实验1——内核API

姓名：王越洋 学号：22009200894

**一、实验目的**

通过本实验，掌握操作系统内核模块的设计与实现，熟悉掌握：

（1）内核链表的创建与操作

（2）内核线程的创建与同步

（3）进程信息的获取与操作

**二、实验环境**

操作系统：OpenKylin

内核版本：Linux 5.10.0

实验平台：虚拟机，使用 GDB-Pod 环境

相关工具：GCC 编译器，Makefile，insmod 和 rmmod 用于加载和卸载内核模块

**三、实验内容**

设计并实现一个内核模块，包括以下功能：

（1）创建一个内核链表，用于存储当前系统中的进程 PID 和进程名。

（2）创建两个内核线程：

（3）线程1：遍历进程链表，并将进程 PID 和进程名加入到内核链表中。

（4）线程2：不断从链表中取出节点并打印节点内容，最后删除节点。

（5）当模块卸载时，停止所有线程并清理链表。

**四、操作步骤**

**1.代码编写**

(1) 编写 kernel\_module.c 文件

包含模块的初始化、两个线程的实现以及资源的清理。

1. #include <linux/module.h>
2. #include <linux/kernel.h>
3. #include <linux/init.h>
4. #include <linux/list.h>
5. #include <linux/sched.h>
6. #include <linux/kthread.h>
7. #include <linux/rcupdate.h>
8. #include <linux/slab.h>
9. #include <linux/delay.h>
10. #include <linux/sched/signal.h>
11. #include <linux/string.h>
12. MODULE\_LICENSE("GPL");
13. MODULE\_AUTHOR("FuShengyuan");
14. MODULE\_DESCRIPTION("A simple kernel module with a list and two threads.");
15. static struct list\_head my\_list;  *// 内核链表头*
16. static struct task\_struct \*thread1, \*thread2;  *// 内核线程*
17. spinlock\_t lock;  *// 自旋锁，用于多线程同步*
18. *// 定义链表节点结构，包含进程PID和进程名*
19. struct pid\_node {
20. int pid;
21. char comm[16];
22. struct list\_head list;  *// 内核链表结构*
23. };
24. *// 线程1：遍历进程链表并将进程信息加入内核链表*
25. static int thread1\_func(void \*data) {
26. struct task\_struct \*task;
27. struct pid\_node \*node;
28. while (!kthread\_should\_stop()) {
29. rcu\_read\_lock();  *// 保护进程链表*
30. for\_each\_process(task) {
31. *// 分配链表节点内存*
32. node = kmalloc(sizeof(\*node), GFP\_KERNEL);
33. if (!node)
34. continue;
35. node->pid = task->pid;
36. strncpy(node->comm, task->comm, sizeof(node->comm));
37. *// 加锁以安全地操作链表*
38. spin\_lock(&lock);
39. list\_add\_tail(&node->list, &my\_list);  *// 插入链表*
40. spin\_unlock(&lock);
41. }
42. rcu\_read\_unlock();
43. msleep(5000);  *// 每隔5秒遍历一次*
44. }
45. return 0;
46. }
47. *// 线程2：从链表中取出并打印进程信息，最后删除节点*
48. static int thread2\_func(void \*data) {
49. struct pid\_node \*node, \*tmp;
50. while (!kthread\_should\_stop()) {
51. spin\_lock(&lock);
52. list\_for\_each\_entry\_safe(node, tmp, &my\_list, list) {
53. printk(KERN\_INFO "PID: %d, Command: %s\n", node->pid, node->comm);
54. list\_del(&node->list);  *// 从链表中删除节点*
55. kfree(node);  *// 释放节点内存*
56. }
57. spin\_unlock(&lock);
58. msleep(5000);  *// 每隔5秒打印一次*
59. }
60. return 0;
61. }
62. *// 模块初始化函数*
63. int kernel\_module\_init(void) {
64. printk(KERN\_INFO "List and thread module init\n");
65. INIT\_LIST\_HEAD(&my\_list);  *// 初始化链表*
66. spin\_lock\_init(&lock);  *// 初始化自旋锁*
67. *// 创建线程1*
68. thread1 = kthread\_create(thread1\_func, NULL, "thread1");
69. if (IS\_ERR(thread1)) {
70. printk(KERN\_ERR "Failed to create thread1\n");
71. return PTR\_ERR(thread1);
72. }
73. wake\_up\_process(thread1);
74. *// 创建线程2*
75. thread2 = kthread\_create(thread2\_func, NULL, "thread2");
76. if (IS\_ERR(thread2)) {
77. printk(KERN\_ERR "Failed to create thread2\n");
78. kthread\_stop(thread1);
79. return PTR\_ERR(thread2);
80. }
81. wake\_up\_process(thread2);
82. return 0;
83. }
84. *// 模块卸载函数*
85. void kernel\_module\_exit(void) {
86. struct pid\_node \*node, \*tmp;
87. printk(KERN\_INFO "List and thread module exit\n");
88. *// 停止线程*
89. if (thread1)
90. kthread\_stop(thread1);
91. if (thread2)
92. kthread\_stop(thread2);
93. *// 清理链表*
94. spin\_lock(&lock);
95. list\_for\_each\_entry\_safe(node, tmp, &my\_list, list) {
96. list\_del(&node->list);
97. kfree(node);
98. }
99. spin\_unlock(&lock);
100. }
101. module\_init(kernel\_module\_init);
102. module\_exit(kernel\_module\_exit);

