操作系统课程实践实验报告

实验题目：编写内核模块

姓 名：叶福伦

目 录

[一 题目介绍 1](#_Toc515881940)

[二 实验思路 1](#_Toc515881941)

[三 遇到问题及解决方法 1](#_Toc515881942)

[四 核心代码及实验结果展示 1](#_Toc515881943)

[五 个人实验改进与总结 1](#_Toc515881944)

[5.1 个人实验改进 1](#_Toc515881945)

[5.2 个人实验总结 1](#_Toc515881946)

[六 参考文献 1](#_Toc515881947)

# 一 题目介绍

（1）设计一个模块，要求列出系统中所有内核线程的程序名、PID、进程状态、进程优先级、父进程的PID。  
（2）设计一个带参数的模块，其参数为某个进程的PID号，模块的功能是列出该进程的家族信息，包括父进程、兄弟进程和子进程的程序名、PID号及进程状态。

# 二 实验思路

# 实验1

编写动态模块代码



图 1实验1模块代码

编写相关脚本

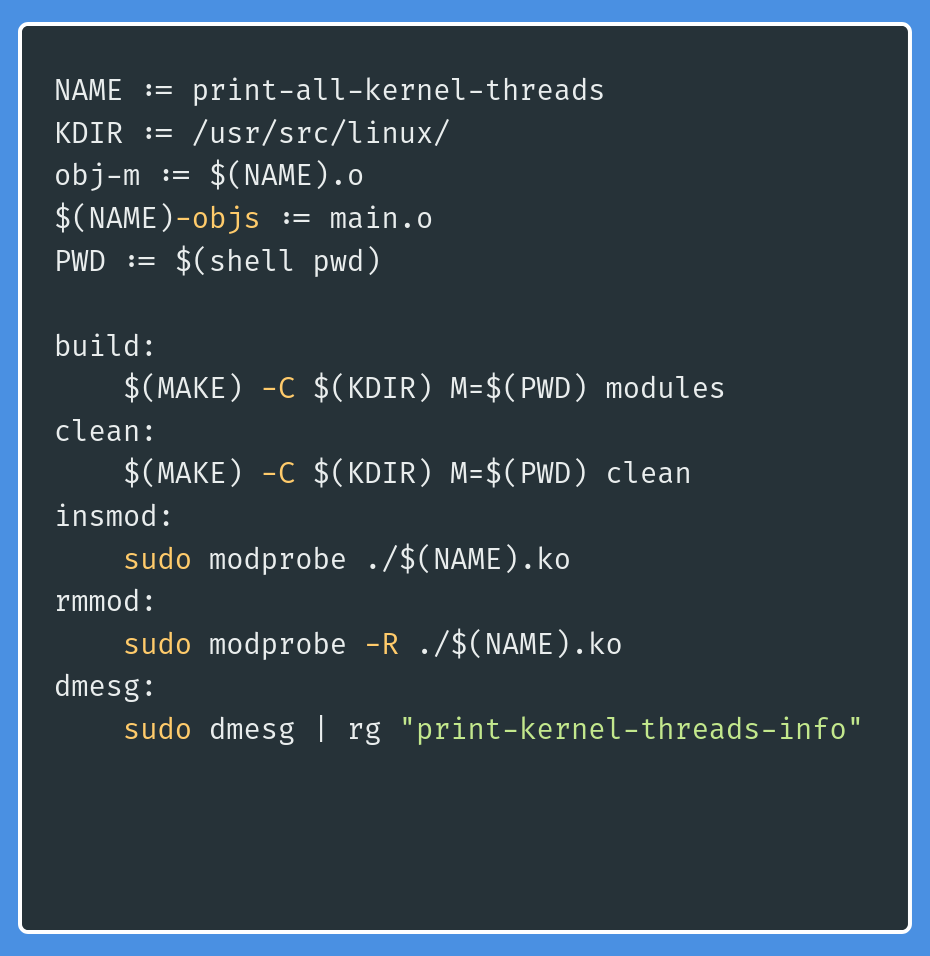


图 2实验2相关脚本

实验2

编写模块代码

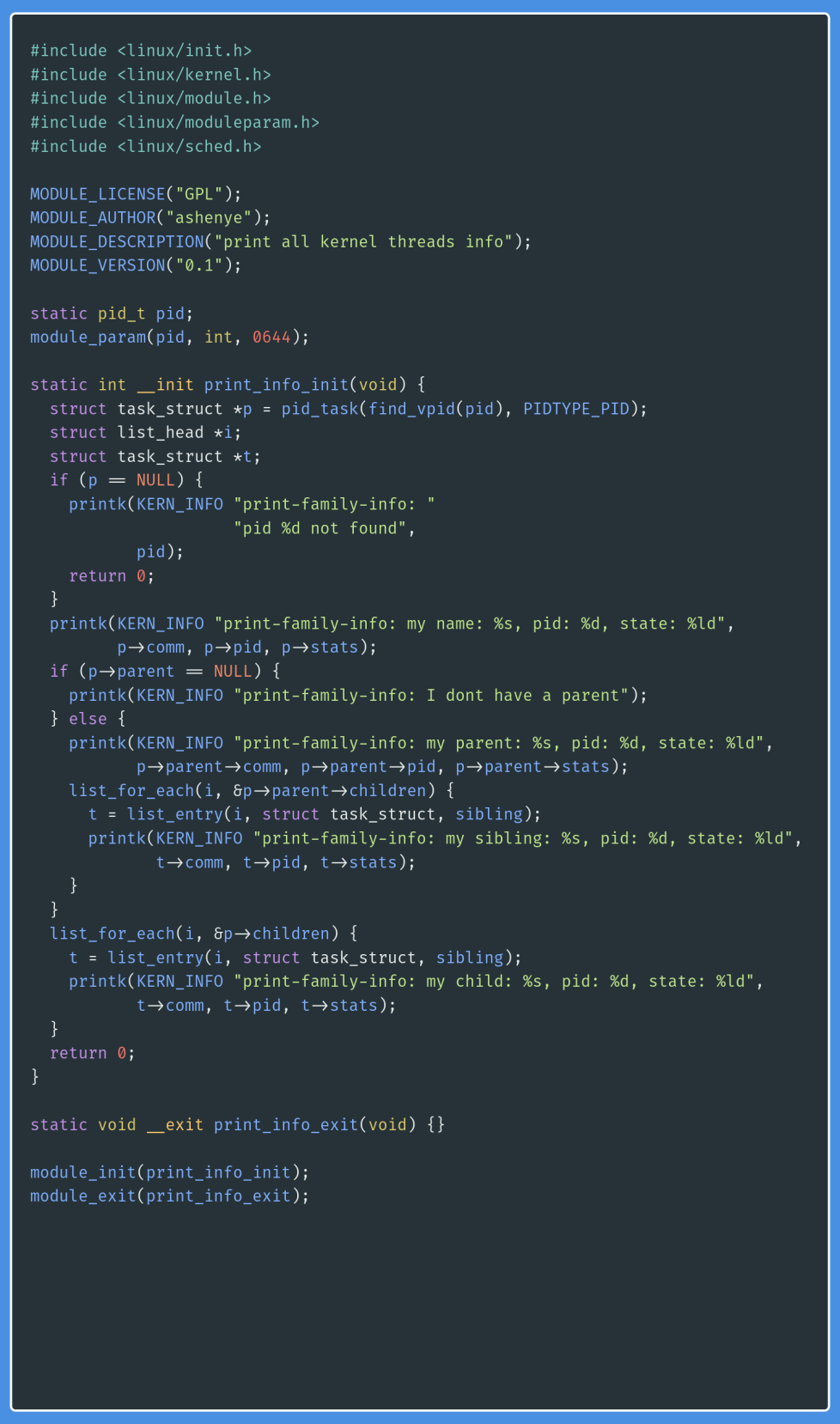


图 3实验2模块代码

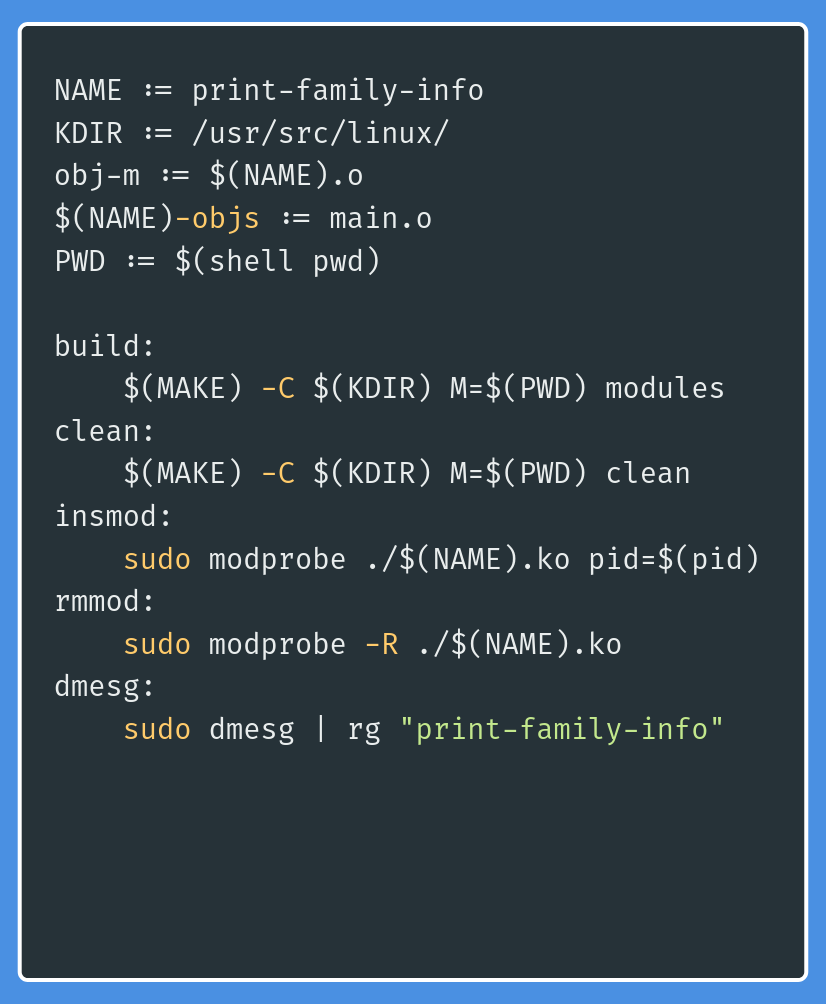


图 4实验2相关脚本

# 三 遇到问题及解决方法

1. clangd报错不能提供正确的提示信息

我用的代码编写环境是vim+clangd, clangd对于涉及操作系统代码的处理不大行。可能是内核的代码量太大了。找了很多教程都不大行。但是有关于操作系统内核或者说模块编程就实验一和二，代码量总体不大。将就着写一下问题也不是特别大。如果需要查找跳转等功能可以用rg, fd等终端工具将就下

