# 基于libfuse的用户态文件系统

## 项目背景

通过在Linux环境基于Libfuse使用C语言实现用户态文件系统。分别实现文件系统的文件读取、文件写入、文件访问、文件重命名、权限修改等常见文件和目录操作，所有操作使用fuse\_operations数据结构和相应函数进行操作函数进行映射。

## 项目运行环境

操作系统:

Linux ubuntu 5.4.0-96-generic #109-Ubuntu SMP Wed Jan 12 16:49:16 UTC 2022 x86\_64 x86\_64 x86\_64 GNU/Linux

编译器:

gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.1)

IDE:

VIM - Vi IMproved 8.1 (2018 May 18, compiled Sep 19 2022 04:59:57)

Vscode

Libfuse版本:

libfuse2 (= 2.9.9-3), libselinux-dev

## 项目实现和思路

FUSE（用户空间中的文件系统）是用户空间程序将文件系统导出到 Linux 内核的接口。 FUSE 项目由两个组件组成：fuse 内核模块（在常规内核存储库中维护）和 libfuse 用户空间库（在该存储库中维护）。 libfuse 提供了与 FUSE 内核模块通信的参考实现。

FUSE 文件系统通常实现为与 libfuse 链接的独立应用程序。 libfuse 提供了挂载文件系统、卸载文件系统、从内核读取请求以及发回响应的函数。 libfuse 提供了两个 API：“高级”同步 API 和“低级”异步 API。在这两种情况下，来自内核的传入请求都使用回调传递给主程序。使用高级 API 时，回调可能会使用文件名和路径而不是 inode，并且当回调函数返回时请求处理完成。使用低级 API 时，回调必须与 inode 一起使用，并且必须使用一组单独的 API 函数显式发送响应。

功过编写Main函数和所需的头文件，然后填写libfuse的重要结构体，来指明其用户相应操作和对应函数执行的关系。

static struct fuse\_operations memfs\_oper = {

.getattr = memfs\_getattr,

.readlink = memfs\_readlink,

.readdir = memfs\_readdir,

.mknod = memfs\_mknod,

.mkdir = memfs\_mkdir,

.symlink = memfs\_symlink,

.unlink = memfs\_unlink,

.rmdir = memfs\_rmdir,

.rename = memfs\_rename,

.link = memfs\_link,

.chmod = memfs\_chmod,

.chown = memfs\_chown,

.truncate = memfs\_truncate,

.utimens = memfs\_utimens,

.open = memfs\_open,

.read = memfs\_read,

.write = memfs\_write,

.release = memfs\_release

};

