**操作系统课程设计报告**

**姓 名：**

**学 号：**

**班 级：**

**专 业： 计算机科学与技术**

**指导教师：**

目录

**[一、课程设计的目的 4](#_Toc170978538)**

**[二、课程设计的要求 4](#_Toc170978539)**

**[三、实验环境 4](#_Toc170978540)**

**[四、课程设计的内容 5](#_Toc170978541)**

**[1.内容一 5](#_Toc170978542)**

**[1.1设计思想 5](#_Toc170978543)**

**[1.2 数据结构 5](#_Toc170978544)**

**[1.3 源代码 5](#_Toc170978545)**

**[1.4 编译、运行过程 9](#_Toc170978546)**

**[1.5 问题以及解决方式 11](#_Toc170978547)**

**[2.内容二 12](#_Toc170978548)**

**[2.1 磁盘管理 12](#_Toc170978549)**

**[2.2 页面置换 17](#_Toc170978550)**

**[3.内容三 23](#_Toc170978551)**

**[3.1 设计思想 23](#_Toc170978552)**

**[3.2 数据结构 23](#_Toc170978553)**

**[3.3 源代码 23](#_Toc170978554)**

**[3.4 编译、运行过程 28](#_Toc170978555)**

**[3.5 问题以及解决方式 28](#_Toc170978556)**

**[4.内容四 30](#_Toc170978557)**

**[4.1 设计思想 30](#_Toc170978558)**

**[4.2 数据结构 30](#_Toc170978559)**

**[4.3 源代码 30](#_Toc170978560)**

**[4.4 编译、运行过程 34](#_Toc170978561)**

**[4.5 问题以及解决方式 35](#_Toc170978562)**

**[5.内容五 36](#_Toc170978563)**

**[5.1 设计思想 36](#_Toc170978564)**

**[5.2 数据结构 36](#_Toc170978565)**

**[5.3 源代码 36](#_Toc170978566)**

**[5.4 编译、运行过程 39](#_Toc170978567)**

**[5.5 问题以及解决方式 41](#_Toc170978568)**

**[五、课程设计总结 42](#_Toc170978569)**

# 

# 一、课程设计的目的

操作系统是计算机系统的核心系统软件，它负责控制和管理整个系统的资源并组织用户协调使用这些资源，使计算机高效的工作。《操作系统系统课程设计》是《操作系统》理论课的必要补充，是复习和检验所学课程的重要手段，本课程设计的目的是综合应用学生所学知识，通过实验环节，加深学生对操作系统基本原理和工作过程的理解，提高学生独立分析问题、解决问题的能力，增强学生的动手能力。

# 二、课程设计的要求

1．分析设计内容，给出解决方案（要说明设计实现的原理，采用的数据结构）。

2．画出程序的基本结构框图和流程图。

3．对程序的每一部分要有详细的设计分析说明。

4．源代码格式要规范。

5．设计合适的测试用例，对得到的运行结果要有分析。

6．设计中遇到的问题，设计的心得体会。

7．按期提交完整的程序代码、可执行程序和课程设计报告。

8. 独立完成，每天填写设计日志（10页）。

9. 目录按你实际内容具体编写

# 三、实验环境

Linux系统版本：Ubuntu 22.04.4 LTS

Linux内核版本：Linux 6.5.0-41-generic

gcc版本：gcc 12.3.0

vim软件版本：vim 8.2

make版本：Make 4.3

# 四、课程设计的内容

## 1.内容一

### 1.1设计思想

本程序旨在实现四种常见的CPU调度算法：先来先服务（FCFS）、最短作业优先（SJF）、时间片轮转（RR）和优先级调度算法，并计算每种算法的平均等待时间和周转时间。这些算法在操作系统中用于管理进程调度，以提高系统效率和资源利用率。

### 1.2 数据结构

typedef struct {

int pid; *// 进程ID*

int arrival\_time; *// 到达时间*

int burst\_time; *// 执行时间*

int priority; *// 优先级*

int wait\_time; *// 等待时间*

int turnaround\_time; *// 周转时间*

int remaining\_time; *// 剩余时间，用于RR算法*

} Process;

该结构体Process用于存储每个进程的相关信息，包括进程ID、到达时间、执行时间、优先级、等待时间、周转时间和剩余时间。

### 1.3 源代码

#include <stdio.h>

#include <string.h>

#define MAX 100

typedef struct {

int pid; *// 进程ID*

int arrival\_time; *// 到达时间*

int burst\_time; *// 执行时间*

int priority; *// 优先级*

int wait\_time; *// 等待时间*

int turnaround\_time; *// 周转时间*

int remaining\_time; *// 剩余时间，用于RR算法*

} Process;

