Hermes：一种基于 Linux 系统的内核模块

庞力铭 [1.1]　吴权达 [1.1]　王开苇 [1]　文宇华 [1]　李威汉 [1]　闻其帅 [0.9]

2023 年 5 月 22 日

1　摘要

在此项目中，我们开发了一个名为 Hermes 的 Linux 内核模块，旨在解决文件和进程跟踪问题。我们的目标是，让它能够像希腊神话中的 Hermes 一样，有效地在进程和文件之间进行信息的快速传递和管理。Hermes 具有读取指定文件中进程或文件信息、记录特定进程访问的文件和 IP 地址、展示进程或文件之间的关系以及记录和展示模块性能等功能。本报告将对 Hermes 模块进行深入的讨论，涵盖其架构、功能描述、数据结构、函数说明和性能测试等方面。

2　引言

操作系统，作为管理计算机硬件和软件资源的核心程序，为计算机系统的运行提供了许多基本的服务和功能，如进程管理、文件系统、内存管理和设备驱动等。在 Linux 操作系统中，其内核采用单一体系结构，并引入了一种全新的内核模块机制。该机制允许用户根据需要动态地装入或从内核移出模块，而无需重新编译内核。

在这样的背景下，我们采用加载内核模块到系统内核镜像的方式进行课程设计，以增强内核的功能，同时保证了内核的稳定性和可扩展性。我们开发的内核模块命名为 Hermes，期待通过这个模块，能在用户态和内核态进行有效的数据交互，以及收集、追踪和记录进程和文件的信息。

3　模块架构与功能描述

本模块主要包含以下几个部分：

1. 模块参数定义：定义了模块需要的各种参数，如进程 ID、程序名等。
2. 模块初始化和结束函数：负责模块的初始化和结束。
3. 读取文件输入到内核：通过调用函数 read_file 来实现。
4. 检查文件被哪些进程访问或并发访问：通过调用函数 find_processes_with_open_file 来实现。

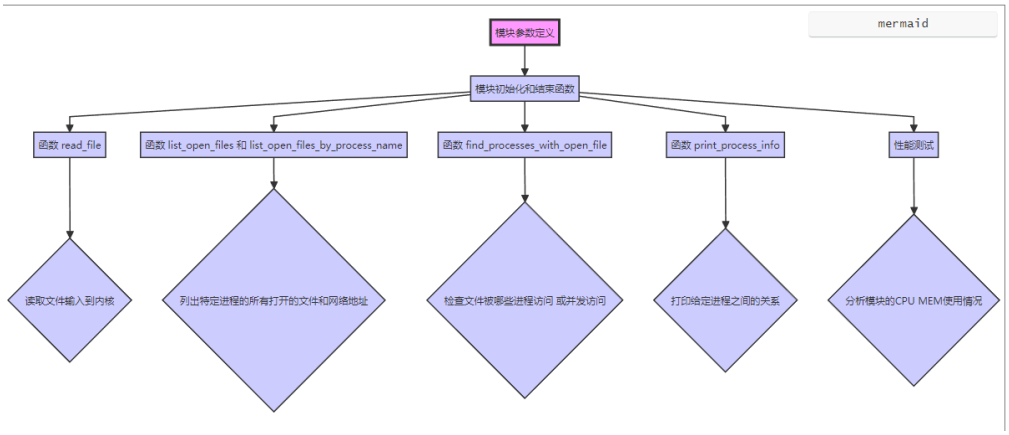


图 1: 模块架构与功能描述

5. 列出特定进程的所有打开的文件和网络地址：通过调用函数 list_open_files 和 list_open_files_by_process_name 来实现。
6. 打印给定进程之间的关系：通过调用函数 print_process_info 来实现。
7. 性能测试：分析模块的 CPU 和内存使用情况。

4 数据结构

4.1 task_struct

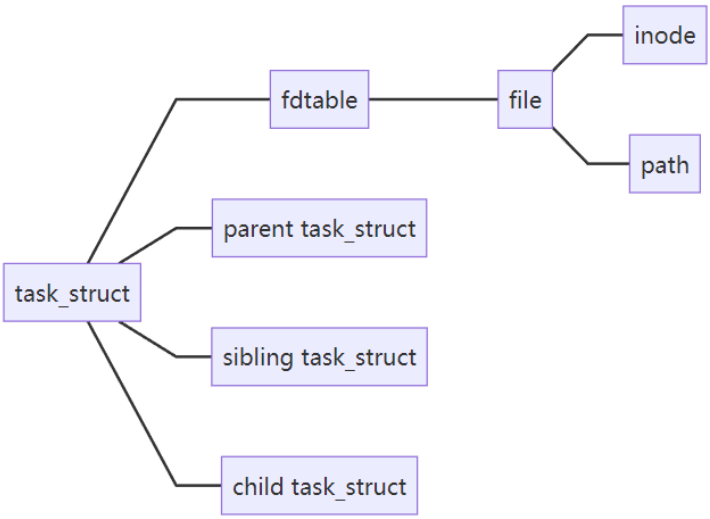


图 2: task_struct

1. task_struct: 这是一个内核数据结构，它表示了一个进程。这个结构体包含了许多有关进程的信息，比如进程的 PID，它的父进程和子进程等。
2. file: file 结构体表示了一个打开的文件。它包含了与该文件相关的信息，如 inode 和访问模式等。
3. fdtable: 这是一个表结构，表示了一个进程所有打开的文件描述符。它是 task_struct 的一个成员。
4. path：表示文件系统路径的数据结构。
5. inode: inode 结构体是文件系统用来表示文件的数据结构。它包含了文件的元数据，比如权限、所有者、大小等。

4.2 file_struct

```
struct file_struct{
    int pid_size;
    int prog_size;
    int file_size;
    char pid[MAX_BUFFER][MAX_SIZE];
    char prog[MAX_BUFFER][MAX_SIZE];
    char file[MAX_BUFFER][MAX_SIZE];
};
```

图 3: file_struct

1. pid_size: 这个字段表明 pid 数组中有效元素的数量。
2. prog_size: 这个字段表明 prog 数组中有效元素的数量。
3. file_size: 这个字段表明 file 数组中有效元素的数量。
4. pid[MAX_BUFFER][MAX_SIZE]: 这是一个二维字符数组，用来存储进程的 PID（进程标识符）。这里，MAX_BUFFER 表示可以存储的 PID 的最大数量，而 MAX_SIZE 表示每个 PID 可以存储的最大字符数量。
5. prog[MAX_BUFFER][MAX_SIZE]: 这是一个二维字符数组，用来存储进程名。MAX_BUFFER 表示可以存储的进程名的最大数量，而 MAX_SIZE 表示每个进程名可以存储的最大字符数量。
6. file[MAX_BUFFER][MAX_SIZE]: 这是一个二维字符数组，用来存储文件名。MAX_BUFFER 表示可以存储的文件名的最大数量，而 MAX_SIZE 表示每个文件名可以存储的最大字符数量。