Memfs\_oper数据结构的成员如下,

Getattr,获取文件属性

readlink读取文件链接

readdir读取目录

mknode创建节点

mkdir创建目录

symlink创建软连接

unlink删除文件

rmdir删除目录

rename文件重命名

link硬链接

chmod修改文件权限

truncate截断文件

open打开文件

read读取文件

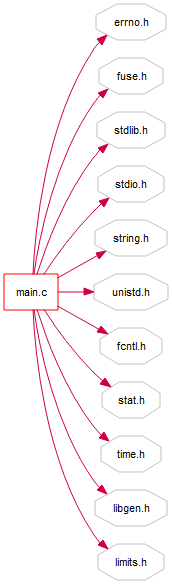
write写入文件

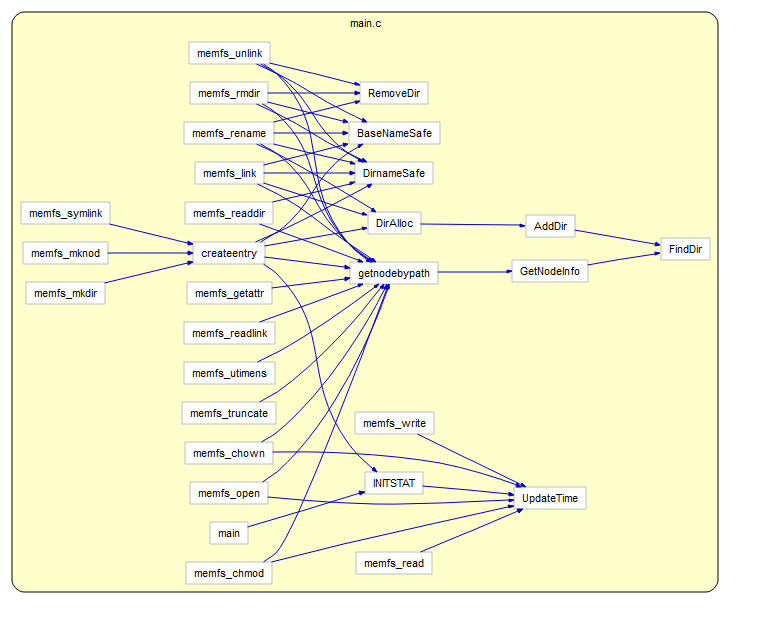
release释放内存

分别实现上述所有数据结构的成员相对的操作函数。

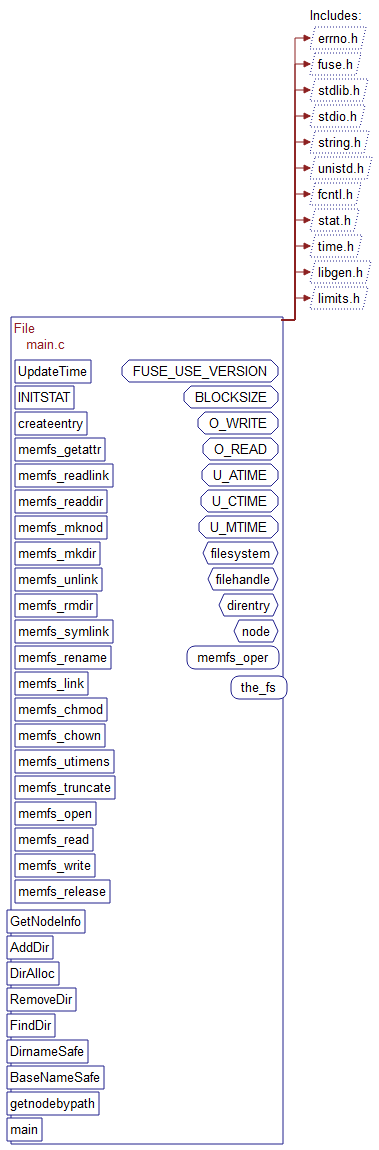
## 项目模块分析

### 项目依赖情况如下，





### 项目声明如下，



### 项目重要头文件如下，

// Linux的头文件，提供错误处理

#include <errno.h>

// Libfuse的头文件，提供相应的数据结构和函数

#include <fuse.h>

// 标准库函数

#include <stdlib.h>

// 标准输入输出函数,提供scanf、printf基本函数

#include <stdio.h>

// 字符串处理

#include <string.h>

// 系统函数

#include <unistd.h>

// 文件操作函数

#include <fcntl.h>

// 文件属性函数

#include <sys/stat.h>

// 获取系统事件

#include <sys/time.h>

// 获取事件

#include <libgen.h>

#include <errno.h>

#include <stdlib.h>

#include <string.h>

#include <sys/stat.h>

// 限制

#include <linux/limits.h>

### 项目重要对象如下，

#define BLOCKSIZE 4096

// 写属性

#define O\_WRITE(flags) ((flags) & (O\_RDWR | O\_WRONLY))

// 读属性

#define O\_READ(flags) (((flags) & (O\_RDWR | O\_RDONLY)) | !O\_WRITE(flags))

// 文件访问时间

#define U\_ATIME (1 << 0)

// 文件创建时间

#define U\_CTIME (1 << 1)

// 文件修改时间

#define U\_MTIME (1 << 2)

### 项目主要数据结构如下，

// 文件系统数据结构

struct filesystem {

// 节点

struct node \*root;

};

// 文件操作

struct filehandle {

struct node \*node;

int o\_flags;

};

struct filesystem the\_fs;

// 目录数据结构

struct direntry {

// 目录名称

char name[PATH\_MAX];

struct node \*node;

// 下一个入口

struct direntry \*next;

};

// 节点数据结构

struct node {

// 节点属性

struct stat vstat;

// 节点数据

void \*data;

