函数实现如下：

```
1 //kernel/irq/manage.c
2 void disable_irq_nosync(unsigned int irq)
3 {
4     __disable_irq_nosync(irq);
5 }
6 EXPORT_SYMBOL(disable_irq_nosync);
7 -----
8 static int __disable_irq_nosync(unsigned int irq)
9 {
10     unsigned long flags;
11     struct irq_desc *desc = irq_get_desc_buslock(irq, &flags,
12     IRQ_GET_DESC_CHECK_GLOBAL);
13
14     if (!desc)
15         return -EINVAL;
16     __disable_irq(desc);
17     irq_put_desc_busunlock(desc, flags);
18     return 0;
19 }
```

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：

```
1 enable_irq
2     -->__enable_irq
3         -->irq_startup
4             -->irq_enable
5                 -->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：

```
1 //kernel/irq/manage.c
2 void enable_irq(unsigned int irq)
3 {
4     unsigned long flags;
5     struct irq_desc *desc = irq_get_desc_buslock(irq, &flags,
6     IRQ_GET_DESC_CHECK_GLOBAL);
7
8     if (!desc)
```

```

8         return;
9         if (WARN(!desc->irq_data.chip,
10             KERN_ERR "enable_irq before setup/request_irq: irq %u\n", irq))
11             goto out;
12
13         __enable_irq(desc);
14     out:
15         irq_put_desc_busunlock(desc, flags);
16     }
17     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：

```

1 synchronize_irq
2     -->__synchronize_hardirq
3         -->raw_spin_lock_irqsave
4             |irqd_irq_inprogress(&desc->irq_data);|
5         -->raw_spin_unlock_irqrestore

```

具体函数实现如下：

```

1 //kernel/irq/manage.c
2 void synchronize_irq(unsigned int irq)
3 {
4     struct irq_desc *desc = irq_to_desc(irq);
5
6     if (desc) {
7         __synchronize_hardirq(desc, true);
8         /*
9          * We made sure that no hardirq handler is
10          * running. Now verify that no threaded handlers are
11          * 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()

2.5 调用栈获取

2.5.1 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：

```
1 //arch/arm64/kernel/stacktrace.c
2 void show_stack(struct task_struct *tsk, unsigned long *sp, const char
   *log_lvl)
3 {
4     dump_backtrace(NULL, tsk, log_lvl);
5     barrier();
6 }
```

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

```
1 void dump_backtrace(struct pt_regs *regs, struct task_struct *tsk, const char
   *log_lvl)
2 {
3     struct stackframe frame;
4     int skip = 0;
5
6     pr_debug("%s(regs = %p tsk = %p)\n", __func__, regs, tsk);
7
8     if (regs) {
9         if (user_mode(regs))
10             return;
11         skip = 1;
```

```

12     }
13
14     if (!tsk)
15         tsk = current;
16
17     if (!try_get_task_stack(tsk)) //如果获取失败
18         return;
19
20     if (tsk == current) {
21         start_backtrace(&frame,
22             (unsigned long)__builtin_frame_address(0),
23             (unsigned long)dump_backtrace);
24     } else {
25         /*
26          * task blocked in __switch_to
27          */
28         start_backtrace(&frame,
29             thread_saved_fp(tsk),
30             thread_saved_pc(tsk));
31     }
32
33     printk("%sCall trace:\n", loglvl);
34     do {
35         /* skip until specified stack frame */
36         if (!skip) {
37             dump_backtrace_entry(frame.pc, loglvl);
38         } else if (frame.fp == regs->regs[29]) {
39             skip = 0;
40             /*
41              * Mostly, this is the case where this function is
42              * called in panic/abort. As exception handler's
43              * stack frame does not contain the corresponding pc
44              * at which an exception has taken place, use regs->pc
45              * 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

```

2.5.2 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，这并不是我们所需要的。

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：

```

1  // #ifdef CONFIG_HAVE_RELIABLE_STACKTRACE
2  /**
3   * stack_trace_save - Save a stack trace into a storage array
4   * @store: Pointer to storage array
5   * @size: Size of the storage array
6   * @skipnr: Number of entries to skip at the start of the stack trace
7   *
8   * Return: Number of trace entries stored.
9   */
10 unsigned int stack_trace_save(unsigned long *store, unsigned int size,
11                               unsigned int skipnr)
12 {
13     stack_trace_consume_fn consume_entry = stack_trace_consume_entry;
14     struct stacktrace_cookie c = {
15         .store = store,
16         .size = size,
17         .skip = skipnr + 1,
18     };
19
20     arch_stack_walk(consume_entry, &c, current, NULL);

```



```

21     return c.len;
22 }
23 EXPORT_SYMBOL_GPL(stack_trace_save);

```

其中这个 `stacktrace_cookie` 结构体是这样定义的：