这个结构体用来保存一些与进程相关的信息，作为全局变量进行传递，接收文件输入，如 PID、进程名和文件名。使用这个结构体中的信息来进行内核模块的函数调用，如查找哪些进程正在访问特定的文件。

5 函数调用与说明

函数调用图如下：

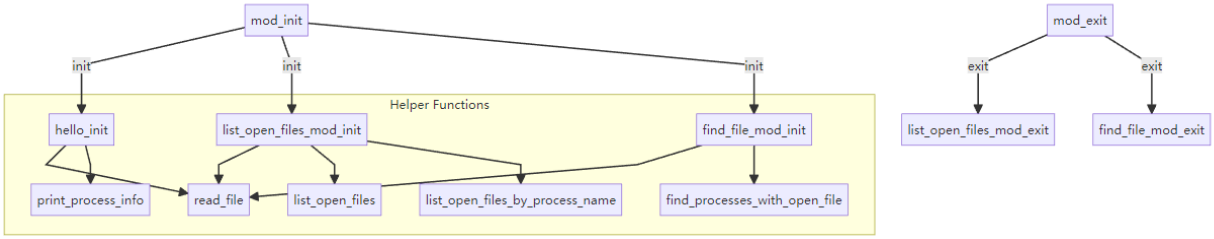


图 4: 函数调用图

具体函数的使用和说明如下：

- void read_file(void): 读取 targets 文件的内容，将进程 ID、进程名和文件名存储到对应的全局变量中。
- void find_processes_with_open_file(char file_name): 查找正在访问指定文件的进程，并打印相关信息。
- void list_open_files(int pid): 列出指定进程正在访问的所有文件和网络地址，并打印相关信息。
- void print_process_info(int pid): 打印给定进程的信息，如父进程和子进程等。
- int init_module(void): 初始化模块，调用 read_file 函数读取 targets 文件的内容，并进行后续处理。
- void cleanup_module(void): 清理模块，释放所有使用的资源。

5.1 读取 ~/targets 文件——read_file()

5.1.1 函数原型

```
char* kernel_fget(char* buf, int max_size, struct file *fp);
```

```c
char *kernel_fget(char* buf, int max_size, struct file *fp){
    int i = 0;
    int read_size;

    if (0 > max_size){
        printk(KERN_EMERG "param max_size is invalid\n");
        return NULL;
    }

    read_size = vfs_read(fp, buf, max_size, &(fp->f_pos));

    if (1 > read_size){
        return NULL;
    }

    while (buf[i++] != '\n' && i < read_size);
    buf[i - 1] = '\0';
    fp->f_pos += i - read_size;
    return buf;
}
```

图 5: vfs_read()

5.1.2 参数说明

- `buf`：读取的字符串存储位置
- `max_size`：最大读取的字符数 + 1
- `fp`：内核态函数 `flip_open` 打开文件后获取的文件描述符

5.1.3 函数功能

内核态从 `fp` 文件描述符中，读取最大 `max_size - 1` 字节的一行字符串，并存储在 `buf` 中。

5.1.4 返回值

如果正确读取，则返回 `buf` 指针，否则返回 `NULL`。

5.1.5 代码段说明

当读取的内容不是换行符并且当前已经读取的长度小于 `read_size`(`vfs_read(fp, buf, max_size, &(fp->f_pos))` 的返回值）时，不做任何处理。否则，截取到第 `i - 1` 的位置，并修改 `f_pos` 文件指针

的位置，并将 buf 返回。

5.2 read_file()

5.2.1 函数功能

读取 target 文件的内容，并将格式化文件的内容存储到相应的数据结构 file_struct 内。

5.2.2 部分代码解释

首先，当设置 mm_segment 类型变量为 KERNEL_DS 时，vfs_read 才可以读取内核内存中的内容。然后，恢复 mm_segment 类型变量为修改之前的值。

file_struct：pid_size 指明 pid 数组长度，pid[MAX_BUFFER][MAX_SIZE] 存储所需要读取的进程号。可以通过以下代码遍历结果：

```
for (i = 0; i < f_s->pid\_size; i++) {
    printk(KERN\_INFO "PID: %s\n", f_s->pid[i]);
}
```

在代码内，可以直接操作 f_s，例如：

```
f_s->pid\_size = 10;
strcpy(f_s->pid[0], "12345");
```

在这段代码中，将 f_s 的 pid 数组相应位置初始化，然后再写入对应的值。

5.2.3 应用方法

引入相应的头文件：

```
#include <linux/fs.h>
```

调用函数：

```
read\_file();
```

声明变量：

```
struct file\_struct *f\_s;
```

可以读取 f_s 中的相应内容。

5.3 记录 pid 或 prog 进程访问了哪些文件和 IP 地址

查询进程信息模块旨在获取操作系统中运行进程的详细信息。task_struct 结构体中包含了进程的诸多信息，如 PID（进程标识符）、进程名、进程状态等。此外，task_struct 结构体中还包含了指向进程打

开的文件和网络连接的指针。查询进程打开的文件和网络地址模块的目的是获取某个进程正在访问的文件以及建立的网络连接信息。

在 Linux 操作系统中，进程访问的文件和网络连接都被视为文件描述符（file descriptor, fd）。在 task_struct 结构体中，有一个指向进程打开的文件描述符表（fdtable）的指针。fdtable 是一个包含文件描述符的数组，数组中的每个元素都是一个指向 file 结构体的指针。file 结构体中包含了文件的路径、访问模式、访问时间等信息。对于网络连接，可以通过判断 file 结构体中的 inode 的模式是否为 S_ISSOCK 来确定是否为套接字文件。

