# 项目背景

在工业控制、机器人控制领域等越来越多使用 Linux 操作系统，但作为一个通用操作系统，Linux 系统在实时性方面天然不具备优势。为了改善 Linux 实时性，需要开发一个探针型的工具用于分析中断较高的原因，以便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。

# 目标描述

## 第一题：内核模块基础框架实现

1.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

2.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

3.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

4.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

以题目一为基础，工具需要追加下列功能：

1.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

2.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

3.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

4.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。

## （可选）第三题：工具稳定性验证和实际效果测试

以题目二为基础，工具需要追加下列功能：

1.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

2.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

# 完成度

目前，我们的赛题完成度如下：

|  |  |  |
| --- | --- | --- |
| 题目编号 | 基本完成情况 | 额外说明 |
| 1 | 基本完成（≈90%） | ① 基本的数据读写、存储等功能已实现；  ② 目前未发现 BUG； |
| 2 | 基本完成 | 目前尚未实现对持有锁信息的抓取，其余功能均已实现 |
| 3（选做） | 部分完成 | ①能够在内核态长时间运行，无明显的内存泄漏与系统卡顿；  ②暂未在高负载场景进行工具的功能性和稳定性测试 |

# 比赛题目分析和相关资料调研

## 选题原因：

负责指导的钟老师可以联系到麒麟的工程师帮忙指导，因此我们的选题从麒麟命制的题目中选择。

对选题进行分析：

- Proj154－Linux 内核实时性瓶颈分析工具（有价值，有一定工作量）

- Proj155－基于麒麟操作系统的LoongArch架构开源浏览器移植及定制优化（好像已经有能用的 Chromium 了，感觉可能比较注重工程实践，理论方面可能较少）

- Proj156－一种基于Linux系统运行android11的解决方案（Android 编译好像比较复杂，而且应该工程量大，可能超出了能力范围）

- Proj157－麒麟系统下基于wayland的截图管控（能做，但和操作系统核心功能之类的可能关系不太大）

- Proj159－基于国产Phytium 芯片的FreeRTOS Coredump机制的实现（可能能做，和芯片有关）

- Proj160－基于Linux系统的崩溃收集及崩溃日志生成组件（不知道项目的亮点在什么地方，好像 Linux 已有这方面的组件）

- Proj161－优麒麟操作系统安全分析工具（对 OS 安全了解少）

- Proj162－linux基于文件扩展属性的执行控制（有价值，有一定工作量）

- Proj163－制作轻量级测试操作系统（应该能做，但如果只是跑测试软件的话可能通用性差一些）

- Proj164－基于RISC-V架构平台的Android系统移植（应该不行，既不了解 RISC-V 也不了解 Android）

- Proj165－实现针对android系统动态攻击的检测功能（应该不行，不了解安全这方面）

- Proj166－基于RISC-V的FreeRTOS国密算法功能实现（似乎写出来难度不太高，但对安全方面缺乏了解）

可能比较适合的是 154 和 162。最后，因为我们当时认为实时性方面的资料更多一些，决定选择 Proj154。

## 具体题目内容分析：

根据我们的理解，此题的目的在于开发一个内核模块，此模块能够检测时间超过阈值的关所有中断或屏蔽指定中断号的中断这一动作的发生，并记录下此次关中断的相关信息，以备对高延迟事件进行分析，进而改进实时性问题。同时，需要采用 procfs 等方式完成与用户的交互。为了增强性能，内部链表需要使用cache机制。同时，尽量对 linux 内核不作改动，以降低使用难度、提高通用性。

## 相关资料调研：

### 内核模块基础框架：

我们可以在insmod这个模块时同时通过输入来设置参数，但是这样不太方便。因此，我们考虑在模块启动时读取一个配置文件来把设置好的参数传入模块中。这个配置文件很简单，仅3行，分别对应3个参数。写一个简单的shell脚本，以在装载模块时读取配置文件把参数传入模块中。用户想要修改参数时，只需修改配置文件，下一次装载模块时即生效。

但是用户可能想在模块运行过程中就直接修改参数。而且修改配置文件、重装模块的操作也略显繁琐。用户希望可以通过简单的shell命令来修改这3个参数，以控制我们模块的行为。模块抓取到有关信息后，用户也希望可以通过shell命令来读取或清空这些信息。然而，内核空间与用户空间之间互相无法随意访问。需要一些特定的方法：系统调用、ioctl、procfs、sysfs、netlink。

系统调用就是我们常用read()，wirte()等，但如果我们要自己添加一个自定义的系统调用，就要重新编译内核，比较麻烦。ioctl是/dev/xxx文件，procfs是/proc/xxx文件，sysfs是/sys/xxx文件，这3种方式的使用方法很相似。netlink适合大数据量，像socket。本题目建议我们使用procfs。

“/proc”文件系统是一个虚拟文件系统。在/proc中，我们可以将对虚拟文件的读写作为与内核通信的手段。procfs提供了接口，我们可以自己在/proc中创建虚拟文件，使得我们自己的内核模块和用户态程序之间可以通过这些文件进行数据传递。

