**题目3：新增Linux驱动程序**

1. **设计目的与要求**

本设计旨在通过在Linux内核中添加新的系驱动程序，深入理解操作系统内核的工作机制和驱动程序的实现方法。此外，通过实际编程练习，增强对Linux内核编程和环境配置的掌握。

1. **设计内容**

增加一个驱动程序（使用内存模拟设备），使用模块编译方式。 要求：

（1）可以动态加载和卸载新的驱动。

（2）通过程序或命令行使用该驱动。

（3）至少能通过该驱动保存256MB的数据，还能将这些数据读取出来。

（4）要重新编译Linux内核，可模仿ramdisk的实现方式。

1. **设备与环境**

操作系统：Ubuntu20.04

虚拟化软件：VMware Workstation 17 Pro

开发环境：Visual Studio Code

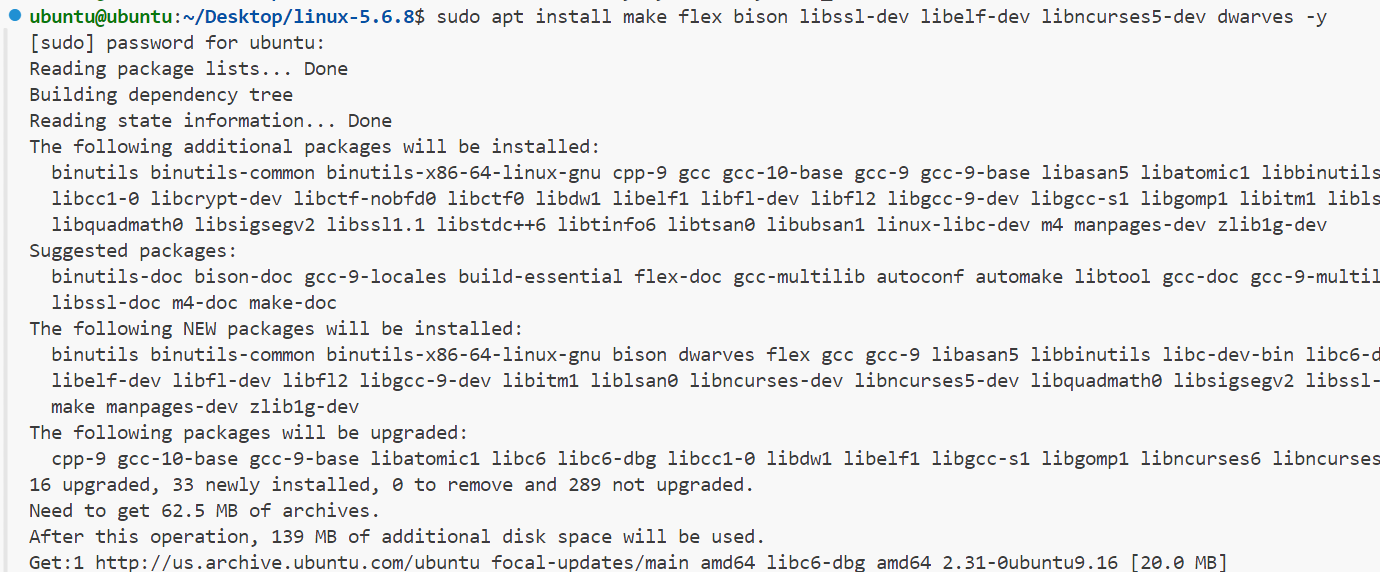
内核版本：5.6.8

1. **设计思想**

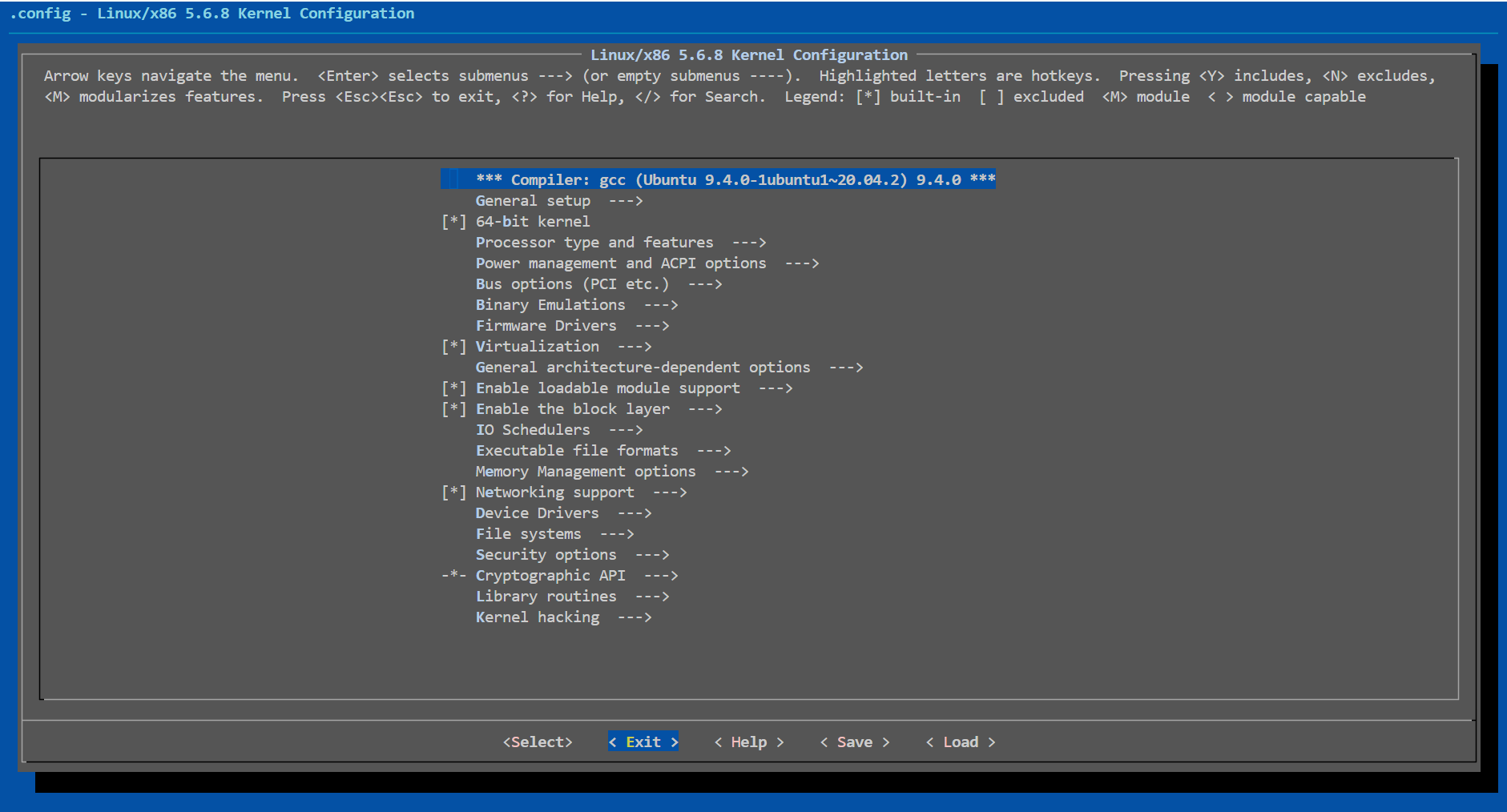
通过内存模拟块设备，设计一个内核模块实现类似 ramdisk 的功能。驱动程序通过内核中的 blk\_mq（多队列块设备接口）实现数据的读写请求处理，分配 256MB 的连续虚拟内存作为模拟设备的存储空间。模块编译完成后，通过动态加载和卸载（insmod 和 rmmod）方式管理驱动。数据通过标准块设备接口（如 /dev/ramdisk）访问，可以使用 dd 或自定义程序读写该设备。最后，通过修改和重新编译 Linux 内核集成驱动，保证驱动与系统兼容性。