在这个模块中，输入参数为一个整数，即目标进程的 PID。模块通过 PID 查找到对应的 task_struct 结构体实例，并提取进程信息。输出结果包括进程名、状态、文件路径、访问模式、访问时间以及网络连接的远程 IP 地址和端口号。以下是函数的调用过程：

```c
void list_open_files(int pid)
{
    struct task_struct *task = get_pid_task(find_vpid(pid), PIDTYPE_PID);

    if (!task) {
        printk(KERN_ERR "Error: Cannot find task with pid %d\n", pid);
        return;
    }

    printk(KERN_INFO "Process PID: %d\n", pid);
    printk(KERN_INFO "Process Name: %s\n", task->comm); // 打印进程名

    struct file *file_ptr;
    struct fdtable *files_table;
    struct path file_path;
    char *file_path_buf;

    spin_lock(&task->files->file_lock);
    files_table = files_fdtable(task->files);
    int fd;

    for (fd = 0; fd < files_table->max_fds; fd++) {
        file_ptr = files_table->fd[fd];
        if (!file_ptr)
            continue;

        if (S_ISSOCK(file_ptr->f_inode->i_mode)) {
            struct socket *sock = sock_from_file(file_ptr);
            if (sock && sock->sk) {
                struct inet_sock *inet = inet_sk(sock->sk);
                char ipbuf[INET_ADDRSTRLEN];
                snprintf(ipbuf, INET_ADDRSTRLEN, "%pI4", &inet->inet_daddr);

                if (inet->inet_daddr != 0) {
                    printk(KERN_INFO "远程 IP 地址: %s:%hu\n", ipbuf, ntohs(inet->inet_dport));
                } else {
                    //printk(KERN_INFO "IP 地址为 0.0.0.0\n");
                }
            }
        }
```

图 6: list_open_files

1. 使用 `get_pid_task` 函数获取与给定 PID 关联的 task_struct 结构。这个结构代表一个特定的进程。
2. 检查是否成功找到了对应的进程（task）。如果找不到，打印一个错误消息，并退出函数。
3. 打印出进程的 PID 和名字。
4. 通过 `files->file_lock` 加锁，保护进程打开的文件的并发访问。
5. 获取进程打开的所有文件描述符的表格（fdtable）。
6. 遍历进程打开的所有文件描述符，获取并打印出每个打开的文件的路径、访问模式和访问时间。
    - 首先，它会检查当前文件描述符对应的 file 结构是否存在。如果不存在，则跳过此文件描述符。

- 然后，它检查文件是否是套接字类型。如果是，获取套接字的远程 IP 和端口，并打印出来。
- 最后，获取当前文件描述符对应的文件路径。它使用 dentry_path_raw 函数获取路径，并将其存储在 file_path_buf 中。然后，它打印出文件的路径、访问模式和访问时间。

7. 释放锁，并打印一个分隔线，表示此进程的文件列表已完成。

5.4 记录文件被哪些进程访问

find_processes_with_open_file 是一个内核函数，它扫描 Linux 系统的进程空间以查找当前正在访问特定文件的所有进程。

5.4.1 函数定义

```
void find_processes_with_open_file(const char *target_filename)
```

5.4.2 参数

- target_filename: 一个字符串指针，表示当前所有运行进程打开的文件中需要查找的文件的绝对路径。

5.4.3 调用过程

函数 find_processes_with_open_file 接受一个参数，即目标文件的路径 target_filename。

首先,在函数体内部,定义了一些所需的变量,包括进程结构 task_struct、路径缓冲 file_path_buf、目标文件路径 target_file_path 和目标文件的索引节点 target_file_inode。

然后尝试通过 kern_path 函数获取 target_filename 文件的路径，如果成功，则将对应的索引节点保存到 target_file_inode 中。

接着使用 for_each_process(task) 迭代器遍历系统中的所有进程。对每一个进程，获取其打开文件的描述符表。这是通过进程的 task->files 字段，和 files_fdtable 函数来完成的。

进一步对描述符表中的每一个文件描述符,通过 files_table->fd[fd] 获取对应的文件指针 file_ptr。判断 file_ptr 是否为空，若为空则跳过当前迭代。

然后获取文件指针对应的路径 file_path，并将其转化为字符串格式存储在 file_path_buf 中。

如果 file_path_buf 不为空，则通过 dentry_path_raw 函数获取文件的路径字符串。如果文件的索引节点和目标文件的索引节点相同，或者文件的路径和目标文件的路径相同，说明当前进程正在访问目标文件。

然后进一步获取文件的访问模式（读取或写入）和访问时间，并格式化为易读的字符串。

最后，通过 printk 函数将进程 ID、进程名、目标文件名、访问模式和访问时间打印到内核日志中。

解锁文件锁，并在遍历完所有的进程后结束。

```c
void find_processes_with_open_file(const char *target_filename)
{
    struct task_struct *task;
    char *file_path_buf;
    struct path target_file_path;
    struct inode *target_file_inode = NULL;

    if (kern_path(target_filename, 0, &target_file_path) == 0) {
        target_file_inode = target_file_path.dentry->d_inode;
    }

    for_each_process(task) {
        struct file *file_ptr;
        struct fdtable *files_table;
        struct path file_path;

        spin_lock(&task->files->file_lock);
        files_table = files_fdtable(task->files);
        int fd;

        for (fd = 0; fd < files_table->max_fds; fd++) {
            file_ptr = files_table->fd[fd];
            if (!file_ptr)
                continue;

            file_path = file_ptr->f_path;
            file_path_buf = (char *)__get_free_page(GFP_KERNEL);
            if (file_path_buf) {
                char *p = dentry_path_raw(file_path.dentry, file_path_buf, MAX_PATH_LEN - 1);
                if (!IS_ERR(p)) {
                    if (file_ptr->f_inode && target_file_inode && file_ptr->f_inode == target_file_inode) {
                        // inode comparison logic
                    } else if (strcmp(p, target_filename) == 0) {
                        // 获取文件访问模式
                        char access_mode[8] = "";
                        if (file_ptr->f_mode & FMODE_READ) {
                            strcat(access_mode, "R");
                        }
                        if (file_ptr->f_mode & FMODE_WRITE) {
                            strcat(access_mode, "W");
                        }

                        // 获取文件访问时间
```

图 7: find_process_with_open_file

5.4.4 错误处理

此功能包括对潜在错误的多项检查：

1. 它检查 `dentry` 和指针是否为 `NULL`，这表明检索文件的元数据时出现问题。
2. 它检查函数 `d_absolute_path` 返回错误指针，这会在检索绝对文件路径时提示错误。