```

1 struct stacktrace_cookie {
2     unsigned long *store;
3     unsigned int size;
4     unsigned int skip;
5     unsigned int len;
6 };

```

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：

```

1 noinline notrace void arch_stack_walk(stack_trace_consume_fn consume_entry,
2                                     void *cookie, struct task_struct *task,
3                                     struct pt_regs *regs)
4 {
5     struct stackframe frame;
6
7     if (regs)
8         start_backtrace(&frame, regs->regs[29], regs->pc);
9     else if (task == current)
10        start_backtrace(&frame,
11                      (unsigned long)__builtin_frame_address(1),
12                      (unsigned long)__builtin_return_address(0));
13    else
14        start_backtrace(&frame, thread_saved_fp(task),
15                      thread_saved_pc(task));
16
17    walk_stackframe(task, &frame, consume_entry, cookie);
18 }

```

2.6 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

2.6.1 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()`。

2.6.2 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;
20         break;
21     default:
22         val = 0;
23     }
24
25     return val;
26 }
```

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。

2.7 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：

```

1  /* Filesystem information: */
2  struct fs_struct      *fs;
3
4  /* Open file information: */
5  struct files_struct   *files;

```

其中 `files_struct` 的相关结构体实现如下：

```

1  struct files_struct {
2      /*
3       * read mostly part
4       */
5      atomic_t count;
6      bool resize_in_progress;
7      wait_queue_head_t resize_wait;
8
9      struct fdtable __rcu *fdt;
10     struct fdtable fdtab;
11     /*
12      * written part on a separate cache line in SMP
13      */
14     spinlock_t file_lock ____cacheline_aligned_in_smp;
15     unsigned int next_fd;
16     unsigned long close_on_exec_init[1];
17     unsigned long open_fds_init[1];
18     unsigned long full_fds_bits_init[1];
19     struct file __rcu * fd_array[NR_OPEN_DEFAULT];
20 };
21
22 struct fdtable {
23     unsigned int max_fds;
24     struct file __rcu **fd;      /* current fd array */
25     unsigned long *close_on_exec;
26     unsigned long *open_fds;
27     unsigned long *full_fds_bits;
28     struct rcu_head rcu;
29 };

```

其中，具体的打开文件信息交由struct fdtable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。

```

1  struct file {
2      union {
3          struct llist_node  fu_llist;
4          struct rcu_head    fu_rcuhead;
5      } f_u;
6      struct path    f_path;
7      struct inode    *f_inode;    /* cached value */
8      const struct file_operations *f_op;
9      .....
10 }
11
12 static inline bool fd_is_open(unsigned int fd, const struct fdtable *fdt)
13 {
14     return test_bit(fd, fdt->open_fds);

```

```
15 } }
```

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。

```
1 struct path {
2     struct vfsmount *mnt;
3     struct dentry *dentry;
4 } __randomize_layout;
5
6 struct dentry {
7     /* RCU lookup touched fields */
8     unsigned int d_flags; /* protected by d_lock */
9     seqcount_spinlock_t d_seq; /* per dentry seqlock */
10    struct hlist_bl_node d_hash; /* lookup hash list */
11    struct dentry *d_parent; /* parent directory */
12    struct qstr d_name;
13    struct inode *d_inode; /* where the name belongs to - NULL is
14                          * negative */
15    unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
16    .....
17 }
```

可以看到struct dentry中有两个成员，一个是struct qstr d_name，另外一个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。

```
1 struct qstr {
2     union {
3         struct {
4             HASH_LEN_DECLARE;
5         };
6         u64 hash_len;
7     };
8     const unsigned char *name;
9 };
```

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kmallocc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

```
1 struct socket *sock_alloc(void)
2 {
3     struct inode *inode;
4     struct socket *sock;
5
6     inode = new_inode_pseudo(sock_mnt->mnt_sb);
7     if (!inode)
8         return NULL;
9
10    sock = SOCKET_I(inode);
```

```

11
12     inode->i_ino = get_next_ino();
13     inode->i_mode = S_IFSOCK | S_IRWXUGO;
14     inode->i_uid = current_fsuid();
15     inode->i_gid = current_fsgid();
16     inode->i_op = &sockfs_inode_ops;
17
18     return sock;
19 }
20 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与文件的绑定函数sock_map_fd()。

```

1  static int sock_map_fd(struct socket *sock, int flags)
2  {
3      struct file *newfile;
4      int fd = get_unused_fd_flags(flags);
5      if (unlikely(fd < 0)) {
6          sock_release(sock);
7          return fd;
8      }
9
10     newfile = sock_alloc_file(sock, flags, NULL);
11     if (!IS_ERR(newfile)) {
12         fd_install(fd, newfile);
13         return fd;
14     }
15
16     put_unused_fd(fd);
17     return PTR_ERR(newfile);
18 }
19
20 struct file *sock_alloc_file(struct socket *sock, int flags, const char
 *dname)
21 {
22     struct file *file;
23
24     if (!dname)
25         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socket_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