// 文件描述符数量

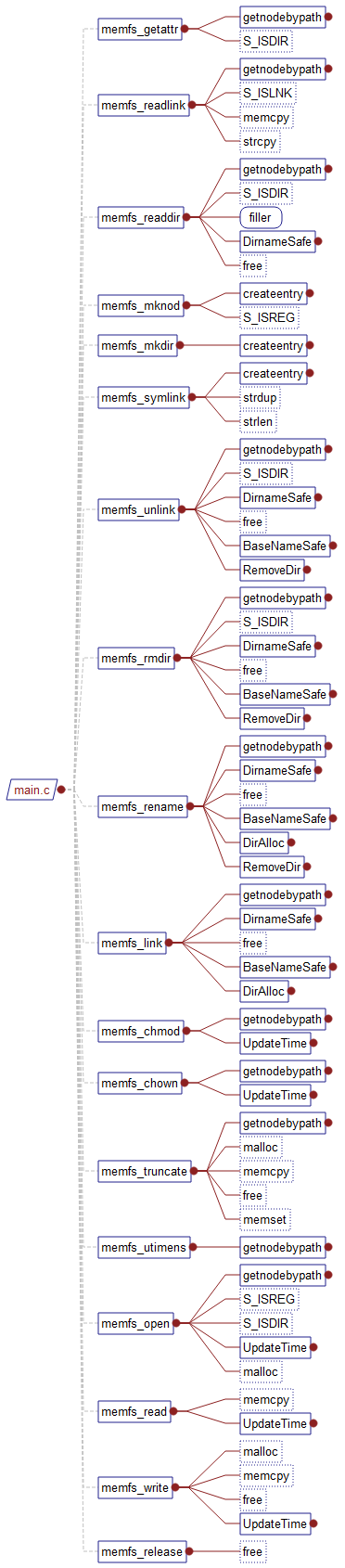
unsigned int fd\_count;

// 删除前关闭文件

int delete\_on\_close;

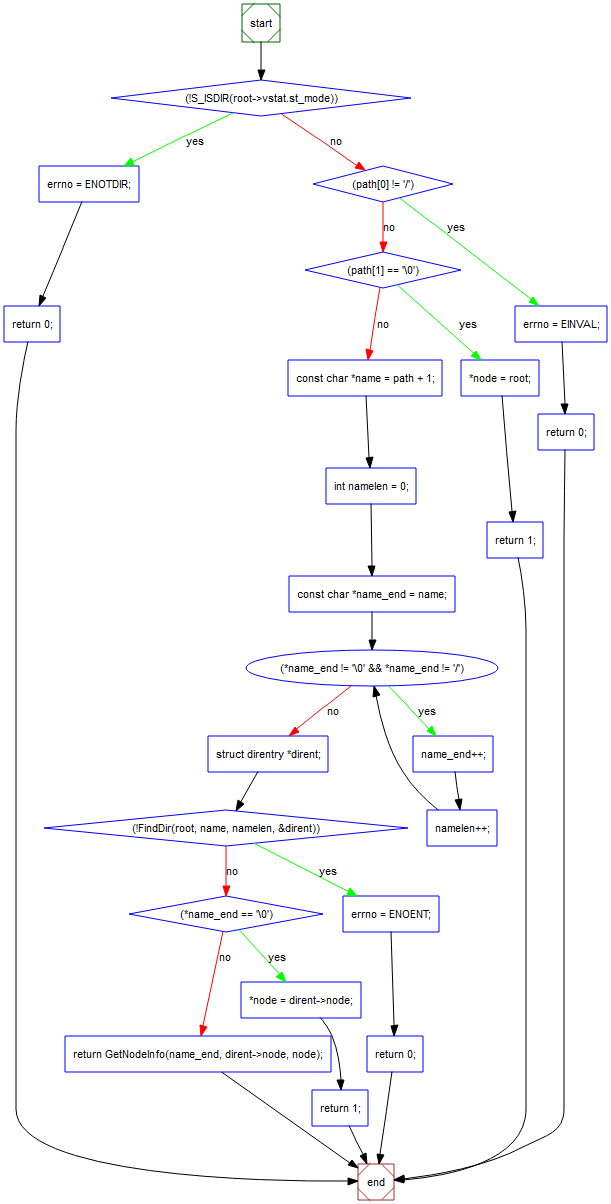
};