*// 按到达时间对进程进行排序的函数*

void sortByArrival(Process p[], int n) {

Process temp;

for(int i=0; i<n-1; i++) {

for(int j=0; j<n-1-i; j++) {

if(p[j].arrival\_time > p[j+1].arrival\_time) {

temp = p[j];

p[j] = p[j+1];

p[j+1] = temp;

}

}

}

}

*// 先来先服务（FCFS）算法*

void FCFS(Process p[], int n) {

int time = 0;

for(int i=0; i<n; i++) {

if(time < p[i].arrival\_time) {

time = p[i].arrival\_time;

}

p[i].wait\_time = time - p[i].arrival\_time; *// 计算等待时间*

time += p[i].burst\_time; *// 更新时间*

p[i].turnaround\_time = p[i].wait\_time + p[i].burst\_time; *// 计算周转时间*

}

}

*// 最短作业优先（SJF）算法*

void SJF(Process p[], int n) {

int time = 0, completed = 0, min\_index;

int visited[MAX] = {0}; *// 访问标记数组*

while(completed != n) {

min\_index = -1;

for(int i=0; i<n; i++) {

if(!visited[i] && (min\_index == -1 || p[i].burst\_time < p[min\_index].burst\_time) && p[i].arrival\_time <= time) {

min\_index = i;

}

}

if(min\_index != -1) {

visited[min\_index] = 1;

time += p[min\_index].burst\_time;

p[min\_index].wait\_time = time - p[min\_index].arrival\_time - p[min\_index].burst\_time;

p[min\_index].turnaround\_time = time - p[min\_index].arrival\_time;

completed++;

} else {

time++;

}

}

}

*// 时间片轮转（RR）算法*

void RR(Process p[], int n, int quantum) {

int time = 0, completed = 0;

while(completed != n) {

int all\_done = 1; *// 标记是否所有进程都完成*

for(int i=0; i<n; i++) {

if(p[i].remaining\_time > 0 && p[i].arrival\_time <= time) {

all\_done = 0;

if(p[i].remaining\_time > quantum) {

time += quantum;

p[i].remaining\_time -= quantum;

} else {

time += p[i].remaining\_time;

p[i].remaining\_time = 0;

p[i].turnaround\_time = time - p[i].arrival\_time;

p[i].wait\_time = p[i].turnaround\_time - p[i].burst\_time;

completed++;

}

}

}

if (all\_done) { *// 如果所有进程都完成，跳出循环*

break;

}

time++;

}

}

*// 优先级调度算法*

void Priority(Process p[], int n) {

int time = 0, completed = 0, min\_index;

int visited[MAX] = {0};

while(completed != n) {

min\_index = -1;

for(int i=0; i<n; i++) {

if(!visited[i] && (min\_index == -1 || p[i].priority < p[min\_index].priority) && p[i].arrival\_time <= time) {

min\_index = i;

}

}

if(min\_index != -1) {

visited[min\_index] = 1;

time += p[min\_index].burst\_time;

p[min\_index].wait\_time = time - p[min\_index].arrival\_time - p[min\_index].burst\_time;

p[min\_index].turnaround\_time = time - p[min\_index].arrival\_time;

completed++;

} else {

time++;

}

}

}

*// 计算平均等待时间和周转时间的函数*

void calculateAverageTimes(Process p[], int n, const char \*algorithm) {

int total\_wait\_time = 0, total\_turnaround\_time = 0;

for(int i=0; i<n; i++) {

total\_wait\_time += p[i].wait\_time;

total\_turnaround\_time += p[i].turnaround\_time;

}

printf("%s算法：\n", algorithm); *// 打印算法名称*

printf("平均等待时间：%.3f\n", (float)total\_wait\_time / n); *// 打印平均等待时间*

printf("平均周转时间：%.3f\n\n", (float)total\_turnaround\_time / n); *// 打印平均周转时间*

}

int main() {

int n = 10, quantum = 2;

Process p[] = {

{1, 0, 5, 1, 0, 0, 5},

{2, 1, 3, 3, 0, 0, 3},

{3, 2, 8, 2, 0, 0, 8},

{4, 3, 6, 4, 0, 0, 6},

{5, 4, 2, 1, 0, 0, 2},

{6, 5, 4, 3, 0, 0, 4},

{7, 6, 7, 2, 0, 0, 7},

{8, 7, 3, 4, 0, 0, 3},

{9, 8, 1, 1, 0, 0, 1},

{10, 9, 5, 3, 0, 0, 5}

};