1. **主要数据结构和流程**

1.下载内核源码并解压，安装编译内核所需的依赖包



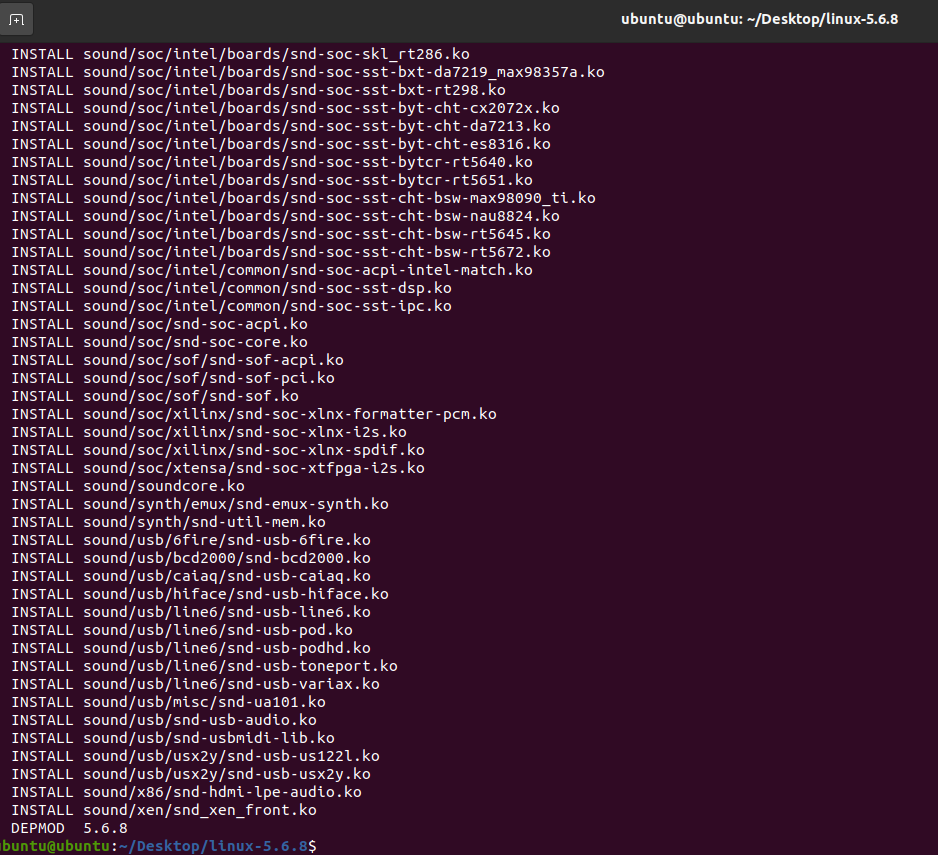
2. 配置内核，输入**make menuconfig**，在跳出的界面中进行配置



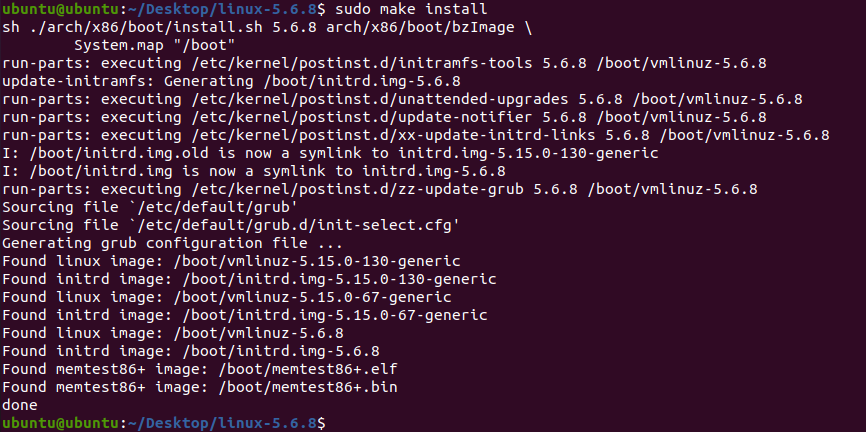
3.编译，查看自己机器处理器的核数，中键入 **make -jn**（n个线程开始编译）