### 项目函数调用情况如下，

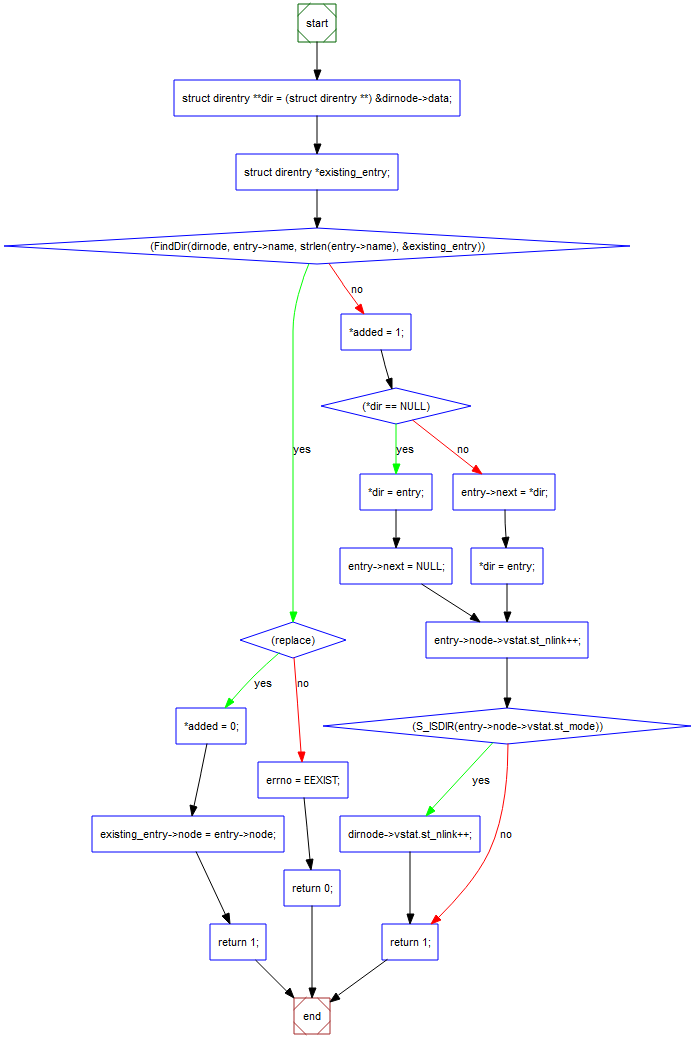


### 项目重要函数流程图如下，

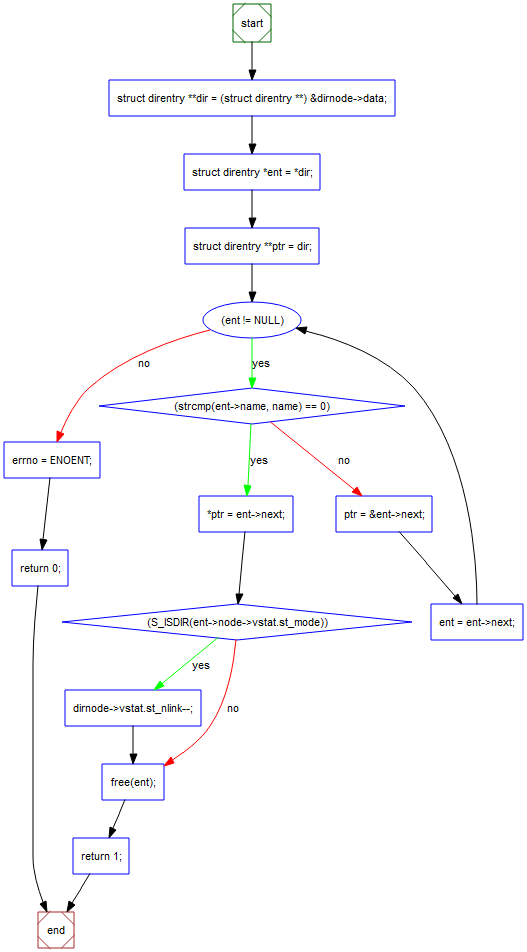
#### 获取节点信息



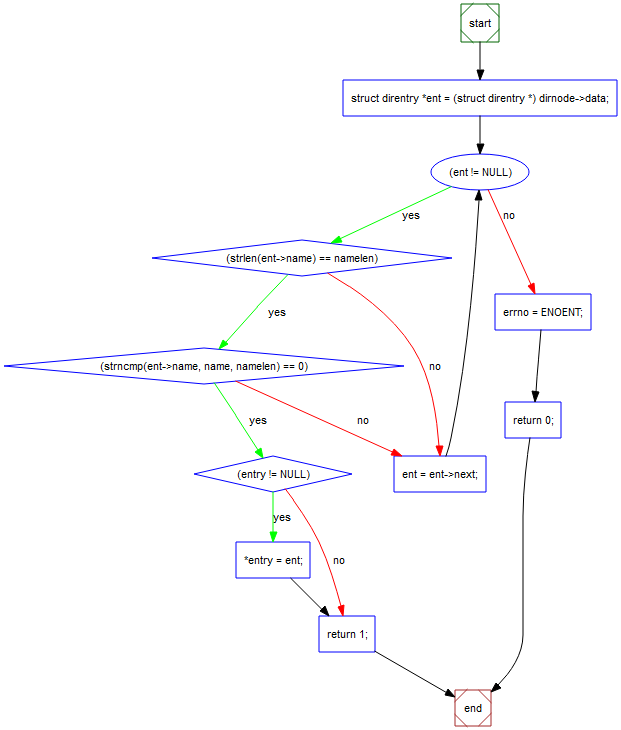