*// 为FCFS和SJF按到达时间排序*

sortByArrival(p, n);

*// 执行FCFS算法并计算平均等待时间和周转时间*

Process temp[MAX];

memcpy(temp, p, n \* sizeof(Process));

FCFS(temp, n);

calculateAverageTimes(temp, n, "FCFS");

*// 执行SJF算法并计算平均等待时间和周转时间*

memcpy(temp, p, n \* sizeof(Process));

SJF(temp, n);

calculateAverageTimes(temp, n, "SJF");

*// 执行RR算法并计算平均等待时间和周转时间*

memcpy(temp, p, n \* sizeof(Process));

RR(temp, n, quantum);

calculateAverageTimes(temp, n, "RR");

*// 执行优先级调度算法并计算平均等待时间和周转时间*

memcpy(temp, p, n \* sizeof(Process));

Priority(temp, n);

calculateAverageTimes(temp, n, "Priority");

return 0;

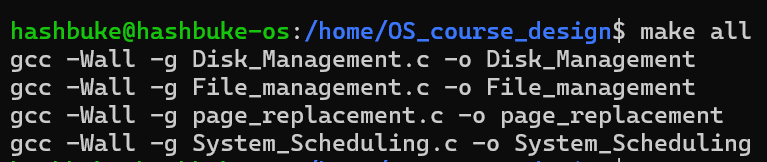
}

### 1.4 编译、运行过程

**编译程序**：

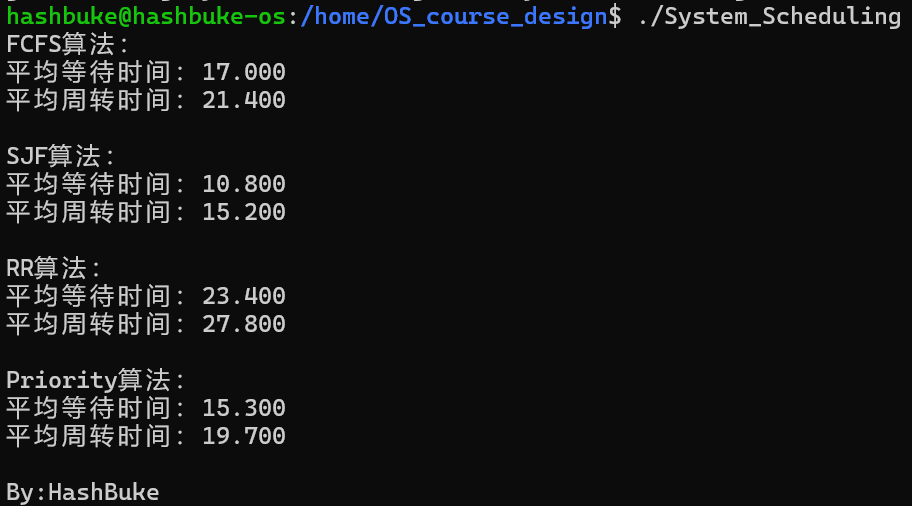
打开终端或命令提示符，导航到包含源代码文件的目录，利用**Makefile**文件使用以下命令编译程序：

make all

****

**运行程序**：

在终端或命令提示符中运行编译后的可执行文件：./System\_Scheduling



程序将按照先来先服务、最短作业优先、时间片轮转和优先级调度算法顺序执行，并输出每种算法的平均等待时间和周转时间。

### 1.5 问题以及解决方式

**遇到的问题与解决方案**

1. 算法实现的复杂性

在实现这些调度算法时，我发现每种算法都有其独特的复杂性。例如，SJF算法需要准确预测每个进程的执行时间，这在实际操作中可能并不容易。为了解决这个问题，我编写了一个函数来模拟执行时间的估计，并引入了一些假设以简化模型。

2. 时间片轮转中的公平性

在实现RR算法时，确保所有进程在给定的时间片内得到公平的CPU时间是一个挑战。我通过创建一个循环队列来管理进程，并在每个时间片结束后将当前进程移到队列末尾，从而确保了调度的公平性。

3. 优先级调度中的优先级反转

在优先级调度算法中，处理优先级反转问题尤为关键。为了应对这一问题，我引入了优先级继承机制，即当高优先级进程等待低优先级进程释放资源时，临时提升低优先级进程的优先级，从而避免优先级反转带来的问题。

**收获体会**

通过这个实验，我深刻体会到了操作系统设计的复杂性和精妙之处。每种调度算法都有其优缺点，如何在不同场景下选择合适的调度算法是一门艺术。同时，这也让我意识到，理论与实际应用之间存在一定的差距，解决实际问题时需要灵活应用理论知识并进行适当的简化和假设。