在这两种情况下，都会记录一条错误消息，并且该函数会继续处理下一个文件。

5.4.5 假设和限制

- 此函数假定 `target_filename` 参数指向表示文件绝对路径的有效空终止字符串。
- 它假定内核的任务列表在调用时是准确的。

- 它受调用进程的权限限制。如果调用进程缺少访问某些文件或进程的必要权限，则这些文件或进程将不会包含在搜索中。

5.5 展示进程之间的父子关系

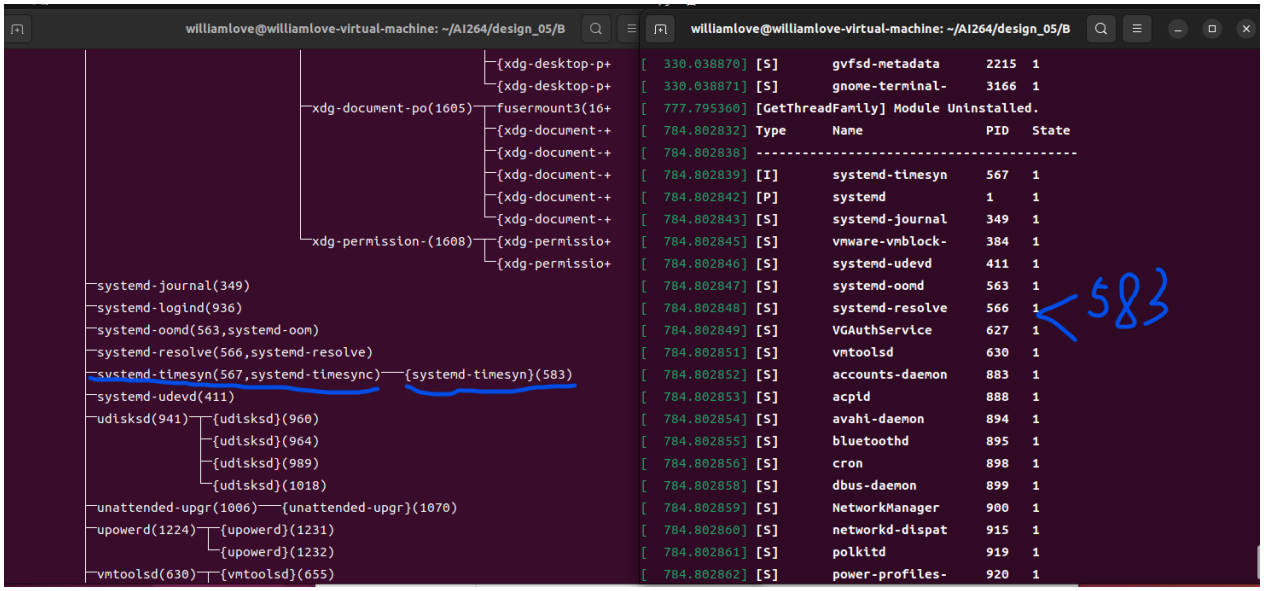


图 8: pstree

Pstree 的局限性：在 Linux 中，进程树（pstree 命令打印的结果）并不能完全准确地反映出进程之间的关系。pstree 命令的输出中，被 {} 括起来的进程并不一定是它们前面的同名进程的子进程，而是进程组内的一部分。他们的父进程实际上是启动这个进程组的进程，也就是进程组头进程。

为了解决 pstree 带来的问题，我们设计了 `print_process_info()` 函数来打印进程间的父子关系。

5.5.1 函数定义

```
void print_process_info(pid_t pid)
```

5.5.2 背景

该函数的目的在于打印出指定进程 ID 的进程信息。在 Linux 中，所有进程都是通过 `task_struct` 结构体来管理的。在这个函数中，你将了解到 `task_struct` 结构体的一些基本成员以及如何通过内核链表来遍历特定的成员。

5.5.3 原理

`print_process_info()` 函数首先根据指定的进程 ID 获取对应的 `task_struct` 结构体。然后，它使用 `printk()` 函数来打印进程的 ID 和名称。

在打印父进程和子进程的信息时,函数使用了 `list_for_each()` 宏来遍历链表。这是因为在 `task_struct` 中, 父进程和所有子进程都是通过内核链表来链接的。遍历过程中, 使用了 `list_entry()` 宏来将链表节点中的 `sibling` 成员转换为整个节点结构体的指针。这个函数的实现中, 主要用到了三个 `task_struct`

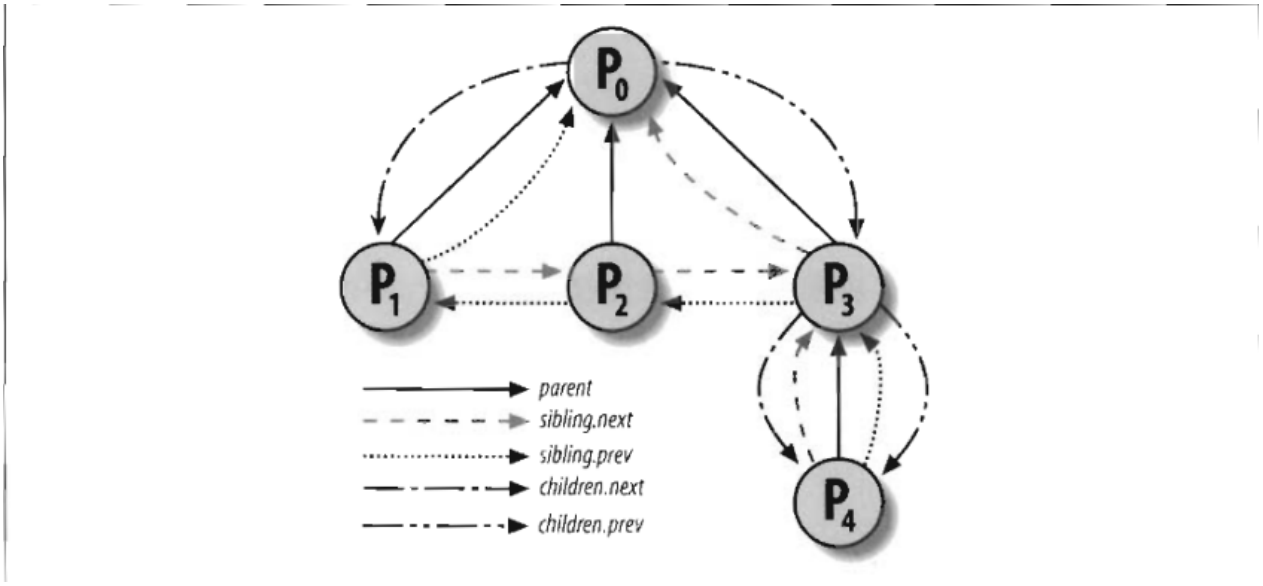


图 9: relation

中的成员:

- `pid`: 进程的 ID
- `comm`: 进程的名称
- `parent`: 指向父进程的指针
- `children`: 指向子进程链表的头部
- `sibling`: 用于把当前进程插入到兄弟链表中