#### 添加目录



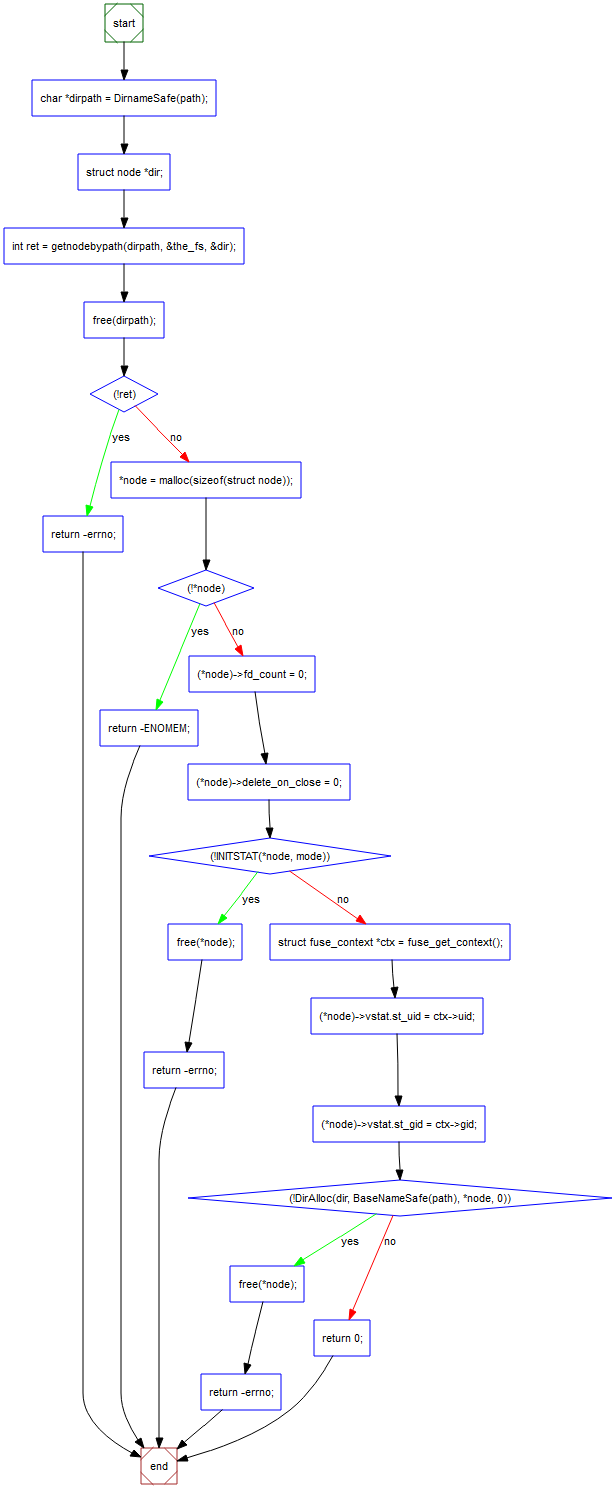
#### 删除目录



#### 查找目录



#### 创建入口



## 遇到的问题

在程序运行的过程中遇到Segment fault错误，通过使用gdb和coredump进行内存分析，从而定位问题。