此外，这个实验还让我认识到，程序的可维护性和扩展性同样重要。在设计和实现过程中，我特别注意代码的模块化和注释，使得代码易于理解和修改。通过不断的调试和优化，我不仅提高了编程能力，也积累了更多解决复杂问题的经验。

## 2.内容二

### 2.1 磁盘管理

#### 2.1.1 设计思想

本程序模拟了一个简单的磁盘管理系统，包含磁盘空间的初始化、显示、分配和释放功能。通过用户菜单交互，可以选择不同的操作来管理磁盘块的分配和释放，以便于理解和学习磁盘管理的基本原理。

**数据结构定义**：使用结构体Block表示磁盘块信息，包括柱面号、磁道号、物理记录号和分配状态。使用二维数组disk表示磁盘，其中每个元素为一个Block。

**初始化磁盘**：initialize\_disk函数初始化磁盘块，将所有块的分配状态设置为未分配，并为每个块分配唯一的物理记录号。

**显示磁盘状态**：display\_disk函数打印磁盘的当前分配状态，已分配块用+表示，未分配块用空格表示。

**分配磁盘块**：allocate\_blocks函数尝试分配指定数量的连续块，若成功则更新这些块的分配状态，并打印分配信息。

**释放磁盘块**：free\_blocks函数根据指定的柱面号、磁道号、物理记录号和长度释放相应的磁盘块，并更新其分配状态。

**用户交互**：主函数main提供一个菜单，用户可以选择显示磁盘状态、分配空间、释放空间或退出程序。

#### 2.1.2 数据结构

typedef struct {

int cylinder; *// 柱面号*

int track; *// 磁道号*

int record; *// 物理记录号*

bool is\_allocated; *// 分配状态*

} Block;

结构体Block用于存储每个磁盘块的信息，包括柱面号、磁道号、物理记录号和分配状态。

#### 2.1.3 源代码

#include <stdio.h>

#include <stdbool.h>

#include <string.h>

#define CYLINDERS 10

#define TRACKS 20

*// 定义磁盘块结构体*

typedef struct {

int cylinder; *// 柱面号*

int track; *// 磁道号*

int record; *// 物理记录号*

bool is\_allocated; *// 分配状态*

} Block;

Block disk[CYLINDERS][TRACKS];

*// 初始化磁盘块，将所有块设置为未分配状态，并分配物理记录号*

void initialize\_disk() {

for (int i = 0; i < CYLINDERS; i++) {

for (int j = 0; j < TRACKS; j++) {

disk[i][j].cylinder = i;

disk[i][j].track = j;

disk[i][j].record = i \* TRACKS + j; *// 物理记录号按照顺序分配*

disk[i][j].is\_allocated = false;

}

}

}

*// 显示磁盘当前分配状态*

void display\_disk() {

printf("\n磁盘空间:\n");

for (int i = 0; i < CYLINDERS; i++) {

printf("|");

for (int j = 0; j < TRACKS; j++) {

if (disk[i][j].is\_allocated) {

printf("+");

} else {

printf(" ");

}

}

printf("|\n");

}

printf("\n");

}

*// 分配指定数量的连续磁盘块*

bool allocate\_blocks(int blocks) {

int count = 0;

for (int i = 0; i < CYLINDERS; i++) {

for (int j = 0; j < TRACKS; j++) {

if (!disk[i][j].is\_allocated) {

count++;

if (count == blocks) {

*// 分配块*

for (int k = 0; k < blocks; k++) {

disk[i][j - k].is\_allocated = true;

printf("分配块: 柱面号=%d, 磁道号=%d, 物理记录号=%d\n", disk[i][j - k].cylinder, disk[i][j - k].track, disk[i][j - k].record);

}

return true;

}

} else {

count = 0;

}

}

}

return false;

}

*// 释放指定的磁盘块*

void free\_blocks(int cylinder, int track, int record, int length) {

int count = 0;

for (int i = cylinder; i < CYLINDERS; i++) {

for (int j = (i == cylinder ? track : 0); j < TRACKS; j++) {

if (disk[i][j].is\_allocated && disk[i][j].record == record + count) {

disk[i][j].is\_allocated = false;

printf("释放块: 柱面号=%d, 磁道号=%d, 物理记录号=%d\n", disk[i][j].cylinder, disk[i][j].track, disk[i][j].record);

count++;

if (count == length) {

return;

}

}

}

}

printf("释放失败: 找不到足够的块或块未分配\n");