5.5.4 参数

- `pid`: 指定的进程 ID，类型为 `pid_t`。

5.5.5 流程

1. 调用 `find_vpid()` 函数，找到与提供的进程 ID 对应的虚拟 PID。
2. 调用 `pid_task()` 函数，根据找到的虚拟 PID 得到对应的进程描述符。
3. 如果进程描述符为 `NULL`，打印错误信息并结束函数。

4. 否则，打印出进程的 ID 和名字。

5. 判断进程是否有父进程，如果有，打印出父进程的 ID 和名字。

6. 使用 ttlist_for_each() 宏和 `list_entry()` 宏，遍历进程的父进程的所有子进程（即当前进程的所有兄弟进程），打印出兄弟进程的 ID 和名字。

7. 同样地，遍历当前进程的所有子进程，打印出子进程的 ID 和名字。

5.5.6 调用

调用 `print_process_info()` 函数，需要传入一个进程 ID。例如：

```
pid_t pid = 12345;
print_process_info(pid);
```

这将打印出进程 12345 的相关信息，包括其本身的 ID 和名称，其父进程的 ID 和名称，以及其所有子进程和兄弟进程的 ID 和名称。

6 性能测试

我们探讨了内核模块的性能分析方法，并在特定条件下对其进行了测试。为了衡量内核模块的性能，我们选择了内存和 CPU 两个方面进行深入研究。我们的目标是提供一个相对简单的但有足够深度的方法，来评估内核模块的性能。

6.1 方法

6.1.1 内存使用量

为了测量加载内核模块时的内存使用量，我们采用 `lsmod` 命令并将其与特定模块名称进行匹配 `lsmod | grep [module name]`。结果中的 `Size` 代表模块加载时的内存使用量（以字节为单位）。

我们引用了一个来源 [1]，这个来源指出我们无法准确地获取到内核模块实际的内存占用，因为内核和内核模块都处于同一地址空间，无法区分哪些内存是由内核自身占用，哪些是由内核模块占用。

我们也尝试了 `sys_info` 和 `si_meminfo(&sys)`，以便获取系统内存使用情况，包括总内存，可用内存，以及系统缓存和共享内存等。具体来说，我们可以了解到系统的总内存大小、当前可用的内存大小、系统缓存和共享内存的使用情况等。但是这种方法并不能得出具体内核模块的内存占用。

6.1.2 CPU 使用时间

可以使用 `vtune`、`trace32`、`oprofile` 等工具来进行 CPU 使用的分析，但是这些工具对我们来说很难上手。在 CPU 使用的分析中，我们决定采用一个相对简单但仍可提供相对性能比较的方法。我们测量函数执行前后的时间戳，以此作为性能比较的参考。

我们也检查了 CPU 的时间片长度，以便了解内核模块执行是否可能被打断。我们采用 `cat...` 命令来查看时间片长度，发现其为 100ms。

```
wqs@wqs-virtual-machine:~/v7$ cat /proc/sys/kernel/sched_rr_timeslice_ms
100
```

图 10: time-cut length

6.1.3 CPU 占用率

CPU 占用指的是计算机中央处理器（CPU）正在执行某个程序时所占用的系统资源的百分比。它是衡量计算机性能的一个重要指标，通常用于评估一个程序的效率和优化性能。

/proc/stat/结构理解

在 Linux 系统中，/proc/stat 文件提供了有关系统 CPU 和其他资源使用情况的信息。/proc/stat 文件中的第一行包含的是有关 CPU 总体使用情况的信息。

```
user    Time spent with normal processing in user mode.
nice    Time spent with niced processes in user mode.
system  Time spent running in kernel mode.
idle    Time spent in vacations twiddling thumbs.
iowait  Time spent waiting for I/O to completed. This is considered idle time too.
irq Time spent serving hardware interrupts.
softirq Time spent serving software interrupts.
steal   Time spent stolen by other operating systems running in a virtual environment.
guest   Time spent for running a virtual CPU or guest OS under the control of the kernel.
```

6.1.4 kernal_read() 读取

需要调试的驱动程序中读写文件数据，比如说当驱动需要记录的日志比较多的情况下，可以将 printk() 函数打印的信息都写到文件做后续分析。在 kernel 中操作文件没有标准库可用，需要利用 kernel 的一些函数，这些函数主要有：filp_open() filp_close(), kernel_read()，kernel_write()。

6.1.5 监控内核模块 CPU 占用

我们利用 kernel_read 读取系统/proc/stat 中提供的 CPU 时间的占用信息来监控当前内核模块的 CPU 使用情况。由于模块运行一次的时间很多，小于一个时间片的长度，所以直接读取得到的信息是没有意义的。所以我们通过将内核模块重复运行 1000 次。然后在内核模块运行过程中，每个一段固定的时间来读取/proc/stat 中的信息。然后计算在这一个时间间隔内，内核模块的 CPU 使用情况。其中当前内核模块的 CPU 利用率：/proc/stat 中 system 项下的时间差值与这个时间段时间的比值。

6.2 结果

6.2.1 内存

我们通过 `lsmod` 命令测量的模块加载时的内存使用量如下图所示：

```
wqs@wqs-virtual-machine:~/v7$ lsmod | grep os
os                     49152  0
```

图 11: MEM usage

6.2.2 CPU 执行模块用时

函数执行时间的测试结果显示，执行一次函数大概需要 900 微秒，这是时间片长度的不到 1/100。这意味着执行这个函数的过程大概率不会被打断。这个时间就是执行用时（以防万一，多次执行）和系统

```
Goodbye, world!
Elapsed time: 858 us
```