### 探针工具方面：

本题目的关键在于，内核模块采用何种方式检测开关中断事件。这部分相对而言在互联网上资料较少，我们花了更多时间进行调研与验证。

关某一 CPU 上所有中断可以使用宏 local\_irq\_disable，开中断可以使用宏 local\_irq\_enable。

禁止指定中断号的中断，直到完成后才返回，可以调用函数 disable\_irq(unsigned int irq)；禁止指定中断号的中断并立即返回可以调用disable\_irq\_nosync(unsigned int irq)；使能指定中断号的中断可以调用enable\_irq(unsigned int irq)。

因此，最容易想到的方法是直接在 linux 内核对应的位置插入代码，这样的好处是方案简单，不需要依赖某些追踪技术，缺点也很明显，需要对内核源代码作直接的改动，随着内核的迭代，代码可能需要随之调整，维护成本高。

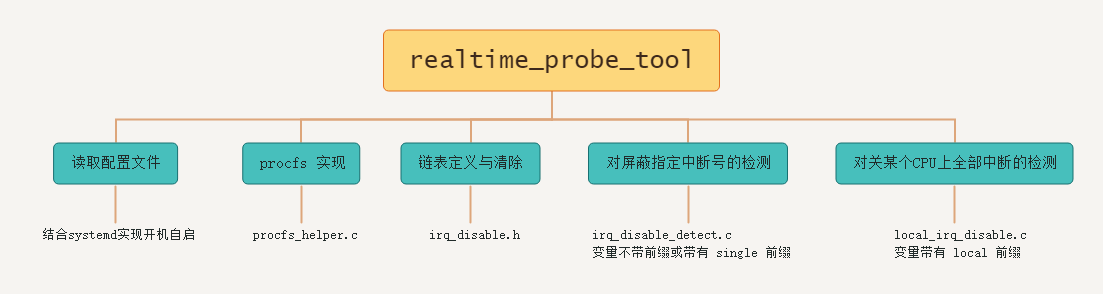
幸运的是，linux 提供了用于在内核中添加追踪点的基础设施。Kprobe 可以用于动态地在内核中添加探测点，可以在进入断点前、结束后执行回调函数，其大致原理为：在 x86\_64 架构中，kprobe 把指令拷贝后，将原始指令入口替换为 int3（内核开启 CONFIG\_OPTPROBES 选项后会用更优的跳转指令来代替），CPU 执行到 int3 时会陷入 trap，在 trap 的处理函数中会设置 kprobe 的调用状态并调用注册的 pre\_handler 回调函数，之后通过设置单步调试 flag，kprobe 单步执行被拷贝的指令，执行完成后执行 post\_handler 回调函数；最后回到正常的执行流程。经测试，可以命中屏蔽或开启指定中断号的一系列函数，但不能命中关某一 CPU 上所有中断的那些宏。事实上，当程序调用 local\_irq\_disable 关某一 CPU 上所有中断时，经反汇编可知，其调用相关的函数具有 notrace 标记，正常情况下 kprobe 不会追踪。事实上，kprobe 不追踪此函数是有原因的：kprobe 本身也会关中断，追踪 local\_irq\_disable 等可能会陷入无限循环的递归调用之中。

经互联网检索，还有一种设计思路，其中心思想在于采样。根据奈奎斯特采样定理，我们只需以阈值的1/2为周期进行采样，具体采样可以利用高精度计时器 hrtimer，每次在到达设定的时间间隔后均会产生一个中断，我们在中断中获取当前的时间，并与原来设定的时间比较，如果检测到大于阈值的延时，则说明此时发生了关中断。经我的实际验证，此方法在技术上可行，且对使用的内核没有特别要求，但也有其弊端：对某些情况难以进行区分，可能会漏掉一部分阈值以上的关中断信息；且基于采样的方法显然无法得出具体的关中断时长，只能给出关中断时长的区间范围，与题目要求的“精度需要达到纳秒级别”不符。

与赛题导师交流后，导师建议我们采用 tracepoint 这一预先安排在内核中各种关键点上的静态探测点机制，这样命中探测点后会调用我们注册的回调函数。经查，虽然此方案需要在内核中打开 CONFIG\_TRACE\_IRQFLAGS 编译选项，可能需要重新编译内核，但其能满足在精度方面的要求，且不需要对内核源代码作改动，相对容易维护，可以说取得了一个平衡。由于 tracepoint 似乎不常单独使用，网络上资料较少，我们在参考了 GitHub 的开源代码后才使其得以工作，最终，我们采用 tracepoint 对屏蔽指定中断号进行监测。

Linux 使用 task\_struct 作为任务控制块，在关中断或开中断时，我们可以通过读取其中的数据成员来读出此时的进程相关的信息，包括进程名、PID、调用栈、文件描述符表等。