1. insmod: ERROR: could not insert module print-all-kernel-threads.ko: Invalid module format

使用maninfo查看.ko文件的元信息，发现版本魔数跟当前内核版本号对不上。linux的动态加载机制看来比较严格. 使用包管理器更新系统头文件

# 四 核心代码及实验结果展示

# carbon(5)

图 5实验1核心代码

相关内核代码

|  |  |  |  |
| --- | --- | --- | --- |
| 符号 | 类型 | 位置 | 说明 |
| init\_task | 全局变量 | include/linux/sched/task.h | 0号进程的pcb |
| task\_struct.tasks | 成员字段 | include/linux/sched.h | 嵌入式设计的链表节点把所有进程的pcb串在一起 |
| list\_head | 结构体 | include/linux/types.h | 嵌入式的链表节点 |
| for\_each\_process | 宏 | include/linux/sched/signal.h | 通过task\_struct.tasks字段实现进程的遍历 |
| next\_task | 宏 | include/linux/sched/signal.h | 得到链表的后继节点 |
| list\_entry\_rcu | 宏 | include/linux/rculist.h | container\_of的包装器，使用rcu机制，保证并发安全 |
| container\_of | 宏 | include/linux/container\_of.h | 传入指向成员字段的指针、结构体、字段名得到结构体头部的指针 |

task\_struct.tasks:

struct task\_struct {

/\*...\*/

struct list\_head tasks;

/\*...\*/

}

list\_head:

struct list\_head {

struct list\_head \*next, \*prev;

};

for\_each\_process:

#define for\_each\_process(p) \

for (p = &init\_task ; (p = next\_task(p)) != &init\_task ; )

next\_task:

#define next\_task(p) \

list\_entry\_rcu((p)->tasks.next, struct task\_struct, tasks)

list\_entry\_rcu:

#define list\_entry\_rcu(ptr, type, member) \

container\_of(READ\_ONCE(ptr), type, member)

container\_of:

#define container\_of(ptr, type, member) ({ \

void \*\_\_mptr = (void \*)(ptr); \

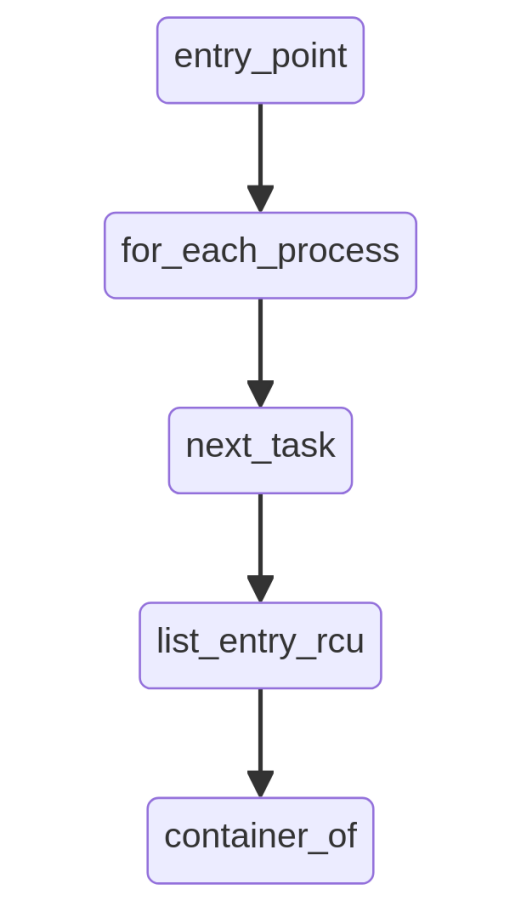
static\_assert(\_\_same\_type(\*(ptr), ((type \*)0)->member) || \