图 12: func-runtime

状态有关，且短时间时间恒定，总体来看稳定在 900us 左右。

我们进行了更进一步的测试，为了准确对比，排除偶然性，每次循环执行函数 100 次，测试结果如下：

- 5 个进程：49647 微秒
- 10 个进程：50601 微秒
- 20 个进程：51808 微秒

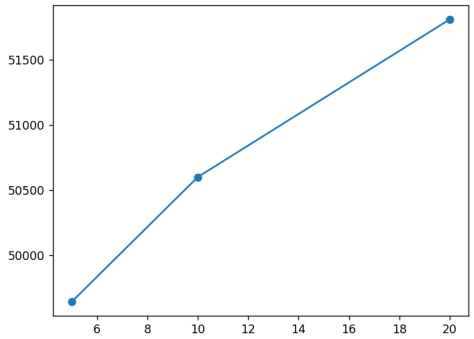


图 13: CPU usage

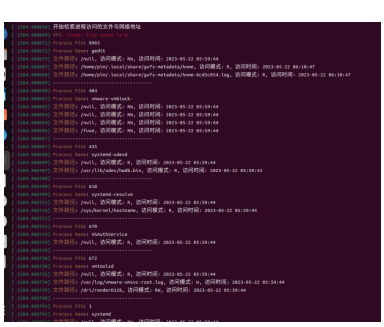
这些结果显示，随着进程数量的增加，函数执行的时间也会相应增长。

6.2.3 CPU 占用率

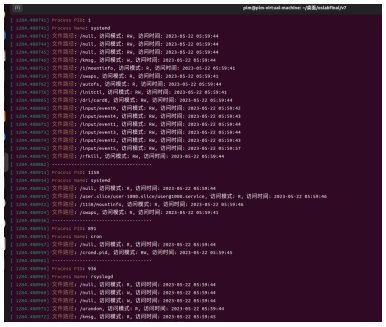
我们也借助 proc/stat 提供的 CPU 信息进行了占用率的测量，在 5、10、20 个进程的情况下，因为内核模块的执行时间小于一个时间片，我们多次循环内核模块中的函数，以减少随机扰动。

结果显示，随着进程监控数的增加，对应模块加载执行的 CPU 占用率几乎不变，约为 0.70(+-0.13)，而 CPU 占用时间显著减少，这与我们上述的分析相吻合。这里我们仅使用内核态 CPU 占用率差值进行分析。

7 结果展示



(a) 进程访问文件信息 1

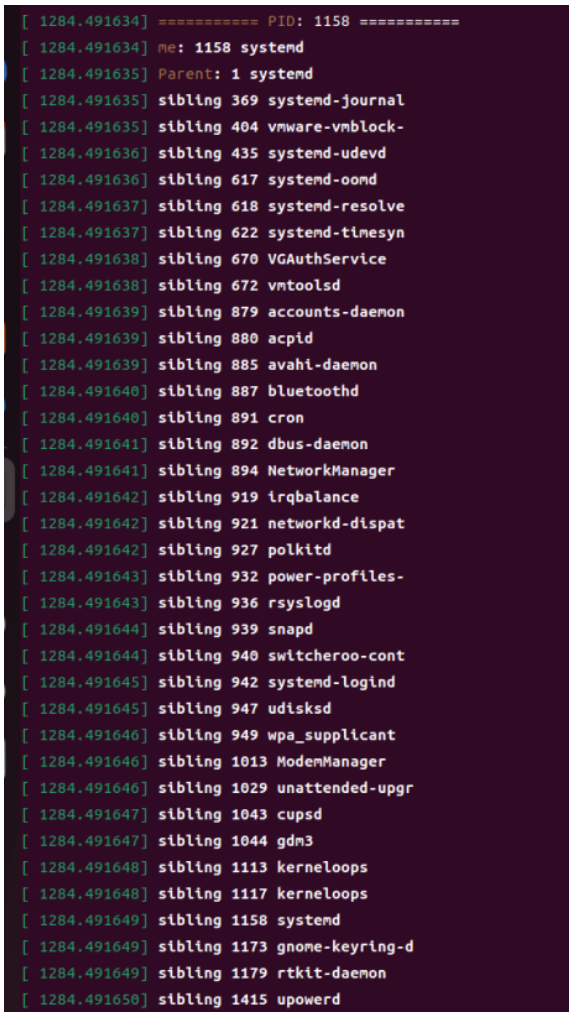


(b) 进程访问文件信息 2

(c) 文件被哪些进程访问信息

图 14: 运行结果

(a) 进程关系 1



(b) 进程关系 2

图 15: 进程之间关系展示

8 具体贡献

贡献比例：庞力铭 1.1 吴权达 1.1 王开苇 1 文宇华 1 李威汉 1 闻其帅 0.9

8.1 庞力铭

- 编写了进程查找文件和 IP 的模块代码，提出并编写了文件记录进程模块的一种思路
- 负责最终模块的整合、测试和调试。
- 整合相关文档，撰写了总体报告的相关部分，包括模块架构、功能描述、函数调用图等。
- 协调组内成员，监督项目进展，并确保任务按时完成。

8.2 吴权达

- 实现了进程树遍历的初步版本，用于遍历进程的 sibling 进程。
- 编写了记录文件被哪些进程访问的内核模块。
- 实现了记录文件的并发访问功能
- 深入研究和理解了 Linux 内核源码和 api，并为项目的代码实现提供了技术支持。
- 对文件并发访问的原理和机制有了深入的了解，对于项目中的并发访问功能提供了宝贵的见解和贡献。

8.3 王开苇

- 编写了读取 ‘targets‘ 文件的模块代码。
- 负责文件读取模块的测试和调试工作。
- 设计了 file_struct 结构体，供项目其他函数调用
- 提供了关于文件读取模块的性能优化建议和改进措施。

8.4 文宇华

- 完成了获取父子进程之间关系模块的代码编写
- 编写内存占用和 CPU 占用内核模块
- 参与了分析内核模块性能的调研

8.5 李威汉

- 完成了获取进程之间关系模块的代码编写，以及该模块的正确性测试。
- 参与了分析内核模块性能的调研与代码编写的相关工作。
- 提供了对性能优化的建议和改进方案。

8.6 闻其帅

- 探索性能分析的可行方法并进行性能分析
- 参与模块的测试和调试工作
- 提供了关于性能优化的建议和改进措施

9 讨论

9.1 其他性能测试方法——perf

9.1.1 介绍

perf 的原理是：依据给定的事件的发生（最常见的是 CPU 周期）进行采样（可设置采样率），采样得到的信息有：目标进程或函数占用 CPU 的时间、申请的缓冲区大小等

9.1.2 功能

可以让 perf 对所有进程（包括整个内核）进行监测，可以看到所有函数对 CPU 的占用情况

内核模块不同于用户进程，它在加载后成为内核代码的一部分，不能像用户进程一样作为一个独立的整体被分析想分析内核模块的性能应该直接从函数入手

使用 perf probe 命令可以自动在内核模块的代码（装载到内核后的二进制代码）中的指定函数中添加探针点

探针点本质上一段额外的代码，在指定函数执行时这段代码也会执行添加探测点之后就可以使用 perf record 并指定探测点为事件

9.1.3 问题

问题 1 我们的系统内核不支持 perf mem 命令；

问题 2 内核模块并不是进程，所以想用 perf 就必须人为指定一些探测点作为 perf 事件（往往是内核模块中某些函数的调用），探针的嵌入和回收较为复杂，我们还没有完成。