# 系统框架设计

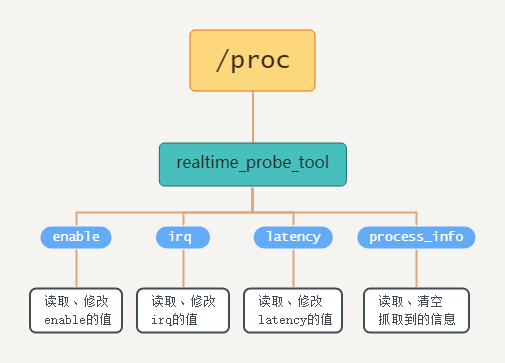


同时，我们也整理了目前测试用的文件。



我们在模块中设置了3个参数：enable, irq, latency。它们分别代表模块开关、中断号、关闭时长阈值。

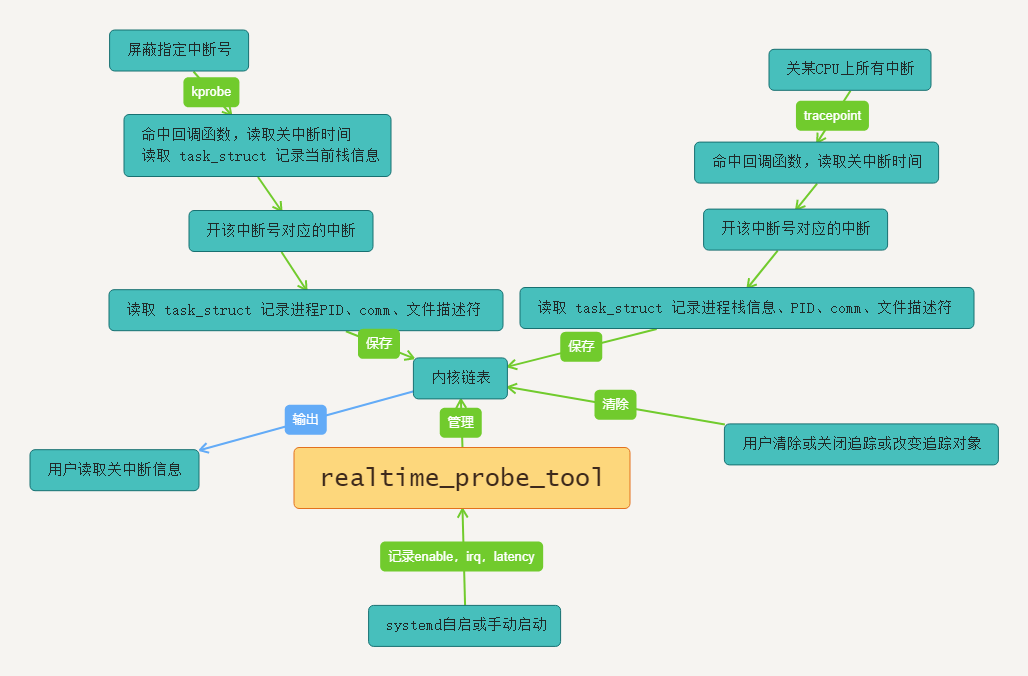
我们在/proc中创建的文件结构如下图。



具体来说，我们在/proc下新建目录realtime\_probe\_tool/ ，在/proc/ realtime\_probe\_tool/下创建4个文件enable, irq, latency, process\_info, 分别实现不同的功能。enable, irq, latency这3个文件，分别提供给用户读取、修改enable, irq, latency这3个参数的值。process\_info文件用于让用户读取、清空我们内核模块抓取到的信息。用户很容易操作这些文件，只需像操作普通文件那样，在shell中用echo、cat命令，或者在C程序中用open()、read()、write()函数即可读、写这几个虚拟文件。

数据存储方面，题目要求使用内核链表，并采用cache机制。内核链表是只有指针域没有数据域的双向循环链表，内核提供了许多方便的API给我们操作内核链表。cache机制就是指slub，slub是slab的升级版。内核同样提供了几个接口让我们利用slub来分配内存。

对于开机自启动功能，我们利用systemd服务来实现。systemd是linux中一种新的初始化系统。把我们的模块写成systemd服务，我们用systemctl命令就可以很方便地管理我们的模块、实现模块开机自启动。



使用 kprobe 在对应的函数上添加动态探测点，执行至此时调用我们的回调函数，实现对屏蔽指定中断号的监测；使用 tracepoint 将我们的另一组回调函数注册到内核中已经预设好的静态监测点，全局开关中断时就会调用。