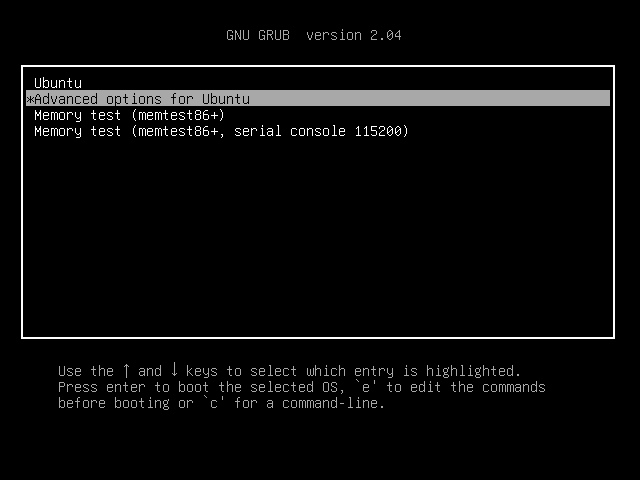
4安装编译好的模块，输入sudo make modules\_install 命令，完成模块的安装操作

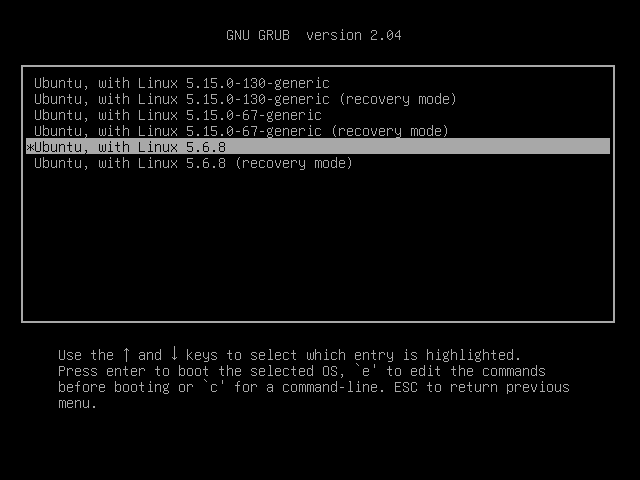


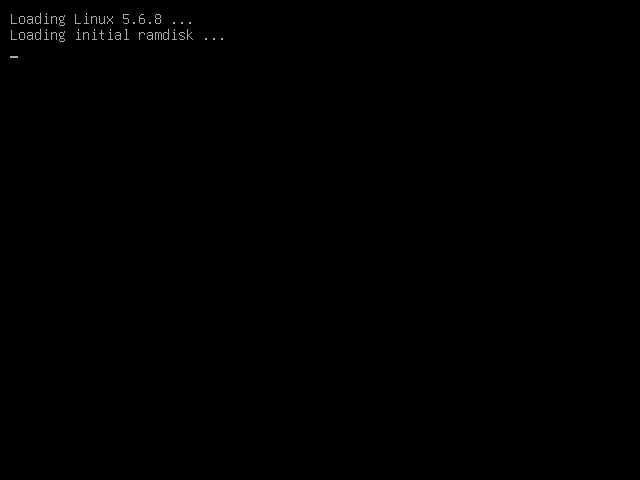
5. 输入 sudo make install安装内核

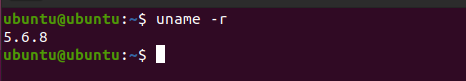


6.重启，内核切换成功









1. **实验测试结果及结果分析**

1、编写ram\_blkdev模块

#include <linux/module.h>

#include <linux/blkdev.h>

#include <linux/init.h>

#include <linux/fs.h>

#include <linux/vmalloc.h>

#include <linux/kernel.h>

#include <linux/string.h>

#include <linux/blk-mq.h>

#define SIMP\_BLKDEV\_DISKNAME "ram\_blkdev"