**2.使用vim将代码上传至虚拟机**

输入`vi 文件名`查看文件，按`ESC`进入编辑模式，而后输入`dd`删除光标当前行及以下的部分，转到本机编辑好的文件，复制，右击鼠标即可将当前剪贴板里的内容粘贴进去，按`ESC`，输入`:wq`保存并退出。

**3. 编译内核模块**

使用`make`编译模块

**4.运行模块并查看日志**使用`insmod`命令加载模块，使用`dmesg`查看模块的日志输出，确认线程1和线程2的运行情况。

**5. 卸载模块**

使用`rmmod`卸载内核模块:

sudo rmmod kernel\_module

**五、代码分析**

**1.模块的初始化**

创建并启动两个内核线程，初始化内核链表和自旋锁。

（1）链表初始化：使用 INIT\_LIST\_HEAD(&my\_list) 初始化链表头，使链表可以被用于存储节点（进程信息）。

（2）自旋锁初始化：spin\_lock\_init(&lock) 初始化自旋锁，确保链表在多线程操作中不会发生竞争问题。

（3）创建内核线程：通过 kthread\_create 创建两个内核线程。线程创建成功后，调用 wake\_up\_process 启动线程。

线程1：用于遍历系统进程并将信息存入链表。

线程2：用于从链表中取出信息并打印。

（4）错误处理：如果创建线程2失败，代码会停止已经启动的线程1以避免资源浪费。

**2.线程1**

遍历系统进程并将进程信息（PID和进程名）存入链表。

（1）进程遍历：使用for\_each\_process(task)遍历系统中每个进程。task\_struct是Linu内核中描述进程的结构体，task->pid是进程ID，task->comm是进程的名字。

（2）动态分配节点：为每个进程动态分配pid\_node结构体内存，存储PID和进程名。使用 kmalloc 分配内存，GFP\_KERNEL 参数表示正常的内存分配方式。

（3）链表操作：通过list\_add\_tail(&node->list, &my\_list)将新的节点插入到链表尾部，保证节点按顺序插入。

（4）同步机制：在添加节点时使用spin\_lock(&lock)加锁，以确保线程安全。在完成链表操作后调用spin\_unlock(&lock)解锁。

（5）睡眠控制：每次遍历进程链表后，线程休眠5秒以避免频繁占用系统资源。

**3.线程2**

从链表中读取并打印进程信息，之后删除链表节点，释放内存。

（1）链表遍历：使用list\_for\_each\_entry\_safe(node,tmp,&my\_list,list)遍历链表，node表示当前遍历到的节点，tmp用于遍历时避免节点被删除后的问题。