# 开发计划

均需完成：

搭建模块开发与内核源代码查看的环境

叶景熙：

编写 procfs 相关的框架部分与用户交互，准备内核链表与cache

实现开机自启动与配置文件的读取

赵子晗：

搭建内核编译与调试环境

实现对屏蔽指定中断号的检测

实现对关某个 CPU 上中断的检测

子模块整合

## 未来展望：

- 就模块对性能的影响进行更深入的测试

- 与现存的同类工具进行比较

- 对关中断的进程持有锁的情况等作分析，增强项目创新性

# 比赛过程中的重要进展

**2022-05-11**

自编译内核成功启动 busybox，可以单步调试

**2022-05-15**

procfs实现了内核态与用户态之间简单的通信（字符串读写）。

**2022-05-21**

编译的内核成功在 QEMU 上引导 CentOS 8.5 启动。

**2022-05-23**

通过编写systemd服务，实现了模块的开机自启动、读取配置文件。

**2022-05-26**

完成对屏蔽指定中断号的监测

**2022-05-28**

成功实现tracepoint的使用

**2022-05-29**

procfs基本实现了预定功能。用户透过procfs可查看、修改模块的一些参数，读取、清空抓取到的数据。

# 系统实现

整个环境基于 CentOS 8.5 x86\_64，Linux 内核版本为 4.18.0，注意您需要检查自己运行的内核是否具有 KProbe 支持，是否开启了CONFIG\_TRACE\_IRQFLAGS 编译选项。如果您在 VMware 里启动QEMU，且宿主机使用 Intel CPU，那么您可以使用我编译好的内核。否则的话，您可能需要自己调整编译选项后重新编译内核，否则本工具将无法运行。

## Procfs实现

内核模块初始化时，用内核提供的API：proc\_mkdir，在/proc目录下新建一个目录realtime\_probe\_tool。再用API：proc\_create，在/proc/realtime\_probe\_tool下创建4个文件enable, irq, latency, process\_info，专门用于实现（自定义地）用户态与内核态的数据通信。每个文件分别用于实现不同的功能。这几个文件不同于一般的文件，在用户态对这些文件进行写操作相当于把数据传到内核态；用户态对这些文件进行读操作相当于把数据从内核态取出来。读写这些虚拟文件时有默认的read, write函数进行操作，但为了实现一些我们想要的功能，我们修改file operations结构体里面的函数指针，指向自己实现的read, write等函数（相当于重载函数）。这个file operations结构体和里面的函数，是我们自定义procfs操作的灵魂。

我们在procfs中创建了4个文件，要实现的功能各不相同，那么他们的read, write函数也不相同。需要注意的是，read、write函数并不能完全自定义的，要受到一定的限制。它们的函数原型是一定的，是内核规定好的。我们可以自定义的内容，是函数名称、参数名称和函数体的内容。

在模块中定义的3个参数enable, irq, latency，对应地在/proc/realtime\_probe\_tool/下创建3个文件enable, irq, latency，用于给用户查询和修改这些参数。如，/proc/realtime\_probe\_tool/enable文件用于给用户查询参数enable的值（read函数实现），并允许用户修改enable的值（write函数实现）。read函数是通过copy\_from\_user()把用户空间字符串复制到内核空间，分析这个字符串，把它转换成int型的参数，遇到非法输入则报错。write函数相反，把int型的参数转成字符串，用simple\_read\_from\_buffer()把字符串从内核传到用户态。

process\_info文件用于读取我们抓取到的信息。在它对应的read函数中，遍历存储着信息的内核链表，把信息输出给用户。在其write函数中主要考虑用户清空信息的功能，分析用户输入的字符串若为’clear’或’0’则把链表中存的信息全删除，释放内存。

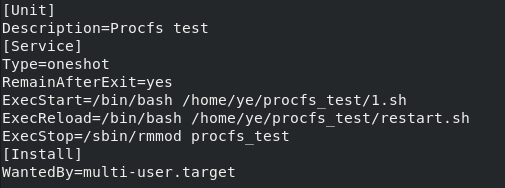
至此，已经实现了用户态和内核态的基本交互，procfs的基本框架已经搭建起来了，为后面与题目二衔接打下了基础。

## cache存储实现

我们可能需要频繁地新增存储数据，因此我们利用cache机制来加快数据的存储。这个cache机制就是slub。我们给数据分配内存空间的时候，若直接调用kmalloc/kfree函数，其实已经是利用了slub机制。但kmalloc函数自动为我们完成了分配内存的全部工作，那么可想而知效率就稍低一点。为了提高效率，针对我们特定的数据，我们采用自定义的slub来存数据。利用kmem\_cache\_create, kmem\_cache\_zalloc, kmem\_cache\_free, kmem\_cache\_destroy这4个内核提供的API来自定义slub。模块初始化时调用kmem\_cache\_create，按我们指定的大小和方式来创建一个新的cache。需要分配内存空间时调用kmem\_cache\_zalloc从刚创建的cache中分配一个内存对象，并把内存对象所代表的内存空间初始化为0。这个内存空间暂时不用时，调用kmem\_cache\_free将内存对象放回指定的cache中，这是与kmem\_cache\_zalloc相反的操作。alloc出去的内存全部free之后，若不再需要这个cache，调用kmem\_cache\_destroy来销毁指定的cache。模块退出时，先kmem\_cache\_free所有分配出去的对象，再kmem\_cache\_destroy销毁。

## 开机自启动实现

编写systemd服务以实现开机自启动。我们使用的CentOS 8.5是支持systemd服务的。在/usr/lib/systemd/system新建一个.service文件，按照一定的格式来编写，如下图（图中各文件放在用户目录下，命名较为随意）。