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。

```
1 #include <linux/kprobes.h>
2 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

```
1 #include <linux/kprobes.h>
2 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

```
1 #include <linux/kprobes.h>
2 void unregister_kprobe(struct kprobe *kp);
3 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

```
1 #include <linux/kprobes.h>
2 #include <linux/ptrace.h>
3 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

```
1 #include <linux/kprobes.h>
2 #include <linux/ptrace.h>
3 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

```
1 #include <linux/kprobes.h>
2 #include <linux/ptrace.h>
3 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

```
1 #include <linux/kprobes.h>
2 int disable_kprobe(struct kprobe *kp);
3 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

```
1 #include <linux/kprobes.h>
2 int enable_kprobe(struct kprobe *kp);
3 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	<code>raw_spin_lock_irq()</code>	<code>raw_spin_unlock_irq()</code>
2	<code>raw_spin_lock_irqsave()</code>	<code>raw_spin_lock_irqrestore()</code>
3	<code>mask_irq()</code>	<code>unmask_irq()</code>

其中，挂载点表示为：

```
1 static char symbol_lock_irq[MAX_SYMBOL_LEN] = "_raw_spin_lock_irq";
2 static char symbol_unlock_irq[MAX_SYMBOL_LEN] = "_raw_spin_unlock_irq";
3
4 static char symbol_lock_irqsave[MAX_SYMBOL_LEN] = "_raw_spin_lock_irqsave";
5 static char symbol_unlock_irqrestore[MAX_SYMBOL_LEN] =
6     "_raw_spin_unlock_irqrestore";
7
7 static char symbol_mask_irq[MAX_SYMBOL_LEN] = "mask_irq";
8 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：

```
1 static struct kprobe kp_lock_irq = {
2     .symbol_name = symbol_lock_irq,
3 };
4
5 static struct kprobe kp_unlock_irq = {
6     .symbol_name = symbol_unlock_irq,
7 };
8
9 static struct kprobe kp_lock_irqsave = {
10     .symbol_name = symbol_lock_irqsave,
11 };
12
13 static struct kprobe kp_unlock_irqrestore = {
14     .symbol_name = symbol_unlock_irqrestore,
15 };
16
17 static struct kprobe kp_mask_irq = {
18     .symbol_name = symbol_mask_irq,
19 };
20
21 static struct kprobe kp_unmask_irq = {
22     .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。

```

1 struct disable_irq_entry {
2     int flag;
3     u64 entry_time;
4     int irq;
5 };
6
7 static struct disable_irq_entry __percpu *pdie_lock_irq;
8 static struct disable_irq_entry __percpu *pdie_lock_irqsave;
9 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。

```

1 static int __init kprobe_init(void)
2 {
3     struct disable_irq_entry *ptrdie_lock_irq, ptrdie_lock_irqsave,
4     ptrdie_mask_irq;
5
6     pdie_lock_irq = alloc_percpu(struct disable_irq_entry);
7     pdie_lock_irqsave = alloc_percpu(struct disable_irq_entry);
8     pdie_mask_irq = alloc_percpu(struct disable_irq_entry);
9
10    if (!pdie_lock_irq || !pdie_lock_irqsave || !pdie_mask_irq) {
11        pr_err("alloc_percpu failed\n");
12        return 0;
13    }
14
15    for_each_online_cpu(i) {
16        ptrdie_lock_irq = per_cpu_ptr(pdie_lock_irq, i);
17        ptrdie_lock_irq->flag = 0;
18
19        ptrdie_lock_irqsave = per_cpu_ptr(pdie_lock_irqsave, i);
20        ptrdie_lock_irqsave->flag = 0;
21
22        ptrdie_mask_irq = per_cpu_ptr(pdie_mask_irq, i);
23        ptrdie_mask_irq->flag = 0;
24    }
25
26    kp_lock_irq.pre_handler = handler_lock_irq;
27    kp_unlock_irq.pre_handler = handler_unlock_irq;
28
29    kp_lock_irqsave.pre_handler = handler_lock_irqsave;
30    kp_unlock_irqrestore.pre_handler = handler_unlock_irqrestore;
```

```

31     kp_mask_irq.pre_handler = handler_mask_irq;
32     kp_unmask_irq.pre_handler = handler_unmask_irq;
33
34     .....
35 }

```

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