#define SIMP\_BLKDEV\_BYTES (256 \* 1024 \* 1024) // 256 MB

#define SECTOR\_SHIFT 9

#define SECTOR\_SIZE (1 << SECTOR\_SHIFT)

// 内存虚拟磁盘空间

static unsigned char \*zombotany\_blkdev\_data = NULL;

// 主设备号

static int simp\_blkdev\_major = 0;

// 设备 gendisk 结构

static struct gendisk \*zombotany\_blkdev\_disk = NULL;

// 请求队列相关结构

static struct blk\_mq\_tag\_set zombotany\_blkdev\_tag\_set;

static struct request\_queue \*zombotany\_blkdev\_queue = NULL;

// 请求处理函数

static blk\_status\_t zombotany\_handle\_request(struct blk\_mq\_hw\_ctx \*hctx, const struct blk\_mq\_queue\_data \*bd)

{

    struct request \*req = bd->rq;

    struct bio\_vec bvec;

    struct req\_iterator iter;

    unsigned long start\_sector = blk\_rq\_pos(req); // 起始扇区

    unsigned char \*disk\_mem = zombotany\_blkdev\_data + (start\_sector << SECTOR\_SHIFT); // 起始位置

    blk\_status\_t status = BLK\_STS\_OK;

    // 遍历请求中的 BIO

    rq\_for\_each\_segment(bvec, req, iter) {

        void \*buffer = kmap\_atomic(bvec.bv\_page) + bvec.bv\_offset;

        if (rq\_data\_dir(req) == READ) {

            // 执行读取操作

            memcpy(buffer, disk\_mem, bvec.bv\_len);

        } else if (rq\_data\_dir(req) == WRITE) {

            // 执行写入操作

            memcpy(disk\_mem, buffer, bvec.bv\_len);

        } else {

            status = BLK\_STS\_IOERR;

            break;

        }

        kunmap\_atomic(buffer);

        disk\_mem += bvec.bv\_len;

    }

    blk\_mq\_end\_request(req, status);

    return status;

}

// 多队列块设备操作

static struct blk\_mq\_ops zombotany\_blkdev\_ops = {

    .queue\_rq = zombotany\_handle\_request,

};

// 块设备操作结构体

static struct block\_device\_operations zombotany\_blkdev\_fops = {

    .owner = THIS\_MODULE,

};

// 模块初始化函数

static int \_\_init zombotany\_blkdev\_init(void)

{

    int ret;

    // 分配虚拟磁盘存储空间

    zombotany\_blkdev\_data = vzalloc(SIMP\_BLKDEV\_BYTES);

    if (!zombotany\_blkdev\_data) {

        printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME ": vzalloc failed\n");

        return -ENOMEM;

    }

    // 动态申请主设备号

    simp\_blkdev\_major = register\_blkdev(0, SIMP\_BLKDEV\_DISKNAME);

    if (simp\_blkdev\_major < 0) {

        printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME ": unable to get major number\n");

        vfree(zombotany\_blkdev\_data);

        return simp\_blkdev\_major;

    }

    // 初始化多队列设置

    memset(&zombotany\_blkdev\_tag\_set, 0, sizeof(zombotany\_blkdev\_tag\_set));

    zombotany\_blkdev\_tag\_set.ops = &zombotany\_blkdev\_ops;

    zombotany\_blkdev\_tag\_set.nr\_hw\_queues = 1;

    zombotany\_blkdev\_tag\_set.queue\_depth = 128;

    zombotany\_blkdev\_tag\_set.numa\_node = NUMA\_NO\_NODE;

    zombotany\_blkdev\_tag\_set.cmd\_size = 0;

    zombotany\_blkdev\_tag\_set.flags = BLK\_MQ\_F\_SHOULD\_MERGE;

    ret = blk\_mq\_alloc\_tag\_set(&zombotany\_blkdev\_tag\_set);

    if (ret) {

        printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME ": blk\_mq\_alloc\_tag\_set failed\n");

        unregister\_blkdev(simp\_blkdev\_major, SIMP\_BLKDEV\_DISKNAME);

        vfree(zombotany\_blkdev\_data);

        return ret;