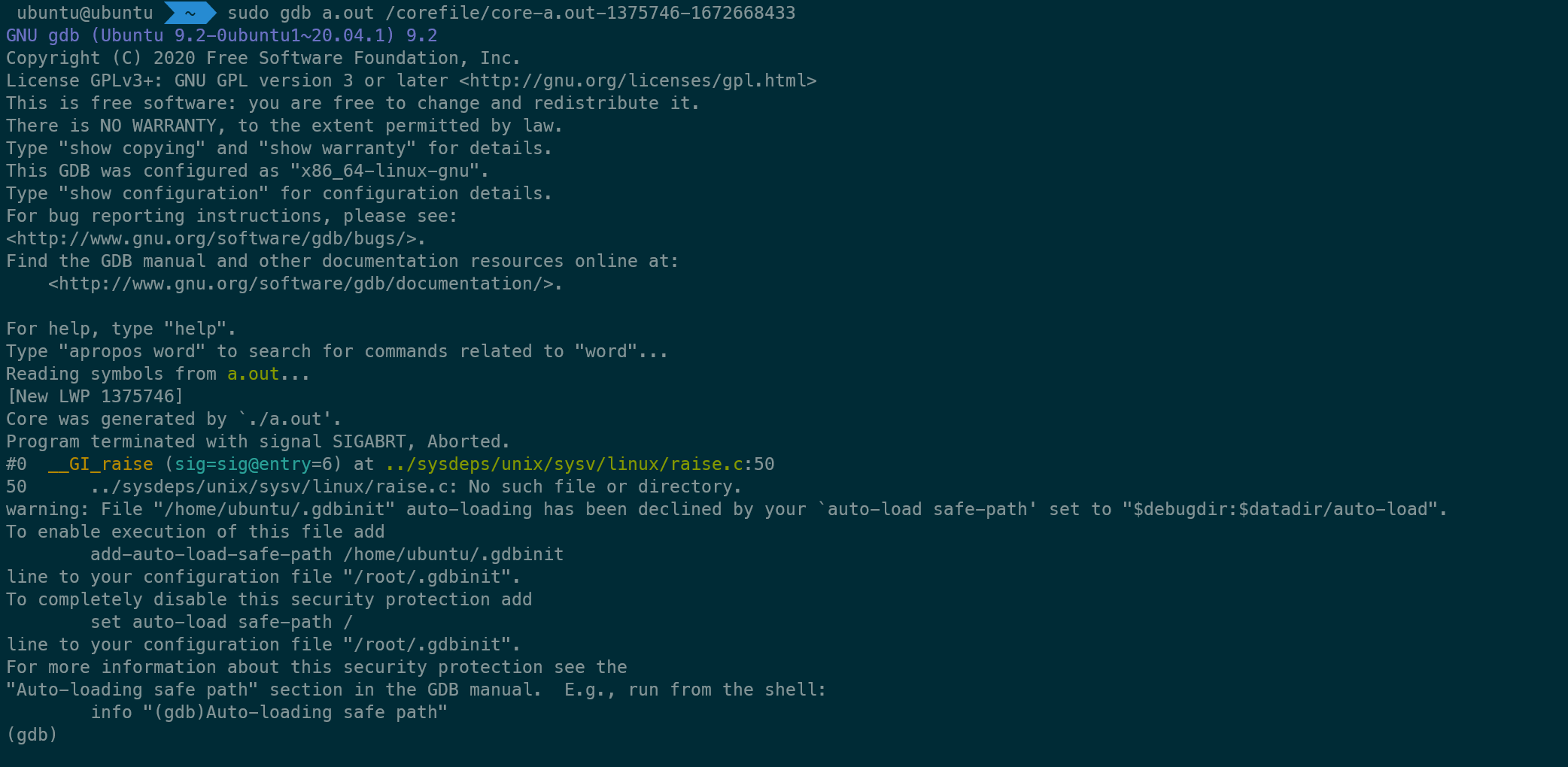
#### 配置内核

%e, %p, %t为格式化字符串

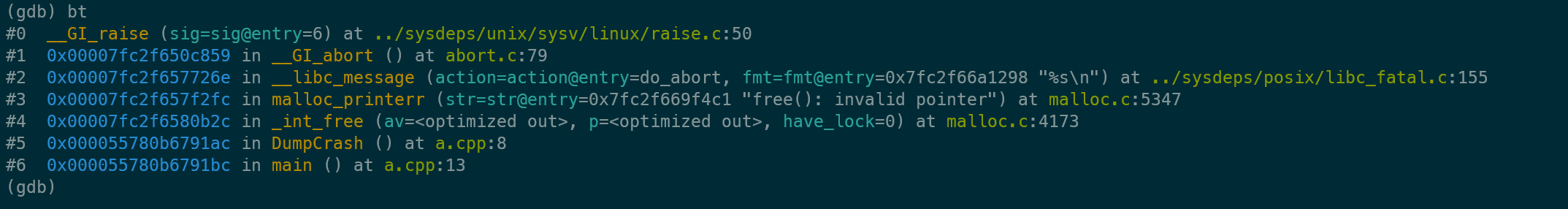
echo "/corefile/core-%e-%p-%t" > /proc/sys/kernel/core\_pattern