```

1  static int __kprobes handler_lock_irq(struct kprobe *p, struct pt_regs
   *regs)
2  {
3      u32 cpu = smp_processor_id();
4      u64 now = ktime_get_ns();
5      struct disable_irq_entry *entry_pos;
6
7      entry_pos = per_cpu_ptr(pdie, cpu);
8      entry_pos->entry_time = now;
9      entry_pos->flag = 1;
10     entry_pos->irq = -1;
11
12     return 0;
13 }
14
15 static int __kprobes handler_unlock_irq(struct kprobe *p, struct pt_regs
   *regs)
16 {
17     u32 cpu = smp_processor_id();
18     u64 now = ktime_get_ns();
19     u64 delta = 0;
20
21     struct disable_irq_entry *entry_pos;
22
23     entry_pos = per_cpu_ptr(pdie, cpu);
24     entry_pos->flag = 0;
25     delta = now - entry_pos->entry_time;
26
27     .....
28 }

```

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。

```

1  static inline struct proc_dir_entry *proc_mkdir(const char *name,
2      struct proc_dir_entry *parent) {return NULL;}
3  #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。

```

1 struct proc_ops {
2     unsigned int proc_flags;
3     int (*proc_open)(struct inode *, struct file *);
4     ssize_t (*proc_read)(struct file *, char __user *, size_t, loff_t *);
5     ssize_t (*proc_read_iter)(struct kiocb *, struct iov_iter *);
6     ssize_t (*proc_write)(struct file *, const char __user *, size_t, loff_t
    *);
7     loff_t (*proc_llseek)(struct file *, loff_t, int);
8     int (*proc_release)(struct inode *, struct file *);
9     __poll_t (*proc_poll)(struct file *, struct poll_table_struct *);
10    long (*proc_ioctl)(struct file *, unsigned int, unsigned long);
11    #ifdef CONFIG_COMPAT
12        long (*proc_compat_ioctl)(struct file *, unsigned int, unsigned
        long);
13    #endif
14    int (*proc_mmap)(struct file *, struct vm_area_struct *);
15    unsigned long (*proc_get_unmapped_area)(struct file *, unsigned long,
        unsigned long, unsigned long, unsigned long);
16 } __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：

```

1 struct seq_file {
2     char *buf; /* seq_file接口使用的缓存页指针 */
3     size_t size; /* seq_file接口使用的缓存页大小 */
4     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 */
5     size_t count; /* buf中可以拷贝到用户态的字符数目 */
6     size_t pad_until;
7     loff_t index; /* start, next的处理的下标pos数值 */
8     loff_t read_pos; /* 当前已拷贝到用户态的数据量大小 */
9     struct mutex lock; /* 针对此seq_file操作的互斥锁 */
10    const struct seq_operations *op; /* 操作实际底层数据的函数 */
11    int poll_event;
12    const struct file *file;
13    void *private;
14 };

```

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。

```

1 struct seq_operations {
2     void * (*start) (struct seq_file *m, loff_t *pos);
3     void (*stop) (struct seq_file *m, void *v);
4     void * (*next) (struct seq_file *m, void *v, loff_t *pos);
5     int (*show) (struct seq_file *m, void *v);
6 };

```

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：

```
1 void seq_printf(struct seq_file *m, const char *fmt, ...);
2 void seq_putc(struct seq_file *m, char c);
3 void seq_puts(struct seq_file *m, const char *s);
4 void seq_put_decimal_u11_width(struct seq_file *m, const char *delimiter,
5                               unsigned long long num, unsigned int width);
6 void seq_put_decimal_u11(struct seq_file *m, const char *delimiter,
7                           unsigned long long num);
8 void seq_put_decimal_l1(struct seq_file *m, const char *delimiter, long long
9                          num);
10 void seq_put_hex_l1(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：

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

以single_open为例：

```
1 int single_open(struct file *file, int (*show)(struct seq_file *, void *),
2               void *data)
3 {
4     struct seq_operations *op = kmalloc(sizeof(*op), GFP_KERNEL_ACCOUNT);
5     int res = -ENOMEM;
6
7     if (op) {
8         op->start = single_start;
9         op->next = single_next;
10        op->stop = single_stop;
11        op->show = show;
12        res = seq_open(file, op);
13        if (!res)
14            ((struct seq_file *)file->private_data)->private = data;
15        else
16            kfree(op);
17    }
18    return res;
19 }
20 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：

```
1 #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：

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

接下来就是具体读写函数的实现：

```
1 static ssize_t param_read(struct file *file, char __user *buf, size_t lbuf,
2 loff_t *ppos)
3 {
4     int nbytes = sprintf(kstring, "%d\n", enable);
5     return simple_read_from_buffer(buf, lbuf, ppos, kstring, nbytes);
6 }
7 static ssize_t param_write(struct file *file, const char __user *buf, size_t
8 lbuf, loff_t *ppos)
9 {
10     ssize_t rc;
11     rc = simple_write_to_buffer(kstring, lbuf, ppos, buf, lbuf);
12     sscanf(kstring, "%d", &enable);
13     optParam(enable);
14     return rc;
15 }
16 static struct proc_ops my_proc_fops = {
17     .proc_write = param_write,
18     .proc_read = param_read,
19 };
```