（2）打印信息：对于每个节点，打印其PID和Command（进程名到内核日志，通过printk完成。

（3）删除节点：遍历过程中，每打印完一个节点的信息后，使用list\_del将该节点从链表中移除，并通过 kfree释放动态分配的内存，防止内存泄漏。

（4）同步机制：由于链表的操作需要保证线程安全，因此同样使用spin\_lock和spin\_unlock进行加锁和解锁，避免与线程1的冲突。

**4.模块的卸载**

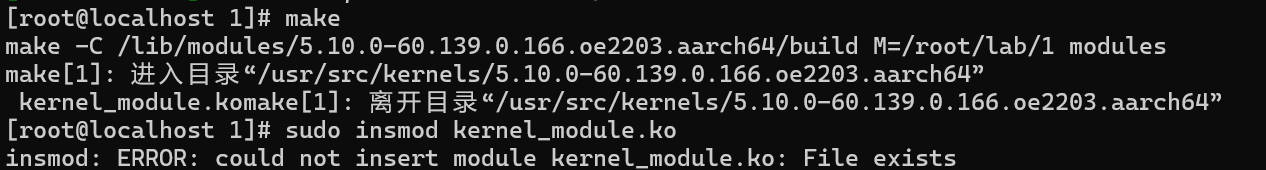
停止线程，清理链表，释放资源。

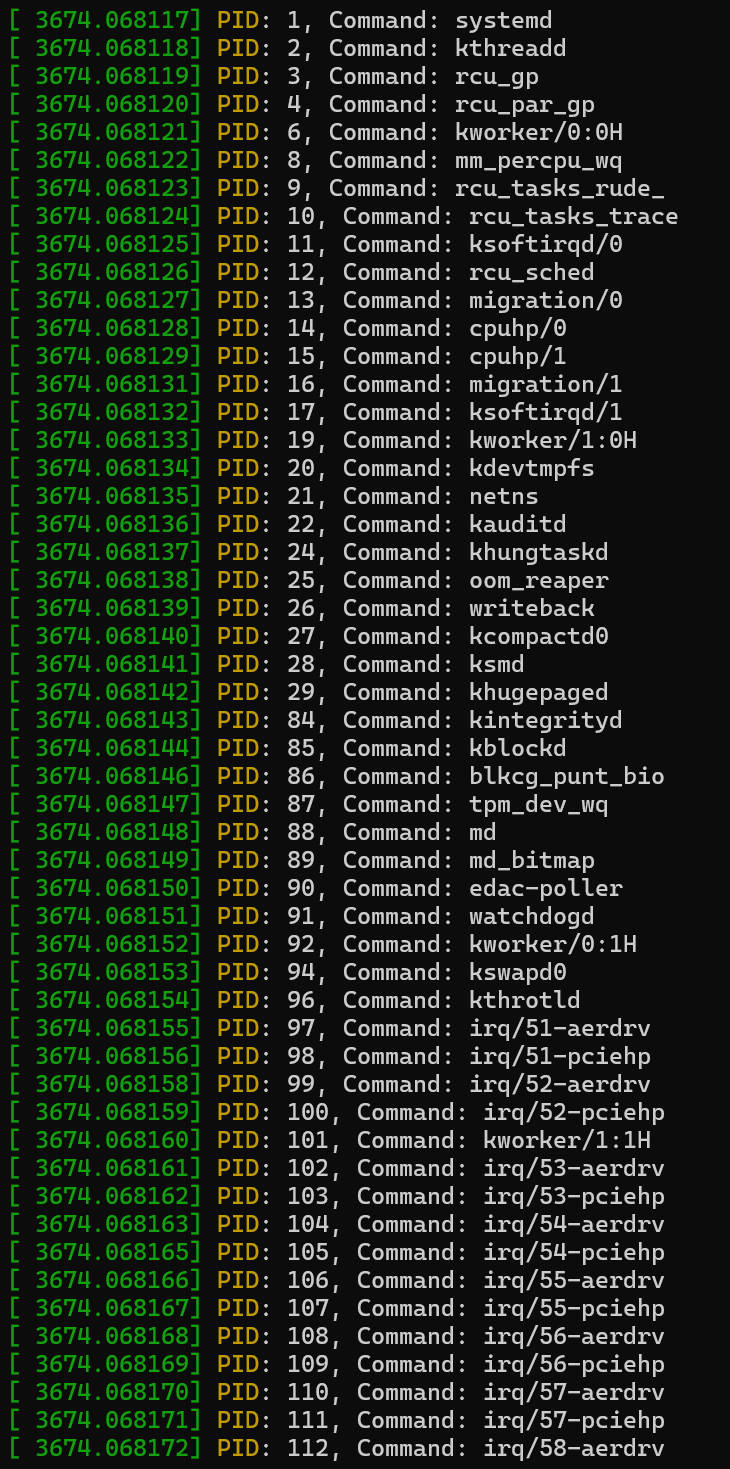
（1）停止线程：调用kthread\_stop停止线程1和线程2，确保线程在模块卸载时不会再运行。这一步会向线程发送一个停止信号，线程函数检测到停止信号后会退出循环并终止。

（2）链表清理：在卸载模块时，遍历链表中的所有节点，使用list\_del将节点从链表中删除，并调用kfree释放每个节点的内存，确保资源正确释放。

（3）同步机制：加锁和解锁仍然用于确保线程安全，防止链表操作在卸载过程中出现竞态条件。

**六、结果分析**

配置并运行过程截图：

结果如图所示：

结果表明内核模块和线程已经成功运行，线程1正确地遍历了系统中的进程链表，并将每个进程的 PID（进程ID）和 Command（进程名）加入到内核链表中，线程2随后遍历了链表并打印了这些信息。