#### 调试

注意这里要开sudo不然权限不足

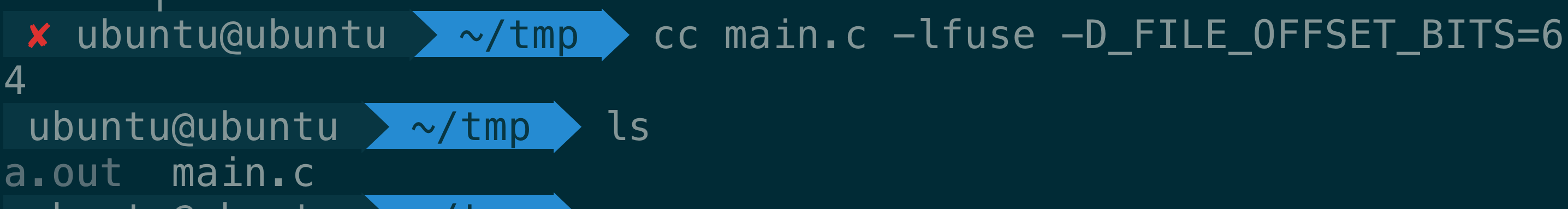


栈回溯，这里可以看到问题发生在的free()函数中，查看源码，发现时free一个野指针导致。

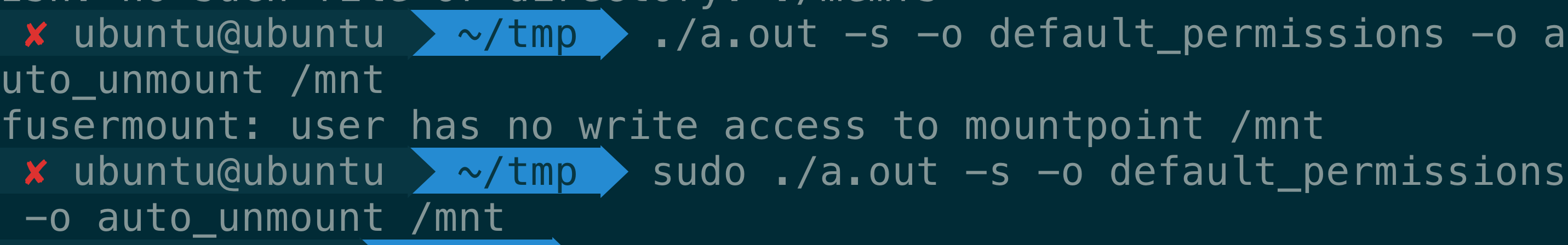


## 运行结果及说明

### 编译程序



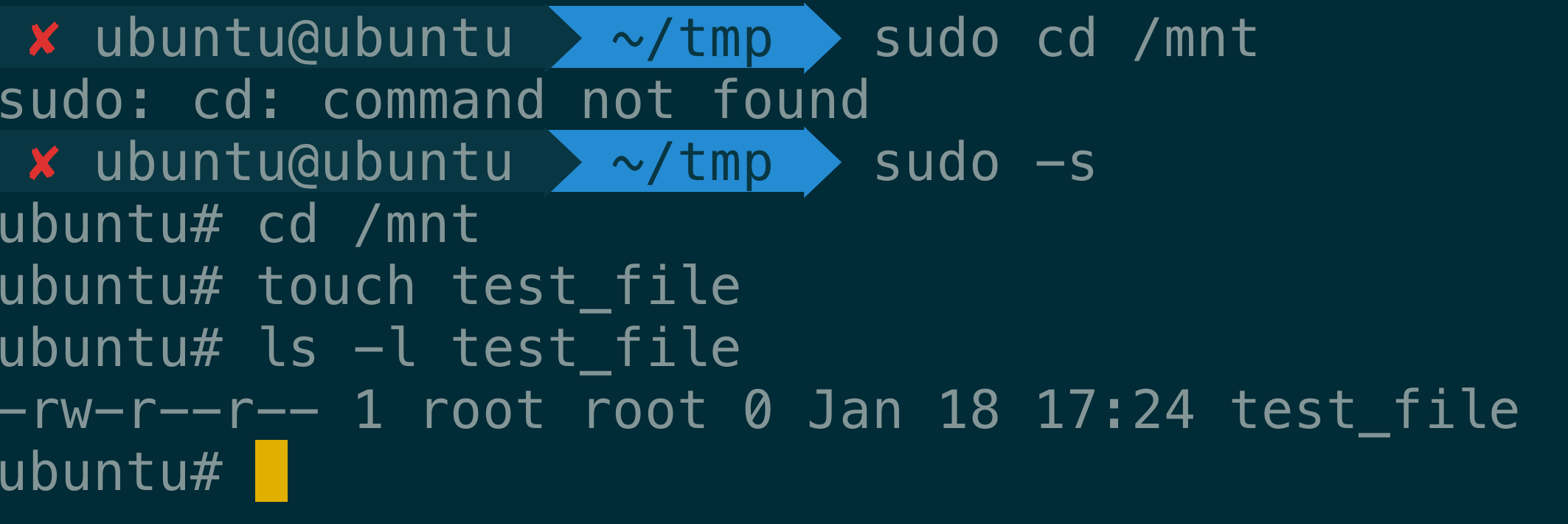