\_\_same\_type(\*(ptr), void), \

"pointer type mismatch in container\_of()"); \

((type \*)(\_\_mptr - offsetof(type, member))); })

宏调用图



首先p指向init\_task，init\_task就是0号进程的pcb，task\_struct结构体的tasks是一个嵌入式的链表结构，然后使用for\_each\_process迭代宏，for\_each\_process内部使用next\_task来实现迭代，next\_task调用list\_entry\_rcu宏，list\_entry\_rcu是container\_of的并发安全的包装器， container\_of根据成员指针和成员相对于头部的偏移量从而计算出结构体头部的指针，这样就能遍历所有进程了。内核线程没有自己的地址空间我们可以根据这个特点找出所有的内核线程，然后把信息打印出来

结果展示

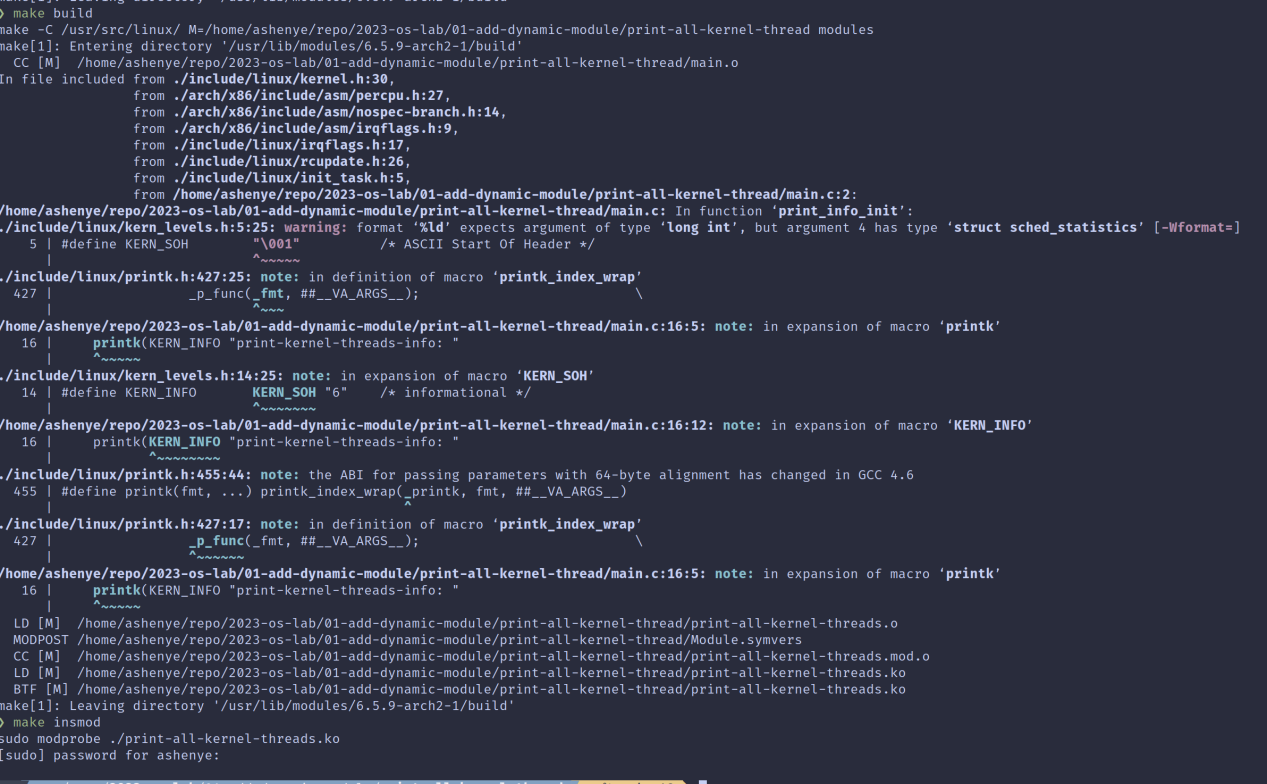


图 6编译内核模块

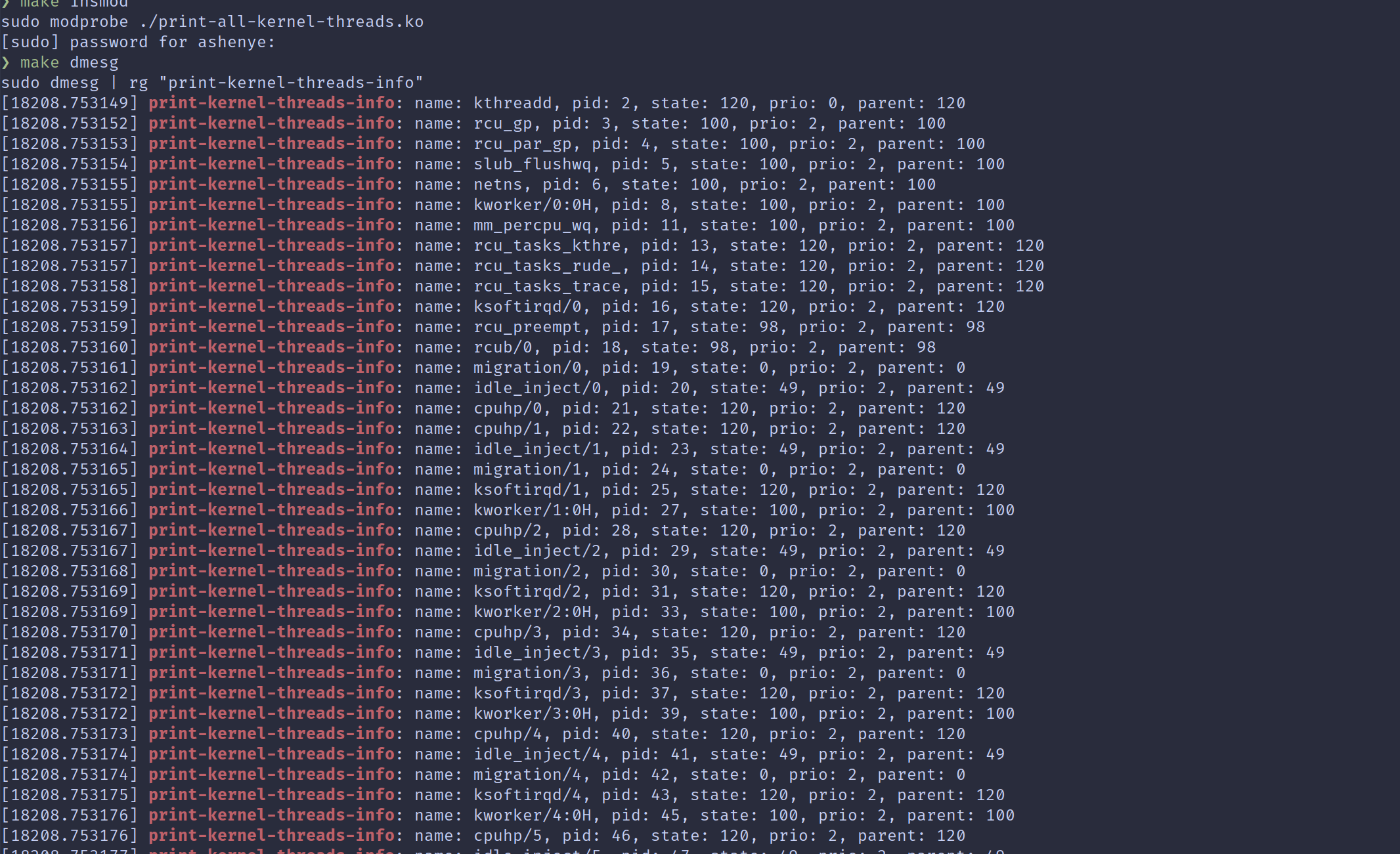


图 7验证结果

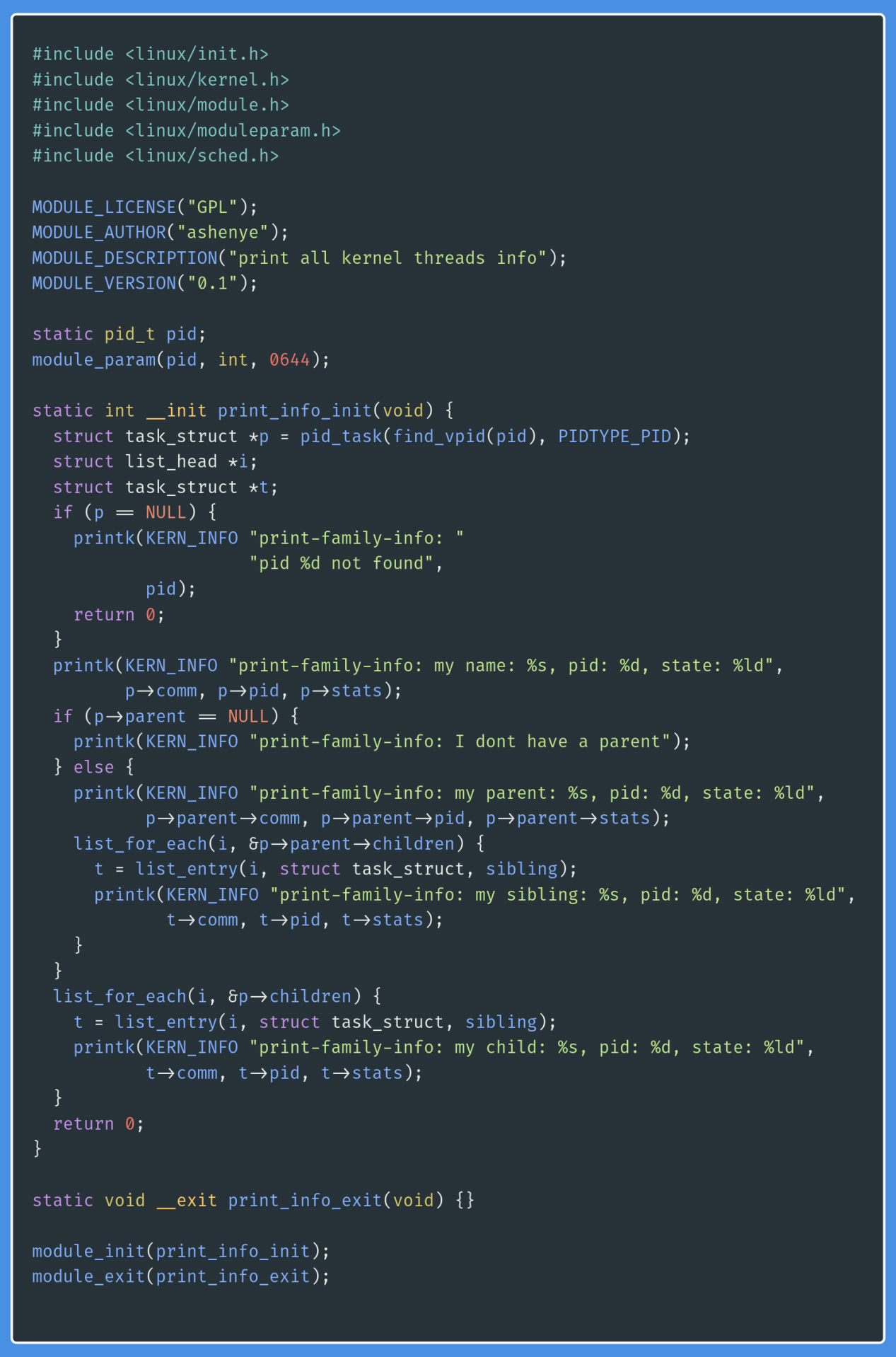


图 8实验2核心代码

相关内核代码

|  |  |  |  |
| --- | --- | --- | --- |
| 符号 | 类型 | 位置 | 说明 |