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。

```

1  struct info_entry {
2      struct list_head node;
3      u32    pid;
4      u32    cpu;
5      u64    time;
6      char    comm[TASK_COMM_LEN];
7      unsigned long    stack_entries[NUM_STACK_ENTRIES];
8      u32    num_stack_entries;
9      int    irq;
10     unsigned long lock_address;
11 };
12
13 struct info_head {
14     atomic_t num;
15     struct list_head list;
16     struct list_head *new;
17 };
18
19 static struct info_head printinfo_head = {
20     .num = {
21         0,
22     },
23     .list = {
24         .prev = &(printinfo_head.list),
25         .next = &(printinfo_head.list),
26     },
27     .new = &(printinfo_head.list),
28 };

```

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

```

1  if (atomic_read(&(printinfo_head.num)) >= cache_size) {
2      lnode = printinfo_head.new->next;

```

```

3         if (!node == &(printinfo_head.list)) {
4             lnode = lnode->next;
5         }
6
7         printinfo_head.new = lnode;
8
9         pos = container_of(lnode, struct info_entry, node);
10
11        pos->cpu = cpu;
12        pos->pid = current->pid;
13        pos->time = delta;
14        memcpy(pos->comm, current->comm, TASK_COMM_LEN);
15        pos->irq = irq;
16        pos->lock_address = lock_address;
17
18        pos->num_stack_entries = stack_trace_save(
19            (unsigned long *) (pos->stack_entries),
20            NUM_STACK_ENTRIES,
21            1);
22
23
24
25    } else {
26        printinfo_entry = kmalloc(sizeof(*printinfo_entry), GFP_KERNEL);
27
28        printinfo_entry->cpu = cpu;
29        printinfo_entry->pid = current->pid;
30        printinfo_entry->time = delta;
31        memcpy(printinfo_entry->comm, current->comm, TASK_COMM_LEN);
32        printinfo_entry->irq = irq;
33        printinfo_entry->lock_address = lock_address;
34
35
36        printinfo_entry->num_stack_entries = stack_trace_save(
37            (unsigned long *) (printinfo_entry->stack_entries),
38            NUM_STACK_ENTRIES,
39            1);
40
41        atomic_inc(&(printinfo_head.num));
42        list_add_tail(&(printinfo_entry->node), &(printinfo_head.list));
43        printinfo_head.new = &(printinfo_entry->node);
44    }

```

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。

```

1  #define kfifo_alloc(fifo, size, gfp_mask)
2  __kfifo_int_must_check_helper(
3  ({
4      typeof((fifo) + 1) __tmp = (fifo);
5      struct __kfifo *__kfifo = &__tmp->kfifo;
6      __is_kfifo_ptr(__tmp) ?
7      __kfifo_alloc(__kfifo, size, sizeof(*__tmp->type), gfp_mask) :
8      -EINVAL;
9  })
10 )

```

kfifo_put()向kfifo缓冲区中放入一个新的数据:

```

1  #define kfifo_put(fifo, val)
2  ({
3      typeof((fifo) + 1) __tmp = (fifo);
4      typeof(*__tmp->const_type) __val = (val);
5      unsigned int __ret;
6      size_t __recsize = sizeof(*__tmp->rectype);
7      struct __kfifo *__kfifo = &__tmp->kfifo;
8      if (__recsize)
9          __ret = __kfifo_in_r(__kfifo, &__val, sizeof(__val),
10                               __recsize);
11      else {
12          __ret = !kfifo_is_full(__tmp);
13          if (__ret) {
14              (__is_kfifo_ptr(__tmp) ?
15               ((typeof(__tmp->type))__kfifo->data) :
16               (__tmp->buf)
17              )[__kfifo->in & __tmp->kfifo.mask] =
18                  *(typeof(__tmp->type))&__val;
19              smp_wmb();
20              __kfifo->in++;
21          }
22      }
23      __ret;
24  })

```

kfifo_get()从kfifo缓冲区中取出一个最“旧”的数据:

```

1  #define kfifo_get(fifo, val)
2  __kfifo_uint_must_check_helper(
3  ({
4      typeof((fifo) + 1) __tmp = (fifo);
5      typeof(__tmp->ptr) __val = (val);
6      unsigned int __ret;
7      const size_t __recsize = sizeof(*__tmp->rectype);
8      struct __kfifo *__kfifo = &__tmp->kfifo;
9      if (__recsize)
10         __ret = __kfifo_out_r(__kfifo, __val, sizeof(*__val),
11                               __recsize);
12      else {
13          __ret = !kfifo_is_empty(__tmp);
14          if (__ret) {
15              *(typeof(__tmp->type))__val =
16                  (__is_kfifo_ptr(__tmp) ?