ExecStart中只需执行加载模块的命令（写成了shell脚本）。这是systemctl start时执行的命令。由于加载模块的命令“刚开始就结束”，Type应该选择oneshot，同时RemainAfter=yes。那么即使进程已结束，systemd也认为该服务正在运行，直到执行systemctl stop命令。ExecStop中只需执行卸载模块的命令（写成了shell脚本）。这样，一个systemd服务就写好了。之后，执行systemctl daemon-reload，这个.service文件就正式生效了。

现在要实现模块的开机自启动非常容易，执行一次systemctl enable命令即可，取消开机自启动则systemctl disable。开启、关闭、重新开启用systemctl start, systemctl stop, systemctl restart。

## 对屏蔽指定中断号的检测

检测指定中断号使用的是kprobe，具体实现见src/irq\_disable\_detect.c，关于这部分的范例和文档记录可以查看records/0522.md#kprobe。

每次有disable\_irq\_nosync、disable\_irq函数调用时会调用我们的pre\_handler\_disable\_irq回调函数，在此会先利用RDI寄存器判断此函数的参数是不是我们指定的中断号，如果是，此时会利用save\_stack\_trace\_tsk函数实现记录栈信息，并记录时间；当有enable\_irq函数调用时会调用回调函数pre\_handler\_enable\_irq，在此也会记录当前时间，如果超过了阈值，会用while (!atomic\_cmpxchg(&single\_list\_mark, 1, 0)) ;上锁，从 task\_struct 中获取文件描述符，加入链表中。

为了防止关闭中断过多导致消耗大量内存，我们在代码中对链表的长度给予限制，当链表长度达到限制时，不会再新增新的结点，而是从链表中取下最早的结点，将新的内容放入其中后再加入链表尾部，这样也可以极大地减少 kmalloc、kfree 调用的次数，节约资源消耗；由于不同进程持有的文件描述符个数不同，我们记录文件描述符时也采用的是链表存储，可能需要频繁申请和释放用于存储文件描述符内容的结点，我们通过利用slub，调用kmem\_cache相关内核函数实现了缓存机制。

## 对关某个 CPU 上所有中断的检测

检测关某个CPU上中断使用的是tracepoint，这部分参考了https://github.com/bristot/rtsl的部分代码，因为我遇到了一些有关符号未定义的困难，具体过程可以参考records/0525.md#TracePoint。

总体上代码逻辑和“对屏蔽指定中断号的检测”相似，但只在判断关中断时间大于阈值时才记录栈信息，这样避免了多余的kmalloc、kfree；而且 kfree 本身需要调用 local\_irq\_save、local\_irq\_restore，甚至有可能引起我们的递归，对此使用DEFINE\_PER\_CPU(bool, local\_tracing);忽略自身产生的中断。

## 多线程支持

对于屏蔽指定中断号的检测而言，由于采用的是kprobe，kprobe已经保证在不同处理器命中同一函数时采用串行化处理，在一个处理器正在执行回调函数时其余处理器不得进入，我们在这一点上不需要采用特别的方法保护全局变量。同时，kprobe在实现时也保证了执行的完整性，即如果某一线程正在执行回调函数时，另一线程调用了取消注册kprobe的函数，那么本次回调函数将会得到完整的执行。

对于关某个 CPU 上中断的检测而言，tracepoint 不具有这样的保护机制，需要我们自己实现。由于每个 CPU 的关中断是相对独立的，我们使用了内核PER CPU 变量，让每个 CPU 各自保存一份记录信息用的链表与缓存，这样多个 CPU 同时关中断时，不会发生数据争用。为了保证安全退出，不会因为链表突然清空而读到空指针，我们定义atomic\_t类型的PER\_CPU变量 local\_list\_mark，其值为1时表示空闲，为0时表示正在操作链表，不能进入临界区；在我们操作链表前会用原子操作atomic\_cmpxchg循环检查其值，一旦为1时立即赋0并进入临界区，退出时用原子操作atomic\_set将值恢复为1。这听起来和自旋锁spinlock十分相似，但事实上不可以直接使用自旋锁，因为经查源代码，自旋锁在spin\_unlock时会打开抢占，而内核中预留的静态探测点却是在关中断指令后、开中断指令前，即正常情况下回调函数执行时抢占应始终处于关闭状态。因此，我们用上述方法实现了一个简单的锁。