    }

    // 初始化请求队列

    zombotany\_blkdev\_queue = blk\_mq\_init\_queue(&zombotany\_blkdev\_tag\_set);

    if (IS\_ERR(zombotany\_blkdev\_queue)) {

        printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME ": blk\_mq\_init\_queue failed\n");

        blk\_mq\_free\_tag\_set(&zombotany\_blkdev\_tag\_set);

        unregister\_blkdev(simp\_blkdev\_major, SIMP\_BLKDEV\_DISKNAME);

        vfree(zombotany\_blkdev\_data);

        return PTR\_ERR(zombotany\_blkdev\_queue);

    }

    // 分配 gendisk 结构

    zombotany\_blkdev\_disk = alloc\_disk(1);

    if (!zombotany\_blkdev\_disk) {

        printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME ": alloc\_disk failed\n");

        blk\_cleanup\_queue(zombotany\_blkdev\_queue);

        blk\_mq\_free\_tag\_set(&zombotany\_blkdev\_tag\_set);

        unregister\_blkdev(simp\_blkdev\_major, SIMP\_BLKDEV\_DISKNAME);

        vfree(zombotany\_blkdev\_data);

        return -ENOMEM;

    }

    // 设置 gendisk 属性

    zombotany\_blkdev\_disk->major = simp\_blkdev\_major;

    zombotany\_blkdev\_disk->first\_minor = 0;

    zombotany\_blkdev\_disk->fops = &zombotany\_blkdev\_fops;

    zombotany\_blkdev\_disk->queue = zombotany\_blkdev\_queue;

    snprintf(zombotany\_blkdev\_disk->disk\_name, 32, SIMP\_BLKDEV\_DISKNAME);

    set\_capacity(zombotany\_blkdev\_disk, SIMP\_BLKDEV\_BYTES >> SECTOR\_SHIFT);

    // 注册磁盘

    add\_disk(zombotany\_blkdev\_disk);

    printk(KERN\_INFO SIMP\_BLKDEV\_DISKNAME ": module loaded\n");

    return 0;

}

// 模块卸载函数

static void \_\_exit zombotany\_blkdev\_exit(void)

{

    if (zombotany\_blkdev\_disk) {

        del\_gendisk(zombotany\_blkdev\_disk);

        put\_disk(zombotany\_blkdev\_disk);

    }

    if (zombotany\_blkdev\_queue) {

        blk\_cleanup\_queue(zombotany\_blkdev\_queue);

    }

    blk\_mq\_free\_tag\_set(&zombotany\_blkdev\_tag\_set);

    if (zombotany\_blkdev\_data) {

        vfree(zombotany\_blkdev\_data);

    }

    unregister\_blkdev(simp\_blkdev\_major, SIMP\_BLKDEV\_DISKNAME);

    printk(KERN\_INFO SIMP\_BLKDEV\_DISKNAME ": module unloaded\n");

}

module\_init(zombotany\_blkdev\_init);

module\_exit(zombotany\_blkdev\_exit);

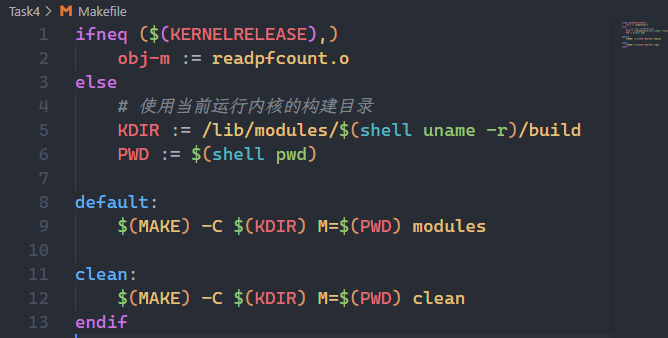
MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Your Name");