```



```

17         ((typeof(__tmp->type))__kfifo->data) :
18         (__tmp->buf)
19     )[__kfifo->out & __tmp->kfifo.mask];
20     smp_wmb();
21     __kfifo->out++;
22 }
23 }
24 __ret;
25 })
26 )

```

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

```

1 struct kfifo g_fifoqueue;
2 /*
3  该函数创建并初始化一个size大小的kfifo。
4  内核使用gfp_mask标识符分配队列的缓冲区内存。
5  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：
6  */
7 int kfifo_alloc(struct kfifo *fifo, unsigned int size, gfp_t gfp_mask);
8 void kfifo_init(struct kfifo *fifo, void *buffer, unsigned int size);

```

静态创建

```

1 DECLARE_KFIFO(name, size) ;
2 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：

```

1 kfifo_in(&output_fifo, info, ret);

```

```

1 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。

```

1 pos->num_stack_entries = stack_trace_save(
2     (unsigned long *) (pos->stack_entries),
3     NUM_STACK_ENTRIES,
4     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。

```

1 for (i = 0; i < pos->num_stack_entries; i++) {
2     seq_printf(p, "%pS\n", (void *)pos->stack_entries[i]);
3 }

```

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。

```

1 pos->cpu = cpu;
2 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。

```

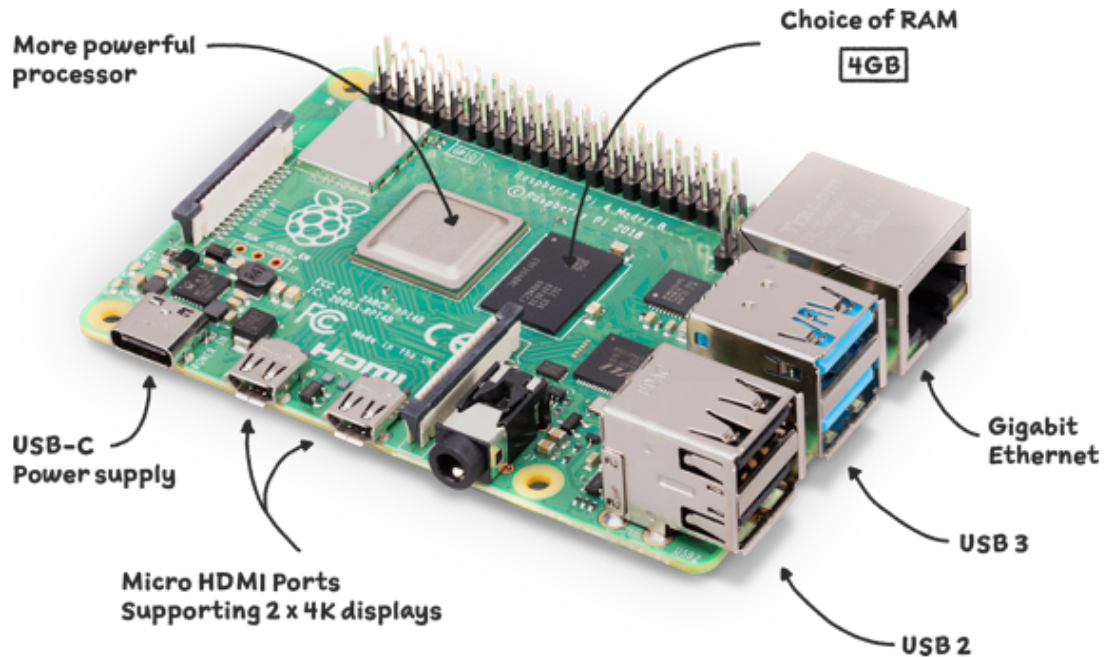
1     files = current->files;
2
3     fdt = files->fdt;
4     max_fds = fdt->max_fds;
5
6     printk("-----\n");
7
8     for (i = 0; i < max_fds; i++) {
9         if (fd_is_open(i, fdt)) {
10
11             f = fdt->fd[i];
12
13             f_name = f->f_path.dentry->d_iname;
14             f_inode = f->f_inode;
15             if (f_inode->i_mode & S_IFSOCK) {
16                 is_sock = 1;
17             } else {
18                 is_sock = 0;
19             }
20             printk("Pid: %d filename: %s Sock: %s\n", current->pid, f_name,
is_sock ? "YES" : "NO");
21
22         }
23     }