清空所有记录时面临的情况相似，我们仍然让清空的线程在获取锁之后才能进入临界区，当线程在清空时，其余线程只能等待，不会发生数据争用。

当用户读取process\_info时，执行该操作的线程需要访问其他 CPU 的 PER\_CPU 变量形式的链表，为此，仍然采用上述加锁的方法。由于抢占是关闭的，此种方法实现的锁应该是可行的。

|  |
| --- |
| 算法：获得锁 |
| while (!atomic\_cmpxchg(&my\_lock, 1, 0)) ;  // 进入临界区 |

|  |
| --- |
| 算法：释放锁 |
| atomic\_set(&my\_lock, 1);  // 离开临界区 |

# 提交仓库目录和文件描述

|-- records -- 文档记录

| |-- img -- 图片

| |-- 0424.md -- 过程文档，下同

| |-- 0501.md

| |-- 0508.md

| |-- 0511.md

| |-- 0515.md

| |-- 0522.md

| |-- 0525.md

| |-- 0604.md

| |-- 初赛文档.docx -- 初赛文档

| |-- mid-term-assessment.md -- 初赛阶段中间考核

|-- src -- 项目源代码

|-- test -- 测试所用文件

| |-- Autostart -- 自动启动与读取配置文件

| | |-- 1.sh -- systemd启动服务脚本

| | |-- procfs\_test.conf -- 加载的配置文件

| | |-- procfs\_test.service -- systemd.service

| | `-- restart.sh -- systemd 重启服务脚本

| |-- CentOS-kernel -- 对应 RPM 构建内核相关文件

| `-- LatencyTest -- 用于屏蔽指定中断号的模块

|-- LICENSE -- 开源协议

`-- README.md -- README

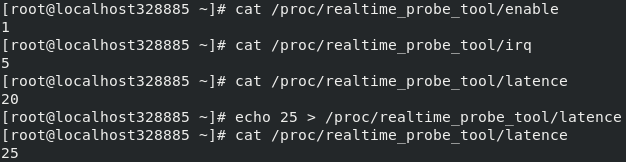
# 系统测试情况

## procfs相关测试

模块装载后/proc下即产生realtime\_probe\_tool目录，目录下5个文件。



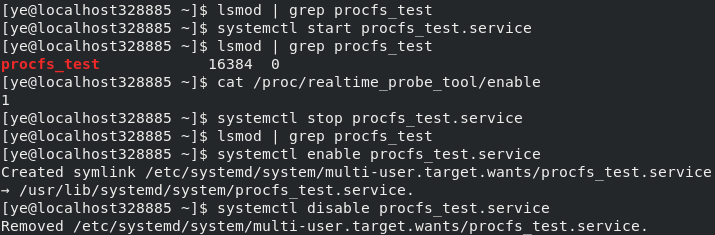
使用cat和echo命令读写enable, irq, latency文件。cat可以读取到相应参数的值，echo命令可以修改相应参数的值。



经测试，procfs的目标功能已经基本实现。

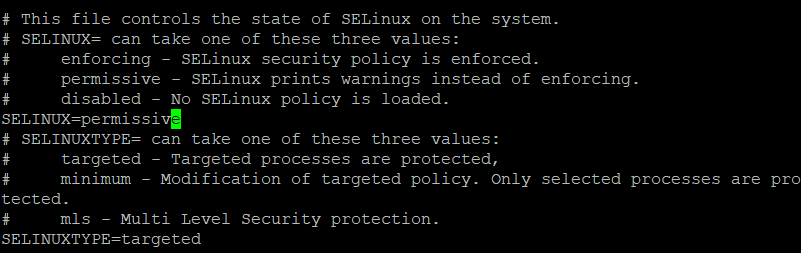
## systemd服务相关测试

利用systemd服务实现procfs\_test模块的开启、关闭（为了方便，模块名称和systemd服务名称均为procfs\_test）。systemctl enable实现模块的开机自启动，systemctl disable命令取消开机自启动。systemctl start开启服务（装载模块），systemctl stop关闭服务（卸载模块）。



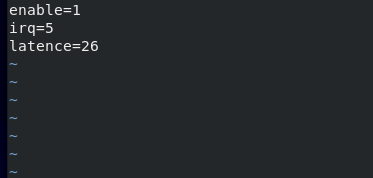
经测试systemd服务的这几个功能可以正常工作。

另外，使用systemd来控制我们的模块前，需要先关闭SELinux。修改/etc/selinux/config文件，将”SELINUX=enforcing”改为”SELINUX=permissive”，再重启即可。