}

*// 主函数，提供用户菜单进行交互*

int main() {

initialize\_disk();

int choice, blocks, length;

int cylinder, track, record;

while (1) {

printf("\n菜单:\n1. 显示磁盘空间\n2. 分配空间\n3. 释放空间\n4. 退出\n选择功能（1-4）：");

scanf("%d", &choice);

switch (choice) {

case 1:

display\_disk();

break;

case 2:

printf("请输入要分配的块数: ");

scanf("%d", &blocks);

if (allocate\_blocks(blocks)) {

printf("分配成功\n");

} else {

printf("分配失败，磁盘空间不足或无法找到连续的空闲块\n");

}

break;

case 3:

printf("请输入要释放的块的柱面号、磁道号、物理记录号和删除长度: ");

scanf("%d %d %d %d", &cylinder, &track, &record, &length);

free\_blocks(cylinder, track, record, length);

break;

case 4:

return 0;

default:

printf("无效选择\n");

}

}

return 0;

}

#### 2.1.4 编译、运行过程

**编译程序**：打开终端或命令提示符。导航到包含源代码文件的目录。使用以下命令编译程序：

gcc -o Disk\_Management Disk\_Management.c

**运行程序**：在终端或命令提示符中运行编译后的可执行文件：

./Disk\_Management



程序将启动并提供菜单，用户可以选择显示磁盘状态、分配空间、释放空间或退出程序。

#### 2.1.5 问题以及解决方式

**遇到的问题与解决方案**

在设计磁盘管理系统时，我需要考虑如何有效地表示磁盘块以及它们的分配状态。我决定使用一个二维数组来模拟磁盘，每个元素表示一个磁盘块，并包含柱面号、磁道号、物理记录号以及分配状态。这种方式直观且易于理解，但需要确保数组的索引与物理记录号的一一对应关系。

分配磁盘块是一个相对复杂的过程。allocate\_blocks函数需要找到连续的未分配块，并将它们标记为已分配。这部分的难点在于如何高效地查找连续的空闲块，并处理边界条件。

**收获体会**

通过这个实验，我加深了对磁盘管理系统基本原理的理解，并且在编写过程中，学会了如何处理二维数组、管理内存分配和释放，以及如何设计一个用户友好的交互界面。

### 2.2 页面置换

#### 2.2.1 设计思想

本程序模拟了虚拟内存管理中的页面置换算法，具体包括先进先出（FIFO）页面置换算法和最近最少使用（LRU）页面置换算法。通过生成随机页面序列并应用两种算法处理缺页中断，旨在演示和比较这两种常见的页面置换策略。

**页面生成**：使用随机数生成函数生成指定数量的页面序列，用于模拟页面访问。

**页面检测**：实现一个函数检测某个页面是否在内存中。

**页面打印**：实现函数打印当前的页面序列和内存状态。

**FIFO算法**：使用一个循环队列来模拟FIFO页面置换策略，记录页面中断次数。

**LRU算法**：使用一个年龄数组记录页面的使用情况，实现LRU页面置换策略，记录页面中断次数。

**主函数**：获取用户输入的内存容量和页面数量，生成随机页面序列，并分别调用FIFO和LRU算法处理页面置换。

#### 2.2.2 数据结构的说明

int pages[]; *// 用于存储生成的随机页面序列*

int memory[]; *// 用于模拟内存块，存储当前在内存中的页面*

int age[]; *// 用于记录每个内存块中页面的使用情况（仅用于LRU算法）*

#### 2.2.3 源代码

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX 100

*// 生成随机页面序列*

void generatePages(int pages[], int n) {

srand(time(NULL));

for (int i = 0; i < n; i++) {

pages[i] = rand() % 10 + 1;

}

}

*// 检查页面是否在内存中*

int isPageInMemory(int page, int memory[], int capacity) {

for (int i = 0; i < capacity; i++) {

if (memory[i] == page) {

return 1;

}

}

return 0;

}

*// 打印页面序列*

void printPages(int pages[], int n) {

for (int i = 0; i < n; i++) {

printf("%d ", pages[i]);

}

printf("\n");

}

*// 打印当前内存状态*

void printMemory(int memory[], int capacity) {

for (int i = 0; i < capacity; i++) {

if (memory[i] == -1) {

printf("| ");

} else {

printf("|%d", memory[i]);

}

}

printf("|\n");

}

*// 先进先出（FIFO）页面置换算法*

void fifo(int pages[], int n, int capacity) {

int memory[capacity];

int pageFaults = 0;

int index = 0;

for (int i = 0; i < capacity; i++) {

memory[i] = -1;