```

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:

```
1 #kernel version
2 Linux shi-Rspi 5.15.0-1008-raspi #8-Ubuntu SMP PREEMPT
3 aarch64 aarch64 aarch64 GNU/Linux
4
5 #CPU
6 Architecture:          aarch64
7   CPU op-mode(s):      32-bit, 64-bit
8   Byte Order:           Little Endian
9   CPU(s):              4
10  On-line CPU(s) list:  0-3
11  Vendor ID:            ARM
12  Model name:           Cortex-A72
13  Model:               3
14  Thread(s) per core:   1
15  Core(s) per cluster:  4
16  Socket(s):            -
17  Cluster(s):           1
18  Stepping:             r0p3
19  CPU max MHz:          1500.0000
20  CPU min MHz:          600.0000
21  BogoMIPS:             108.00
22  Flags:                 fp asimd evtstrm crc32 cpuid
23  Vulnerabilities:
24    Itlb multihit:       Not affected
25    L1tf:                Not affected
26    Mds:                 Not affected
27    Meltdown:            Not affected
28    Spec store bypass:    vulnerable
29    Spectre v1:          Mitigation; __user pointer sanitization
30    Spectre v2:          vulnerable
```

31

Srbd:

Not affected

32

Tsx async abort:

Not affected

33

34

#disk

35

Filesystem

Size

Used

Avail

Use%

Mounted on

36

tmpfs

379M

3.6M

376M

1%

/run

37

/dev/mmcblk0p2

15G

11G

3.3G

77%

/

38

tmpfs

1.9G

0

1.9G

0%

/dev/shm

39

tmpfs

5.0M

4.0K

5.0M

1%

/run/lock

40

/dev/mmcblk0p1

253M

120M

133M

48%

/boot/firmware

41

tmpfs

379M

76K

379M

1%

/run/user/127

42

tmpfs

379M

68K

379M

1%

/run/user/1000

43

44

#mem

45

total

used

free

shared

buff/cache

available

46

Mem:

3.7Gi

682Mi

1.3Gi

5.0Mi

1.7Gi

2.9Gi

47

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

```
root@shi-Rspi:/proc# ls
1 1094 1183 147 1537 2017 2354 317 449 47430 590 71721 72505 891 cgroups irq_time_info partitions vmallocinfo
10 11 1189 1472 1538 2030 2379 32 45 48 618 71724 72509 892 cmdline kallsyms pressure
100 110 119 1479 1542 2044 2389 33 452 49 619 71726 72528 9 consoles kcore schedstat
101 1103 12 1482 1589 205 24 36 453 564 620 71736 72559 917 cpuinfo keys scsi
102 1105 120 1486 1617 2063 2402 362 454 565 69697 71737 72677 918 crypto key-users self
1025 111 122 1498 17 2072 2417 37 456 566 698 71738 727 927 devices kmsg slabinfo
1028 1113 1242 15 178 2084 2458 38 4592 567 699 71739 730 932 device-tree kpagecgroup softirqs
1029 1118 1244 1502 171 21 2482 4 4593 568 701 71740 731 95 diskstats kpagecount stat
103 112 1251 1504 18 2116 2493 42 46 580 706 71741 732 952 driver kpageflags swaps
1050 113 1293 1507 19 2117 25 422 460 581 712 71745 734 96 dynamic_debug loadavg sys
106 1134 13 1509 1989 2121 26 423 461 582 713 71746 735 97 execdomains locks sysrq-trigger
107 1135 1303 1513 1992 2123 262 437 462 583 71481 71750 740 98 fb mdstat sysvipc
1070 1137 131 1518 1993 2146 263 438 463 584 715 718 8 99 filesystems meminfo thread-self
1079 114 134 1521 2 2181 27 439 47 585 71641 71842 819 994 fs misc timer_list
108 116 1344 1523 20 2191 2904 44 4730 586 71680 71877 822 asound modules tty
1087 117 14 1529 2000 2252 3 440 47316 587 71694 71882 852 bootconfig iomem mounts uptime
1091 118 145 1532 2001 2326 30 441 47321 588 71702 71994 853 buddyinfo ioports net version
1092 1180 146 1533 2002 2349 31 445 474 589 71718 721 882 bus irq pagetypeinfo version_signature
root@shi-Rspi:/proc#
```

各子节点

```
root@shi-Rspi:/proc/irq_time_info# pwd
/proc/irq_time_info
root@shi-Rspi:/proc/irq_time_info# ls
enable hd_irq_id output threshold_ns
root@shi-Rspi:/proc/irq_time_info#
```

节点交互

这里由于还没有打开模块开关，所以 output 暂时没有数据

```
root@shi-Rspi:/proc/irq_time_info# ls
enable hd_irq_id output threshold_ns
root@shi-Rspi:/proc/irq_time_info# cat *
0
0
1000
root@shi-Rspi:/proc/irq_time_info#
```


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@shi-Rspi:/proc/irq_time_info# cat enable
0
root@shi-Rspi:/proc/irq_time_info# echo 1 > enable
root@shi-Rspi:/proc/irq_time_info#
root@shi-Rspi:/proc/irq_time_info# cat enable
1
root@shi-Rspi:/proc/irq_time_info#
```

4.2.2 系统全部关中断时长(ns级)