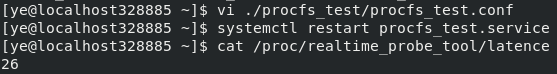


## 配置文件相关测试

vi编辑器打开配置文件，修改参数latency为26。



重启模块，会自动读取配置文件。读latency文件查询参数latency的值，发现已修改为26。说明配置成功。



经测试，配置文件相关功能正常。

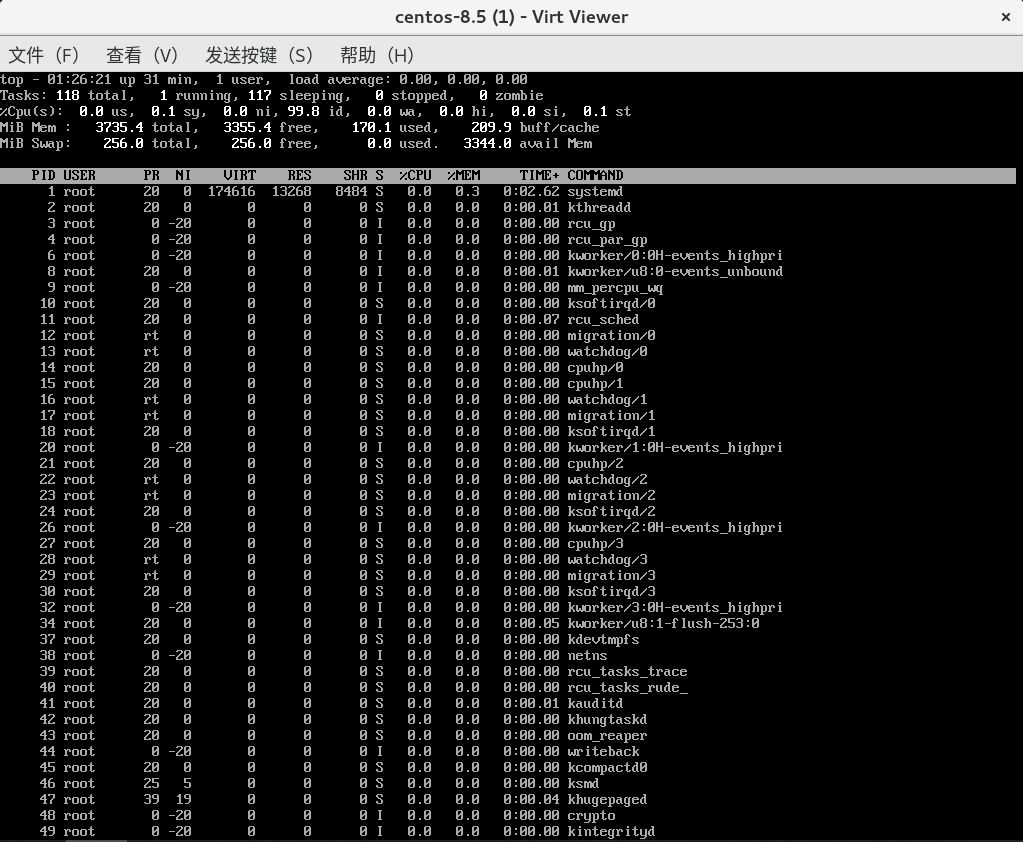
## 编译内核

这部分并不属于题目要求，但没有在编译时打开 CONFIG\_TRACE\_IRQFLAGS 的内核无法成功地注册 tracepoint，进而不能检测关闭某CPU上所有中断的发生。这一部分在records/0515.md#自编译内核引导-centos-启动 中有较为详细的介绍。