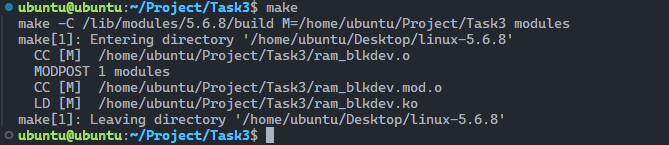
MODULE\_DESCRIPTION("A simple Linux block device module using blk-mq");

注意：Linux 5.6.8版本不再支持旧的接口，比如 blk\_fetch\_request 和 blk\_end\_request\_all。这些接口已经在较新的内核版本中被移除或替换，甚至 blk\_init\_queue 也被替代为多队列接口。

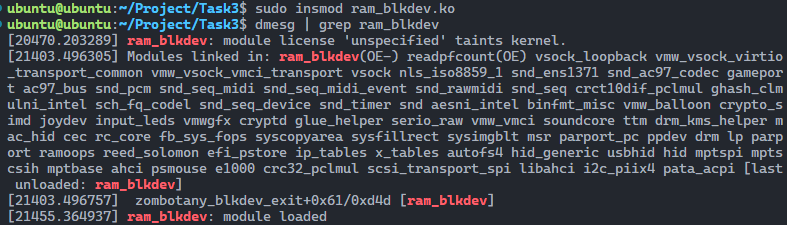
2、Makefile

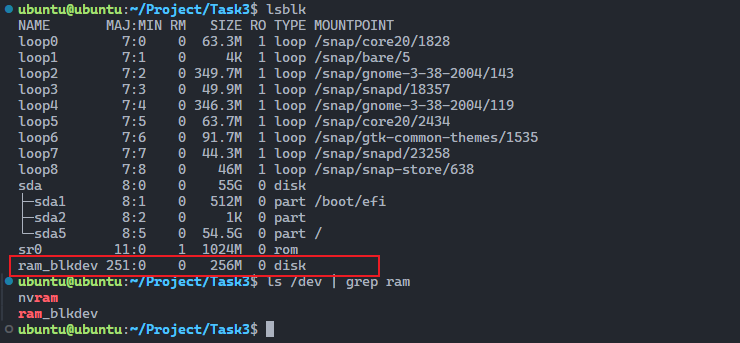


1. Make

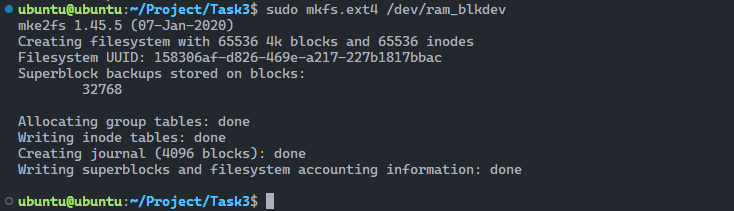


1. 安装模块

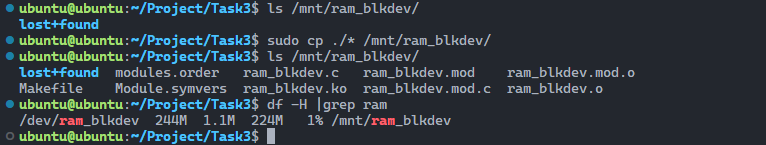




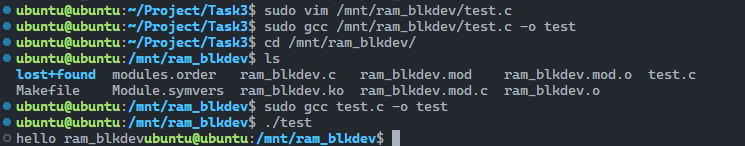
可以看到驱动已经被load过了，且大小为256M，接下来进行格式化和目录挂载，如下图所示



1. 使用bash命令进行文件操作



可以看到ram\_blkdev可以正确存储、加载、修改文件



1. 最后卸载模块，首先取消挂载，可以看到调用该设备的用户数量为0