```
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24270352
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24287500
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24293611
[cpu:1] [pid: 727] spin_locksave_duration_time(ns): 24225594
[cpu:3] [pid: 317] spin_locksave_duration_time(ns): 24244685
[cpu:1] [pid: 727] spin_locksave_duration_time(ns): 24232631
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 24305407
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 24313759
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24368054
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 24338962
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 24346684
[cpu:3] [pid: 317] spin_locksave_duration_time(ns): 24343369
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24399887
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24406776
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24416887
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24435645
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24441794
[cpu:1] [pid: 727] spin_locksave_duration_time(ns): 24374239
[cpu:1] [pid: 727] spin_locksave_duration_time(ns): 24380924
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24467108
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 24456460
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24490404
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 24465497
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24504478
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24515070
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24531903
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24537958
[cpu:1] [pid: 727] spin_locksave_duration_time(ns): 24469941
[cpu:1] [pid: 727] spin_locksave_duration_time(ns): 24476978
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 24540810
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 24548144
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 24572069
[cpu:0] [pid: 718] spin_locksave_duration_time(ns): 24607902
```

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#
```

可以看到修改阈值以后，捕获到的所有有关中断时长都在所设阈值之上：

```
[cpu:0] [pid: 727] spin_locksave_duration_time(ns): 27468333
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 27448945
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 27493555
[cpu:0] [pid: 727] spin_locksave_duration_time(ns): 27531110
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 27500907
[cpu:0] [pid: 727] spin_locksave_duration_time(ns): 27539554
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 27528073
[cpu:1] [pid: 718] spin_locksave_duration_time(ns): 27507091
[cpu:0] [pid: 727] spin_locksave_duration_time(ns): 27565664
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 27535369
[cpu:0] [pid: 727] spin_locksave_duration_time(ns): 27572387
[cpu:1] [pid: 718] spin_locksave_duration_time(ns): 27517276
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 27583424
[cpu:1] [pid: 718] spin_locksave_duration_time(ns): 27560905
[cpu:0] [pid: 727] spin_locksave_duration_time(ns): 27620997
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 27592368
[cpu:1] [pid: 718] spin_locksave_duration_time(ns): 27570961
[cpu:0] [pid: 727] spin_locksave_duration_time(ns): 27629089
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 27633182
[cpu:1] [pid: 718] spin_locksave_duration_time(ns): 27628830
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 27655589
[cpu:0] [pid: 727] spin_locksave_duration_time(ns): 27693162
[cpu:1] [pid: 718] spin_locksave_duration_time(ns): 27637978
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 27662626
[cpu:0] [pid: 727] spin_locksave_duration_time(ns): 27702218
[cpu:1] [pid: 718] spin_locksave_duration_time(ns): 27681829
[cpu:1] [pid: 718] spin_locksave_duration_time(ns): 27689274
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 27714588
[cpu:0] [pid: 727] spin_locksave_duration_time(ns): 27752069
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 27722107
[cpu:0] [pid: 727] spin_locksave_duration_time(ns): 27761383
[cpu:2] [pid: 718] spin_locksave_duration_time(ns): 27751125
[cpu:0] [pid: 727] spin_locksave_duration_time(ns): 27788735
```

5. 总结

5.1 项目开发进展

日期	进展
4.15-4.22	查看 proc 的实现，学习proc的接口API及其使用
4.22-4.29	查看kernel documents 及博客 等关于 kprobes机制的分析，尝试编写 kprobes demo
4.29-5.6	分析中断流程的过程，寻找挂载点
5.6-5.13	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
5.13-5.20	实现stack_trace 及持有锁 等功能
5.20-5.27	实现文件&socket等信息的提取
5.27-6.3	整理前面学习文章，撰写初赛文档

5.2 遇到的问题和解决办法

1. 中断挂载点问题

- 1 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：
- 2 - 在内核模块中实现tracepoint的使用
- 3 - 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
- 4 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

- 1 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：

```
1 CONFIG_KPROBES=y
2 CONFIG_MODULES=y
3 CONFIG_MODULE_UNLOAD=y
4 CONFIG_KALLSYMS=y
5 CONFIG_KALLSYMS_AL=y
6 CONFIG_DEBUG_INFO=y
```

6.2 克隆并运行

克隆仓库：

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

编译及插入模块

```
1 make
2 make run
```

查看实验结果

```
1 cd /proc/irq_info
2 # enable  模块开关 0-关闭 1-启用
3 # threshold  阈值大小
4 # hd_irq_id  硬件中断号
5 # output  信息输出
6
7 eg:
8 echo 1 > enable      #开启模块
9 echo 2000 > threshold #更新阈值大小
10 cat output          #查看信息输出
```