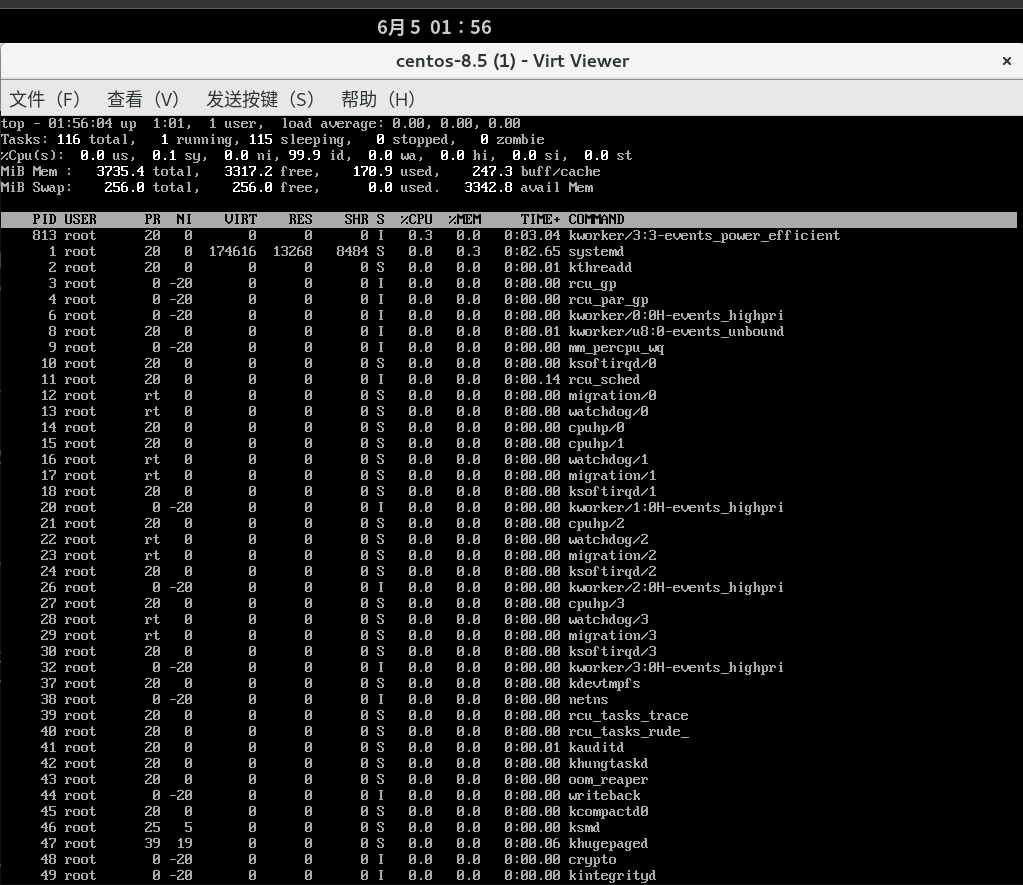
## 探针工具测试

演示视频：https://cloud.189.cn/web/share?code=vE7Rvy7NjqAn 访问码：rtj8

为了测试模块的长期运行不会导致内核崩溃，在设置成对关某个 CPU 上所有中断的检测后，运行30分钟，此时操作系统没有异样，top 命令输出如下：



继续运行30分钟，操作系统仍无异样，无卡顿或死锁、内核崩溃，top 命令输出如下：



未观测到明显的内存泄漏。

# 遇到的主要问题和解决方法

## procfs相关问题

①使用cat命令读取procfs中我们自己创建的文件时（如cat /proc/realtime\_probe\_tool/enable），和enable文件对应的read函数（由我们自定义）被重复调用了2次，因为在dmesg命令查看到read函数中的printk被执行了2次。而如果不使用shell的cat命令，改用C语言的read()，则一切正常，printk只执行了1次。网上关于这个问题的资料也不多。我想可能跟cat命令的原理有关？我做了许多尝试，譬如在文件的字符串末尾添加’\0’，但还是无济于事。最终，还是搜索到了这个问题的答案——确实跟cat命令的原理有关。它就是会连续2次调用read函数。不过按照我们代码的设计，用cat还是能正确地把内容输出。若真遇到cat不能用的情况，麻烦点用C语言的read()也完全没问题。这问题就告一段落了。

②另一个问题是关于procfs结构的问题。初学时，我看到一个范例在/proc下只新建了1个文件，我也跟着做。我们的procfs要实现多种信息的通信（读取参数值、读抓取到的信息、修改参数值、清空信息等等），我就一直思考用这1个文件如何实现众多的功能？像是走入了死胡同。后来在网上浏览他人的代码，才发觉完全可以在proc中多新建几个文件！这样就很容易实现多个功能分开处理了。

## 对关某个 CPU 上中断的检测相关问题

详见records/0525.md#关中断时动态分配内存 与 records/0604.md#kfree-引发的递归

## 多线程相关问题

在调试时，明显感到模块不稳定，有时问题并不立即显现，而是运行好几分钟后才出现。经过多次尝试，我初步学会了查看内核崩溃时产生的信息，结合内核源代码进行分析，实在不明白的地方就一边摸索着一边及时输出调试信息；对于自己不熟悉的，要一点一点地验证。

## 自编译内核引导 CentOS 相关问题

详见records/0515.md#探索曲折

解决：通过对比能在 QEMU 中引导 CentOS 的内核启动后的 lsmod 输出，查找遗漏的编译选项，最终成功启动。

# 分工和协作

## 叶景熙：

由于接触linux时间不长（仅1个月），主要负责比较简单的第一题——procfs实现内核态与用户态交互、模块开机自启动、cache机制存储数据。另外负责了初赛文档框架的起草。

## 赵子晗：

负责内核编译与单步调试环境的搭建、第二题的探针工具具体编程实现，同时撰写过程文档。

# 比赛收获

## 叶景熙：

对linux内核相关开发有了概念。报名比赛之时，我才刚接触linux一个月。当时安装linux也并不是因为兴趣，而只是为了完成操作系统课程的实验任务。后来了解到操作系统大赛，抱着学习的态度，在队友的建议下，我们报了功能赛。在麒麟软件导师的指导和自己的摸索下，我入门了linux内核开发。使用虚拟机、搭建samba在windows上访问linux、用source insight查看linux内核源代码、编写内核模块、了解内核编程与用户程序编程的差异……可以说，这次比赛为我打开了linux世界的大门。若没有参加操作系统大赛，我对linux的了解就仅止步于操作系统课程了。

## 赵子晗：

首先，我要感谢向我们推荐这个比赛的钟老师，没有他的大力支持，我们不可能有机会了解、参加；同时也要感谢麒麟软件的郭皓、高承博、张远航、马玉昆等导师，他们牺牲自己的时间，为我们传授正确的思路，没有他们的指点，我们将走向迷航；最后，感谢对我有充分信任和支持的队友。这次的比赛让我对Linux 开发有了初步的认识，在面对一些没有现成答案的问题时多方查找求证，能初步完成我感到非常幸运；也让我认识到和其他参赛队伍相比我们还有非常多的不足，时间分配不合理，对 Linux 内核架构缺乏深入了解，软件工程方面的意识缺乏，等等。大学的时间其实很快，不管怎样，这次初赛的经历仍然是宝贵而有意义的。