| pid\_task | 函数 | 原型：include/linux/pid.h  定义：kernel/pid.c | 获取任务的任务描述符信息，此任务在进程pid的使用链表中，并且搜索的链表的起始元素的下标为参数type的值 |
| pid | 结构体 | include/linux/pid.h | 为了描述PID信息 |
| pid\_type | 枚举 | include/linux/pid.h | 描述进程ID的类型 |
| find\_vpid | 函数 | 原型：include/linux/pid.h  定义：kernel/pid.c | 此函数根据提供的局部进程号获取对应的进程描述符 |
| find\_pid\_ns | 函数 | 原型：include/linux/pid.h  定义：kernel/pid.c | 在某个命名空间下查询nr对应的进程描述符 |
| idr\_find | 函数 | 原型：include/linux/idr.h  定义：lib/idr.c | 涉及idr整数id管理机制，通过id来找到对应id的对象 |

pid\_task

struct task\_struct \*pid\_task(struct pid \*pid, enum pid\_type type)

{

struct task\_struct \*result = NULL;

if (pid) {

struct hlist\_node \*first;

first = rcu\_dereference\_check(hlist\_first\_rcu(&pid->tasks[type]),

lockdep\_tasklist\_lock\_is\_held());

if (first)

result = hlist\_entry(first, struct task\_struct, pid\_links[(type)]);

}

return result;

}

pid:

struct pid

{

refcount\_t count;

unsigned int level;

spinlock\_t lock;

/\* lists of tasks that use this pid \*/

struct hlist\_head tasks[PIDTYPE\_MAX];

struct hlist\_head inodes;

/\* wait queue for pidfd notifications \*/

wait\_queue\_head\_t wait\_pidfd;

struct rcu\_head rcu;

struct upid numbers[];

};

find\_vpid:

struct pid \*find\_vpid(int nr)

{

return find\_pid\_ns(nr, task\_active\_pid\_ns(current));

}

find\_pid\_ns:

struct pid \*find\_pid\_ns(int nr, struct pid\_namespace \*ns)