9.2 Linux 内核链表和 list 遍历分析

9.2.1 从进程到线程一 Linux 中的 task_struct 结构分析

Linux 进程管理之 task_struct 结构体。除了 0 号进程以外，其他进程都是有父进程的。全部进程其实就是一颗进程树，相关成员变量如下所示:

- parent: 指向其父进程。当它终止时，必须向它的父进程发送信号。
- children: 指向子进程链表的头部。链表中的所有元素都是它的子进程。
- sibling: 用于把当前进程插入到兄弟链表中。

通常情况下，real_parent 和 parent 是一样的，但是也会有另外的情况存在。例如，bash 创建一个进程，那进程的 parent 和 real_parent 就都是 bash。如果在 bash 上使用 GDB 来 debug 一个进程，这个时候 GDB 是 parent，bash 是这个进程的 real_parent。

9.2.2 linux 内核链表

深入分析 Linux 内核链表。链表数据结构的定义很简单: list_head 结构包含两个指向 list_head 结构的指针 prev 和 next，由此可见，内核的链表具备双链表功能，实际上，通常它都组织成双循环链表。Linux 用头指针的 next 是否指向自己来判断链表是否为空

9.2.3 list_for 内核链表遍历

在 Linux 内核中，list_entry 宏可以将链表节点中的某个成员（一般是指向下一个节点的指针）转换为整个节点结构体的指针。通过这种方式，我们可以方便地访问链表节点结构体中的所有成员变量。

9.2.4 list_for_each_entry 和 list_entry

linux 操作系统内核中 list_for_each_entry 和 list_entry 两个函数有什么区别:

list_for_each_entry 和 list_entry 是 Linux 内核中的两个常用宏，它们都用于遍历链表。

● list_for_each_entry(pos, head, member) 宏是一个 for 循环，在循环体内使用 pos 访问当前节点，head 是链表头指针，member 是链表节点结构体中的成员指针。这个宏适合在访问每个节点时需要使用节点指针的情况下使用。

● list_entry(ptr, type, member) 宏其实是将链表节点结构体中的某个成员（即 member）转换为该节点的起始地址（即结构体指针），其中 ptr 是指向该节点成员的指针，type 表示该节点所在的结构体类型。这个宏适合在已经有节点成员指针（ptr）的情况下，需要获取整个节点结构体的指针的情况下使用。

list_entry 会将 ptr 转换为包含它的结构体指针，并返回这个指针，而使用这个结构体指针可以直接用于访问结构体中的数据。

综上所述，list_for_each_entry 主要是用来遍历链表，并且在遍历过程中需要访问每个节点的指针；list_entry 则主要用来将链表节点中的某个成员转换为整个节点结构体的指针。

（“list_entry 则主要用来将链表节点中的某个成员转换为整个节点结构体的指针”这句话的含义如下:

在 Linux 内核中，list_entry 宏可以将链表节点中的某个成员（一般是指向下一个节点的指针）转换为整个节点结构体的指针。通过这种方式，我们可以方便地访问链表节点结构体中的所有成员变量。

9.3 文件描述符 fd 和文件指针 filp

每个进程在 PCB (Process Control Block) 即进程控制块中都保存着一份文件描述符表，文件描述符就是这个表的索引，文件描述表中每个表项都有一个指向已打开文件的指针。

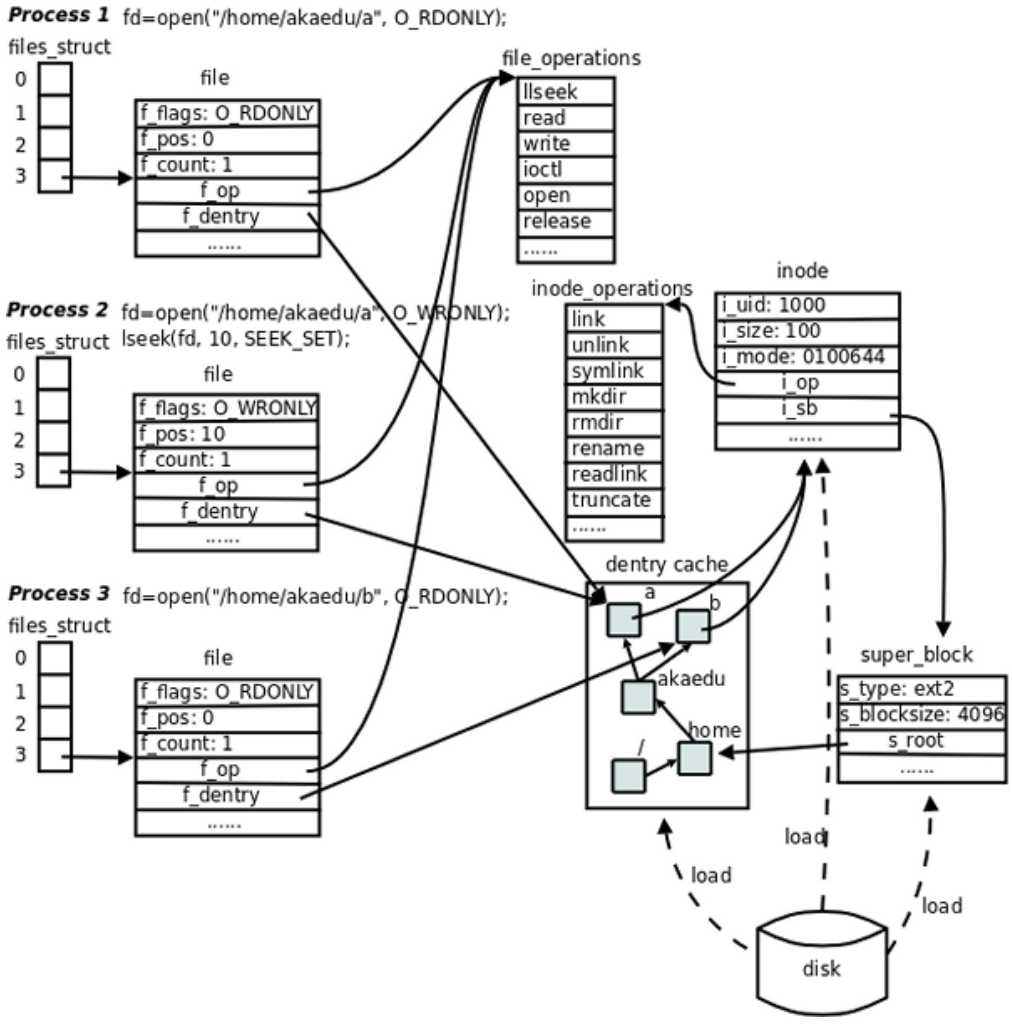


图 16: Linux 内核的 VFS 子系统

file tablefile table 是全局唯一的表，由系统内核维护。这个表记录了所有进程打开的文件的状态 (是否可读、可写等状态)，同时它也映射到 inode table 中的 entry。