}

printf("FIFO 过程:\n");

for (int i = 0; i < n; i++) {

printf("当前插入页: %d\n", pages[i]);

if (!isPageInMemory(pages[i], memory, capacity)) {

memory[index] = pages[i];

index = (index + 1) % capacity;

pageFaults++;

}

printMemory(memory, capacity);

}

printf("FIFO 页面中断次数: %d\n", pageFaults);

}

*// 最近最少使用（LRU）页面置换算法*

void lru(int pages[], int n, int capacity) {

int memory[capacity];

int pageFaults = 0;

int age[capacity];

for (int i = 0; i < capacity; i++) {

memory[i] = -1;

age[i] = 0;

}

printf("LRU 过程:\n");

for (int i = 0; i < n; i++) {

printf("当前插入页: %d\n", pages[i]);

int minAge = i;

int minIndex = -1;

if (!isPageInMemory(pages[i], memory, capacity)) {

for (int j = 0; j < capacity; j++) {

if (memory[j] == -1) {

minIndex = j;

break;

}

if (age[j] < minAge) {

minAge = age[j];

minIndex = j;

}

}

memory[minIndex] = pages[i];

pageFaults++;

}

for (int j = 0; j < capacity; j++) {

if (memory[j] != -1) {

age[j]++;

}

}

age[minIndex] = 0;

printMemory(memory, capacity);

}

printf("LRU 页面中断次数: %d\n", pageFaults);

}

*// 主函数，提供用户输入和调用页面置换算法*

int main() {

int capacity, n;

printf("输入一次可以保存在内存中的页数：");

scanf("%d", &capacity);

printf("输入要处理的页数：");

scanf("%d", &n);

int pages[n];

generatePages(pages, n);

printf("Pages: ");

printPages(pages, n);

fifo(pages, n, capacity);

lru(pages, n, capacity);

return 0;

}

#### 2.2.4 编译、运行过程

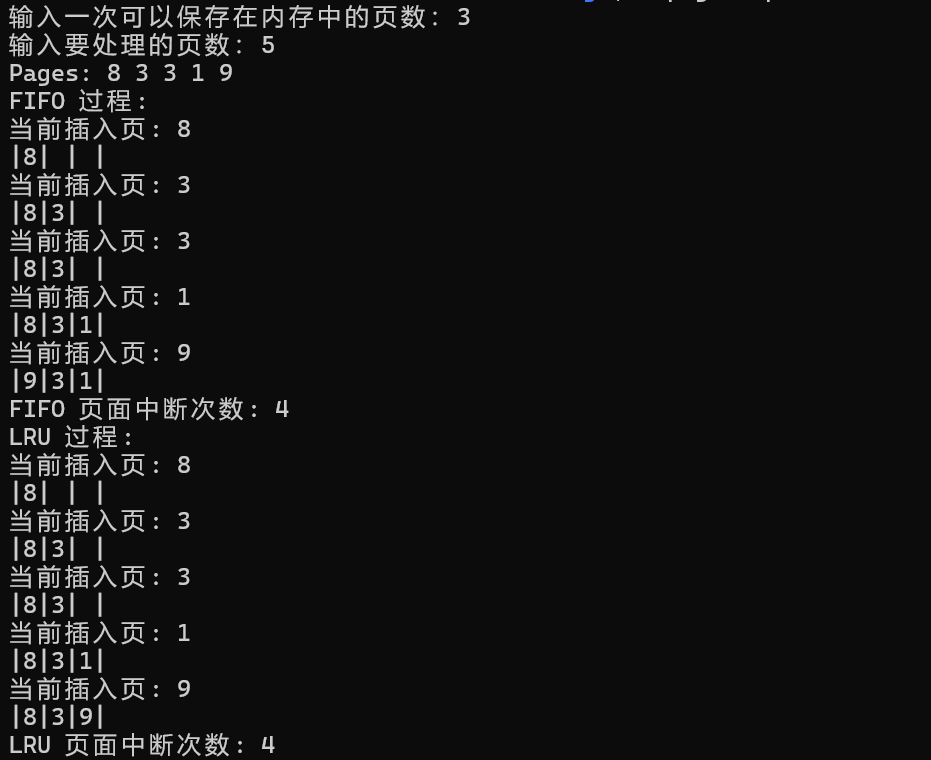
**编译程序**：打开终端或命令提示符，导航到包含源代码文件的目录。使用以下命令编译程序：

gcc -o page\_replacement page\_replacement.c

**运行程序**：在终端或命令提示符中运行编译后的可执行文件：

./page\_replacement

程序将启动并要求用户输入内存容量和要处理的页面数量，然后生成随机页面序列并依次执行FIFO和LRU页面置换算法，最后输出每种算法的页面中断次数。



#### 2.2.5 问题以及解决方式

**遇到的问题与解决方案**

在生成随机页面序列的功能上，我需要确保生成的页面序列具有随机性并且覆盖到可能的页面范围。为此，我使用了C语言中的标准库函数生成随机数，但遇到的问题是每次运行程序时，随机序列竟然没有变化。经过调研，我了解到需要在程序开始时使用当前时间来初始化随机数种子，这样才能确保每次运行程序时生成的序列是不同的。