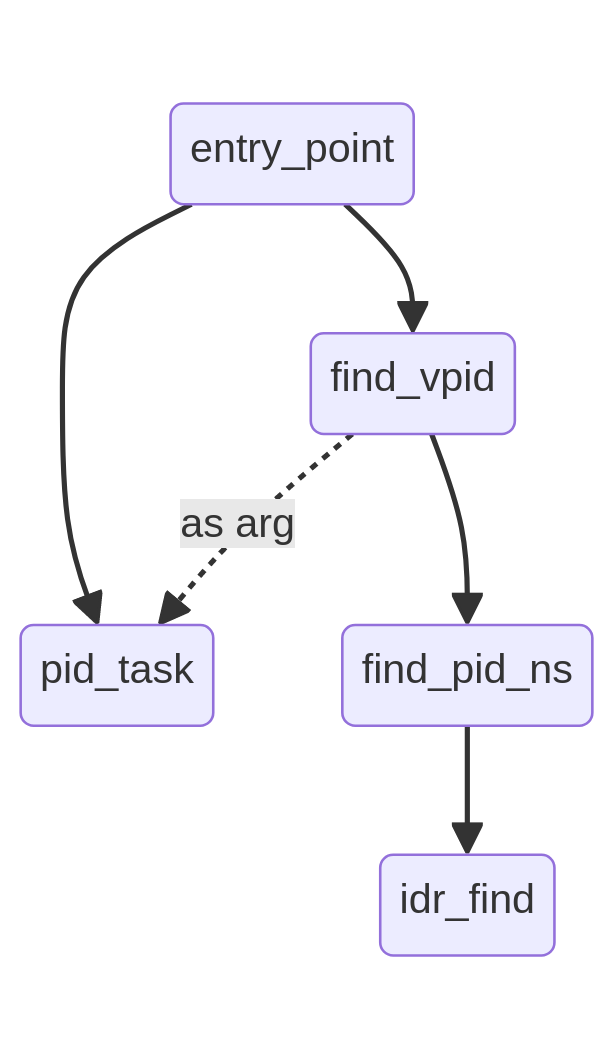
{

return idr\_find(&ns->idr, nr);

}

主要讲解如何从id找到对象的过程，遍历兄弟或者是遍历孩子的内核代码思路和上一个实验类似

主要的函数调用过程



模块的入口函数调用find\_vpid来通过pid来拿到pid描述符，find\_vpid实际调用了find\_pid\_ns 在当前的命名空间下查找该进程， find\_pid\_ns把当前的命名空间和id传入idr\_find从而找到pid描述符。idr\_find涉及内核对于整形id的管理分配机制不多赘述。入口函数执行完find\_vpid后拿到了pid描述符，再把pid描述符传入pid\_task并指定查找类型为PIDTYPE\_PID的pcb。函数内部在进程pid的使用链表中，并且搜索的链表的起始元素的下标为参数type的值。