### 挂载目录



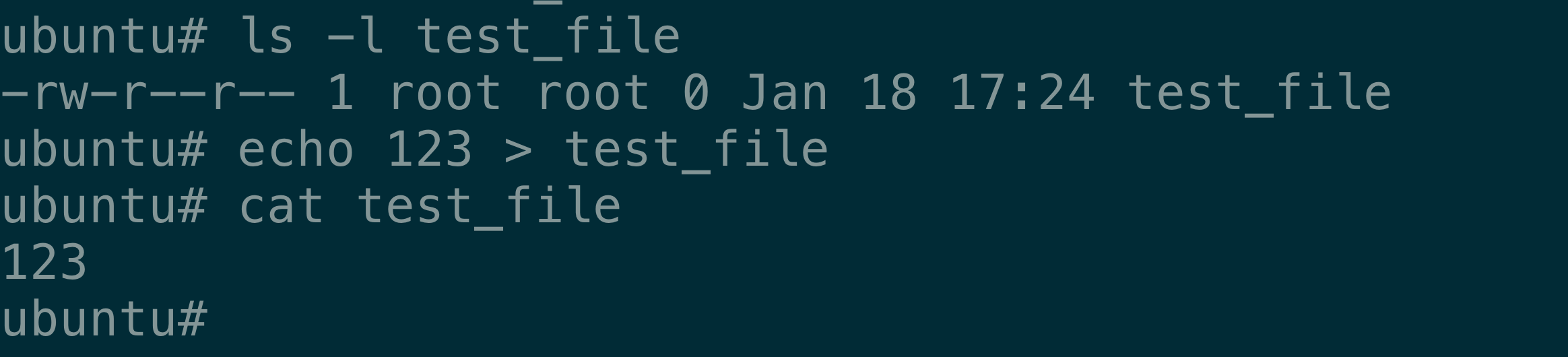
查看挂载结果



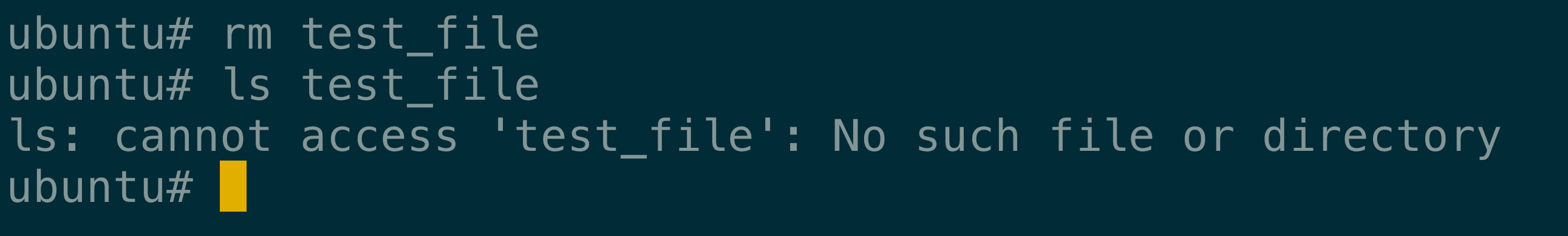
### 创建文件



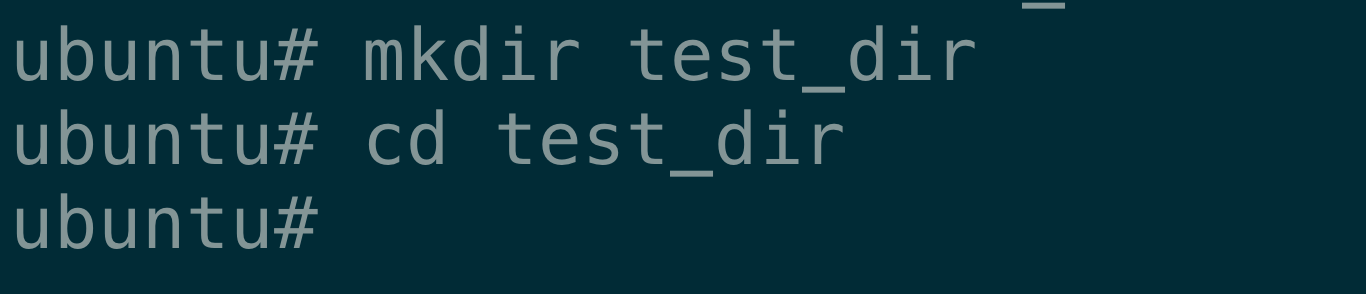
### 打开并且写入文件



### 删除文件



### 创建目录



## 运行截图