在设计页面置换算法时，我需要实现两种经典的算法：先进先出（FIFO）和最近最少使用（LRU）。在编写FIFO算法时，我发现需要一种简单有效的方法来管理内存中的页面顺序。我选择了一个循环队列的方式来管理页面的替换，确保每次页面中断时都能正确地替换最早进入内存的页面。

在实现LRU算法时，挑战在于如何有效地跟踪每个页面的最近使用时间。我决定使用一个数组来记录每个页面的“年龄”，并在每次页面访问时更新它们。这个方案虽然简单，但需要在每次页面访问时遍历整个内存来找到最老的页面。通过这种方法，我能够准确地实现LRU的页面替换逻辑。

在调试过程中，我遇到了几个边界情况，例如内存刚开始时为空的处理，以及页面命中时不需要进行替换的处理。这些细节问题需要通过反复测试和调整代码来解决。在调试过程中，我还实现了多个打印功能，用于输出当前的内存状态和页面序列，以便更直观地观察算法的运行过程。

**收获体会**

通过实验，我深刻体会到了内存管理和页面置换算法在操作系统中的重要性。每一种算法都有其独特的应用场景和优缺点，理解并实现这些算法不仅加深了我对操作系统原理的理解，也锻炼了我的编程能力和问题解决能力。在实际应用中，选择合适的页面置换策略可以显著提高系统性能，这也是我在编写和优化这段程序时的最大收获。

## 3.内容三

### 3.1 设计思想

本程序旨在实现一个简单的Linux字符设备驱动程序，用于演示字符设备驱动的基本功能，包括设备的打开、关闭、读写操作。程序还包括一个用户空间测试程序，用于与驱动程序进行交互，验证驱动程序的功能。

**字符设备驱动程序**：初始化字符设备并注册设备号，创建设备类和设备实例，实现文件操作接口，包括打开、关闭、读、写操作。在读操作中返回当前系统时间，在写操作中接受一个用户输入的数字，并将其打印为二进制格式。

**用户空间测试程序**：打开字符设备文件，读操作获取当前系统时间，写操作将用户输入的数字发送给驱动程序，将输入的数字转换为二进制字符串并显示。

**Makefile**：用于编译内核模块。

### 3.2 数据结构

static int major\_number; *// 设备的主设备号*

static struct class\* hashbuke\_class = NULL; *// 设备类指针*

static struct device\* hashbuke\_device = NULL; *// 设备指针*

### 3.3 源代码

#### 驱动程序源代码 hashbuke\_driver.c

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/uaccess.h>

#include <linux/time.h>

#include <linux/device.h>

#include <linux/kdev\_t.h>

#define DEVICE\_NAME "hashbuke"

#define CLASS\_NAME "hashbuke\_class"

static int major\_number;

static struct class\* hashbuke\_class = NULL;

static struct device\* hashbuke\_device = NULL;

*// 文件操作接口函数声明*

static ssize\_t hashbuke\_read(struct file \*filep, char \*buffer, size\_t len, loff\_t \*offset);

static ssize\_t hashbuke\_write(struct file \*filep, const char \*buffer, size\_t len, loff\_t \*offset);

static int hashbuke\_open(struct inode \*inodep, struct file \*filep);

static int hashbuke\_release(struct inode \*inodep, struct file \*filep);

*// 文件操作结构体*

static struct file\_operations fops = {

.open = hashbuke\_open,

.read = hashbuke\_read,

.write = hashbuke\_write,

.release = hashbuke\_release,

};

*// 初始化模块*

static int \_\_init hashbuke\_init(void) {

printk(KERN\_INFO "Hello, HashBuke\n");

major\_number = register\_chrdev(0, DEVICE\_NAME, &fops);

if (major\_number < 0) {

printk(KERN\_ALERT "注册字符设备失败\n");

return major\_number;

}

hashbuke\_class = class\_create(THIS\_MODULE, CLASS\_NAME);

if (IS\_ERR(hashbuke\_class)) {

unregister\_chrdev(major\_number, DEVICE\_NAME);

printk(KERN\_ALERT "注册设备类别失败\n");

return PTR\_ERR(hashbuke\_class);