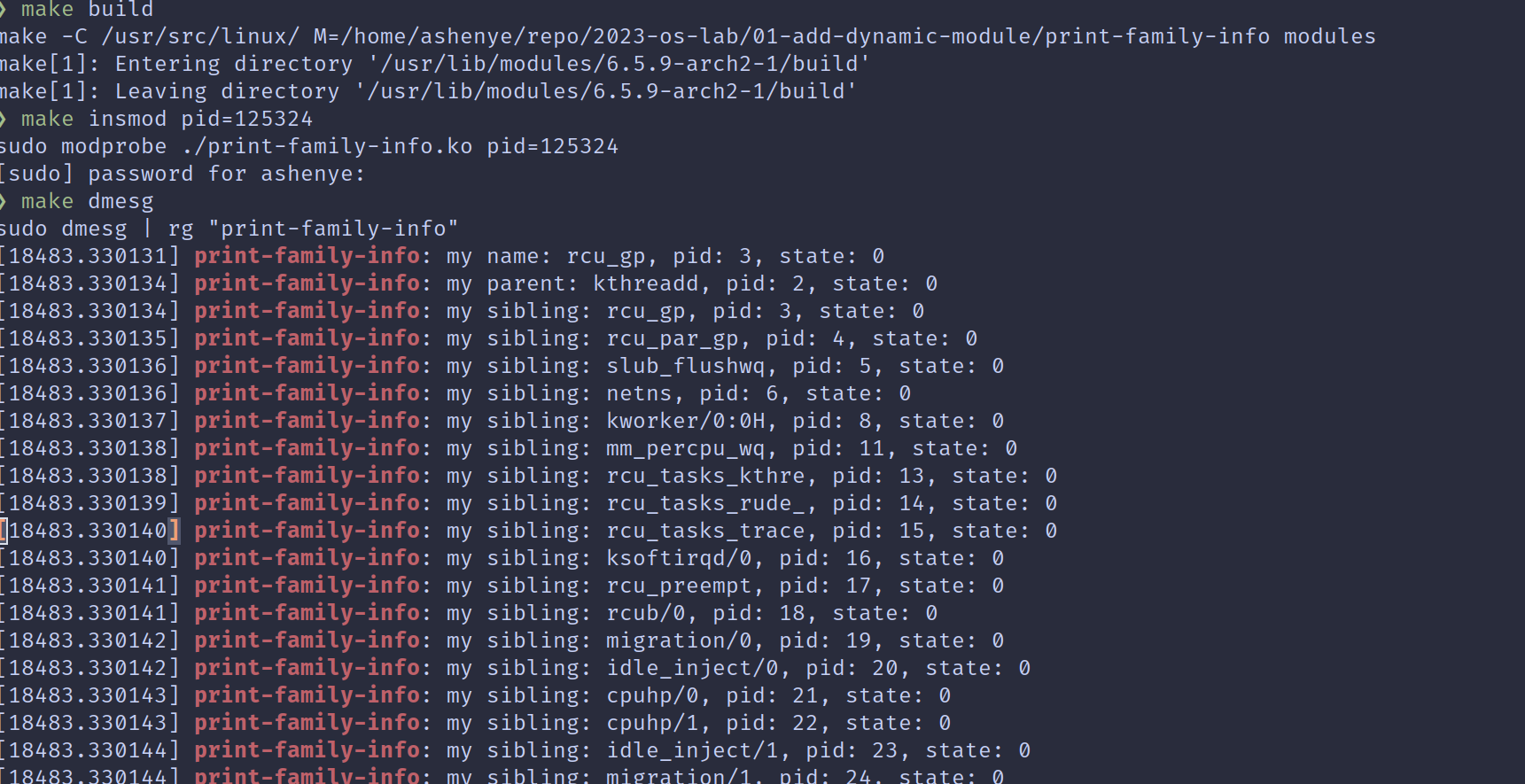


图 9编译运行验证

# 五 个人实验改进与总结

## 5.1 个人实验改进

## 对实验进行了模块化，分为了两个部分，一个用于打印所有内核线程信息，另一个用于打印特定进程的家族信息。这种结构本身是一项改进，因为它使代码更有组织性，并且更容易理解和维护。

## 以下是我的代码改进点：

## 1. 模块化设计：将代码分成两个模块（`print-all-kernel-thread`和`print-family-info`）是一项有益的改进。这种模块化设计使每个模块专注于不同的功能，提高了代码的可重用性和可维护性。这使得未来可以更容易地扩展或修改每个模块，而不会对整个代码产生太大的影响。

## 2. Makefile：每个模块都有自己的Makefile，这有助于编译和构建代码，保持代码的独立性。这种做法使得代码更易于构建，减少了潜在的构建错误。

## 3. 清晰的文件结构：代码树结构相对清晰，`doc`文件夹用于文档，每个模块都有自己的文件夹，这有助于组织和查找代码文件。

## 4. 可读性和注释：每个模块的`main.c`文件中有适当的注释和文档，以解释模块的功能、输入、输出等信息，那么代码的可读性将大大提高。

## 5.2 个人实验总结

在本学期修习操作系统实验课期间，我学到了如何编写内核模块，具体地说，我完成了两个任务：设计一个模块用于列出系统中所有内核线程的信息，以及设计一个带参数的模块用于列出特定进程的家族信息。这些任务让我深入了解了内核模块的开发以及操作系统内部的工作原理，同时也带来了一些重要的体会。

首先，编写内核模块是一项挑战性的任务，要求对Linux内核的机制有深入的理解。在完成这两个任务时，我学到了如何访问内核数据结构、如何与进程控制块（PCB）交互，以及如何处理进程的各种属性。这为我提供了更深入的操作系统知识，并增强了我对操作系统内部机制的理解。

其次，任务一中的模块让我学会了如何遍历系统中所有的内核线程，提取它们的信息，并将其呈现出来。这需要对内核数据结构和线程管理的深刻理解。这个任务使我认识到了内核线程在系统中的重要性，因为它们通常执行关键任务，如调度、中断处理等。

任务二则要求我设计一个带参数的内核模块，根据传入的PID号来列出该进程的家族信息。这扩展了我的内核模块开发技能，因为我需要处理模块参数、查找家族成员并显示它们的信息。这个任务还加强了我对进程管理和进程之间关系的理解。

这两个任务的完成也教育了我代码的健壮性和可维护性的重要性。内核模块可能对系统稳定性产生重大影响，因此必须确保代码质量高，避免引入潜在的漏洞。

总之，这两个任务让我不仅获得了内核模块开发的实际经验，还深入了解了操作系统的内部机制和进程管理。这些体会将在我未来的学术和职业生涯中发挥关键作用，为我在操作系统领域取得更多成就提供了坚实的基础。我期待继续学习和探索操作系统的深奥之处。

# 六 参考文献

https://blog.csdn.net/weixin\_43614380/article/details/111053959