**七、问题回答**

（1）kmalloc,kfree与C语言的malloc,free有何不同？

kmalloc 和 kfree 是 Linux 内核提供的内存分配函数，用于内核态分配和释放内存。它们分配的内存位于内核空间，通常具有较高的分配效率和可预测的内存布局。

而 malloc 和 free 是用户态的标准库函数，用于在用户态程序中分配和释放内存，分配的内存位于用户空间。

使用 kmalloc 时，可以选择 GFP\_KERNEL、GFP\_ATOMIC 等标志，以适应不同的内存分配需求；而 malloc 没有这些选择。

（2）通过何种函数获得进程的ρcb

在 Linux 内核中，可以通过 current 宏直接访问当前进程的 PCB（task\_struct 结构），而对于其他进程，可以通过进程的 PID 来获得 PCB。例如，内核函数 find\_task\_by\_vpid() 可根据进程 ID 返回对应的 task\_struct。

（3）两个内核线程是如何实现同步互斥的

使用了自旋锁 (spinlock\_t) 实现同步互斥​，线程 1 通过自旋锁将进程信息加入链表，线程 2 也通过自旋锁从链表中删除节点并释放内存。这样保证了两个线程不会同时访问或修改链表数据，避免了竞态条件。

（4）内核线程和用户空间线程有何不同

内核线程运行在内核态，可以直接访问内核数据结构，并且拥有更高的优先级和权限。

用户空间线程则运行在用户态，需要通过系统调用与内核交互，无法直接访问内核数据结构。

内核线程的调度可以通过 kthread 系列函数进行控制，例如创建和停止线程；而用户空间线程通常使用 POSIX 标准库创建和控制。

（5）使用内核链表而不是自行编写一个链表有什么好处

内核链表（list\_head）经过内核优化，使用方便且效率高，还提供了多个宏和遍历函数，使链表操作更加简洁、安全和可靠​。

自行编写的链表可能会引入不必要的复杂性和潜在的错误，特别是在内核中实现时错误代价更大。内核链表提供了线程安全和更高效的管理方式，可以避免内存泄漏和其他常见问题。

**八、实验收获**

1.深入理解了内核模块的工作原理

通过编写内核模块，实践了如何在Linux内核中加载、卸载模块，并理解了内核模块与用户空间的区别。了解了内核模块是如何通过insmod加载到内核中，并通过 rmmod 卸载，实践了内核模块生命周期的管理。

2.掌握了内核链表的使用

通过使用内核链表（list\_head 结构）实现了对动态进程数据的存储与管理。与用户态的链表不同，内核链表没有数据域，灵活地支持各种结构体的扩展。在实验中，我掌握了如何初始化链表、添加和删除节点，以及如何安全地遍历链表。

3．学习了内核线程的创建与管理

通过kthread\_create创建了两个内核线程，并使用kthread\_stop停止线程。这使我理解了内核线程与用户线程的不同之处，并掌握了内核线程的启动、唤醒和停止的机制。

4.掌握了多线程同步技术（自旋锁）

使用了自旋锁（spin\_lock）来确保链表在多线程环境下的安全性。自旋锁的特点是不会导致线程休眠，适用于短时间的临界区保护。在实验中，我学会了如何正确地使用自旋锁进行线程同步，并理解了自旋锁与其他锁机制（如互斥锁）的区别。

5.增强了对进程管理的理解

通过for\_each\_process遍历系统的所有进程，掌握了如何从内核空间获取系统中的进程信息，如进程ID（PID）和进程名。这让我对内核中的进程调度和管理有了更深入的理解。

6.熟悉了内核的调试和日志系统

使用printk向内核日志中输出调试信息，并通过dmesg查看内核日志输出，掌握了如何在没有用户态接口的情况下进行调试。这使得我在实验中能够验证模块的行为，快速定位问题。

**八、实验反思**

1.内存管理需谨慎

在内核模块中使用动态内存分配（kmalloc）时，我意识到内存管理的挑战性。与用户态程序不同，内核空间中的内存分配和释放错误会导致整个系统的不稳定甚至崩溃。实验初期忘记释放链表节点会导致内存泄漏问题，需通过遍历链表并调用 kfree 解决了这一问题。

2.线程同步的重要性

多线程操作共享数据结构时，线程安全问题不容忽视。实验中链表是共享资源，线程1在添加节点，线程2在删除节点，必须确保这两个线程不会同时操作链表。