当程序向内核发起 system call open()，内核将会

1. 允许程序请求

2. 创建一个 entry 插入到 file table，并返回 file descriptor

3. 程序把 fd 插入到 fds 中。

当程序再次发起 read()system call 时，需要把相关的 fd 传给内核，内核定位到具体的文件 (fd $->$ file table $->$ inode table) 向磁盘发起读取请求，再把读取到的数据返回给程序处理。

9.4 总体性能测量方法

9.4.1 利用 sys_info 测量内存使用

1. 理解 sys_info

”struct sysinfo” 是 Linux 内核中的一个 C 结构体，用于提供关于系统硬件和软件资源的信息。它包含了以下的各个字段，可以通过使用 ”sysinfo” 系统调用来获取这些信息。

在使用该系统调用时，需要传递一个指向 ”struct sysinfo” 结构体的指针作为参数，该结构体将被填充为系统的信息。

2. 理解 si_meminfo 函数

在 Linux 中，si_meminfo(&sys) 函数是一种用于获取系统内存信息的方式。它是在内核中实现的，可以通过调用该函数来获取系统内存使用情况的详细信息。

该函数的参数是一个指向 ”struct sysinfo” 结构体的指针，该结构体包含了关于系统的内存使用情况的信息。si_meminfo(&sys) 函数会填充该结构体的相关字段，以反映系统的内存使用情况。

具体来说，si_meminfo(&sys) 函数会将系统的内存信息填充到 ”struct sysinfo” 结构体的以下字段中

```
void si_meminfo(struct sysinfo *val)
{
        val->totalram = totalram_pages;
        val->sharedram = 0;
        val->freeram = global_page_state(NR_FREE_PAGES);
        val->bufferram = nr_blockdev_pages();
        val->totalhigh = totalhigh_pages;
        val->freehigh = nr_free_highpages();
        val->mem_unit = PAGE_SIZE;
}
```

3. 从 sys_info 中获取内存使用情况

首先创建”struct sysinfo sys” 变量，实际上是一个指向 ”struct sysinfo” 结构体的指针，用于接收从系统调用中返回的系统信息。在调用 ”sysinfo” 时指定将系统信息存储在哪个变量中。

将”struct sysinfo sys” 变量传给 si_meminfo 函数。通过调用该函数并解析返回的结构体，我们可以了解系统内存的使用情况，包括系统的总内存大小、当前可用的内存大小、系统缓存和共享内存的使用情况等。

具体实现在附录 1

9.4.2 利用 proc/stat/测量 CPU 使用

一种获取 CPU 使用情况的方法是使用 ‘/proc/stat‘ 文件。该文件包含了当前系统的各个 CPU 的运行情况，包括 CPU 时间以及一些统计信息等。具体来说，可以通过解析该文件的内容，计算出 CPU 的使用情况。

具体代码在附录 2

10 参考文献：

[1] https://www.kernel.org/doc/html/v5.7/admin-guide/mm/sysfs-rmap.html

[2] https://www.kernel.org/doc/html/latest/kbuild/modules.html

[3] https://github.com/sysprog21/lkmpg

11 附录

11.1 附录 1——sys_info 获取内存使用量

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/mm.h>
#include <asm/page.h>
static int sysinfo_init()
{
    struct sysinfo sys;
    si_meminfo(&sys);
    printk("free mem is: %ld\n", (sys.freeram << (PAGE_SHIFT - 10)));
    return 0;
}

static void sysinfo_exit()
{
    printk("exit module sysinfo_test\n");
}
module_init(sysinfo_init);
module_exit(sysinfo_exit);
```

11.2 附录 2——利用 proc/stat/测量 CPU 使用

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/delay.h>

MODULE_LICENSE("GPL");

static int cpu_usage_init(void)
{
    struct file *proc_stat;
    char buf[256];
    int len;
    unsigned long long user, nice, system, idle, iowait, irq, softirq, steal, guest, guest_nice;
    unsigned long long total1, total2, busy1, busy2, busy, usage;

    // 打开 /proc/stat 文件
    proc_stat = filp_open("/proc/stat", O_RDONLY, 0);
    if (!proc_stat || IS_ERR(proc_stat)) {
        printk(KERN_ERR "Failed to open /proc/stat\n");
        return -1;
    }

    // 读取 /proc/stat 文件的内容
    len = kernel_read(proc_stat, buf, sizeof(buf), 0);
    if (len <= 0) {
        printk(KERN_ERR "Failed to read /proc/stat\n");
        filp_close(proc_stat, NULL);
        return -1;
    }

    filp_close(proc_stat, NULL);

    // 解析 /proc/stat 文件的内容，获取 CPU 时间信息
    sscanf(buf, "cpu %llu %llu %llu %llu %llu %llu %llu %llu %llu %llu", &user, &nice, &system, &idl

    // 计算 CPU 的使用情况
    total1 = user + nice + system + idle + iowait + irq + softirq + steal;
```

```
    busy1 = user + nice + system + irq + softirq + steal;

    printk("user is %llu",user);
    msleep(500);    // 等待 500ms

    proc_stat = filp_open("/proc/stat", O_RDONLY, 0);
    if (!proc_stat || IS_ERR(proc_stat)) {
        printk(KERN_ERR "Failed to open /proc/stat\n");
        return -1;
    }
    len = kernel_read(proc_stat, buf, sizeof(buf), 0);
    if (len <= 0) {
        printk(KERN_ERR "Failed to read /proc/stat\n");
        filp_close(proc_stat, NULL);
        return -1;
    }
    filp_close(proc_stat, NULL);
    sscanf(buf, "cpu %llu %llu %llu %llu %llu %llu %llu %llu %llu %llu", &user, &nice, &system, &idl
    total2 = user + nice + system + idle + iowait + irq + softirq + steal;
    busy2 = user + nice + system + irq + softirq + steal;
    busy = busy2 - busy1;
    usage = (busy * 100) / (total2 - total1);
    printk("cpu total1 is %llu",total1);
    printk("cpu total2 is %llu",total2);
    printk("cpu busy1 is %llu",busy1);
    printk("cpu busy2 is %llu",busy2);
    printk("CPU usage: %llu%%\n", usage);
    return 0;
}
static void cpu_usage_exit(void)
{
    printk(KERN_INFO "CPU usage module unloaded.\n");
}
module_init(cpu_usage_init);
module_exit(cpu_usage_exit);
```