}

hashbuke\_device = device\_create(hashbuke\_class, NULL, MKDEV(major\_number, 0), NULL, DEVICE\_NAME);

if (IS\_ERR(hashbuke\_device)) {

class\_destroy(hashbuke\_class);

unregister\_chrdev(major\_number, DEVICE\_NAME);

printk(KERN\_ALERT "创建设备失败\n");

return PTR\_ERR(hashbuke\_device);

}

printk(KERN\_INFO "HashBuke: 设备类创建正确\n");

return 0;

}

*// 退出模块*

static void \_\_exit hashbuke\_exit(void) {

device\_destroy(hashbuke\_class, MKDEV(major\_number, 0));

class\_unregister(hashbuke\_class);

class\_destroy(hashbuke\_class);

unregister\_chrdev(major\_number, DEVICE\_NAME);

printk(KERN\_INFO "Bye, HashBuke\n");

}

*// 打开设备*

static int hashbuke\_open(struct inode \*inodep, struct file \*filep) {

printk(KERN\_INFO "HashBuke:设备已打开\n");

return 0;

}

*// 释放设备*

static int hashbuke\_release(struct inode \*inodep, struct file \*filep) {

printk(KERN\_INFO "HashBuke: 设备成功关闭\n");

return 0;

}

*// 读取设备数据*

static ssize\_t hashbuke\_read(struct file \*filep, char \*buffer, size\_t len, loff\_t \*offset) {

struct timespec64 current\_time;

char time\_buffer[30];

int error\_count;

ktime\_get\_real\_ts64(&current\_time);

snprintf(time\_buffer, sizeof(time\_buffer), "%lld\n", current\_time.tv\_sec);

error\_count = copy\_to\_user(buffer, time\_buffer, strlen(time\_buffer));

if (error\_count == 0) {

printk(KERN\_INFO "HashBuke: 将当前时间发送给用户\n");

return 0;

} else {

printk(KERN\_INFO "HashBuke: 无法将当前时间发送给用户\n");

return -EFAULT;

}

}

*// 写入设备数据*

static ssize\_t hashbuke\_write(struct file \*filep, const char \*buffer, size\_t len, loff\_t \*offset) {

char input\_buffer[10];

long number;

int error\_count;

error\_count = copy\_from\_user(input\_buffer, buffer, len);

if (error\_count != 0) {

printk(KERN\_INFO "HashBuke: 无法接收用户的输入\n");

return -EFAULT;

}

input\_buffer[len] = '\0';

if (kstrtol(input\_buffer, 10, &number) != 0) {

printk(KERN\_INFO "HashBuke: 无法将输入转换为长整型\n");

return -EINVAL;

}

printk(KERN\_INFO "HashBuke: 已收到 %ld, 二进制表示: %lx\n", number, number);

return len;

}

module\_init(hashbuke\_init);

module\_exit(hashbuke\_exit);

MODULE\_LICENSE("GPL");

MODULE\_DESCRIPTION("一个简单的 Linux 字符驱动程序 HashBuke");

MODULE\_VERSION("1.0");

#### 测试程序 test\_hashbuke.c

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

#include <string.h>

#define DEVICE "/dev/hashbuke"

*// 将数字转换为二进制字符串的函数*

void to\_binary(long n, char \*binary\_str) {

int i;

for (i = 0; i < 64; i++) {

binary\_str[63 - i] = (n & 1) ? '1' : '0';

n >>= 1;

}

binary\_str[64] = '\0';

}

int main() {

int fd;

char buffer[256];

ssize\_t ret;

long number;

fd = open(DEVICE, O\_RDWR);

if (fd < 0) {

perror("Failed to open the device");

return 1;

}

*// 读取设备数据*

ret = read(fd, buffer, sizeof(buffer));

if (ret < 0) {

perror("Failed to read the message from the device");

return 1;

}

printf("Current time in seconds since 1970-01-01: %s", buffer);

*// 写入设备数据*

printf("Enter a number: ");

scanf("%ld", &number);

snprintf(buffer, sizeof(buffer), "%ld", number);

ret = write(fd, buffer, strlen(buffer));

if (ret < 0) {

perror("Failed to write the message to the device");

return 1;

}

*// 显示输入数字的二进制表示*

char binary\_str[65];

to\_binary(number, binary\_str);

printf("Binary representation: %s\n", binary\_str);

close(fd);

return 0;

}

#### Makefile

obj-m += hashbuke\_driver.o

all:

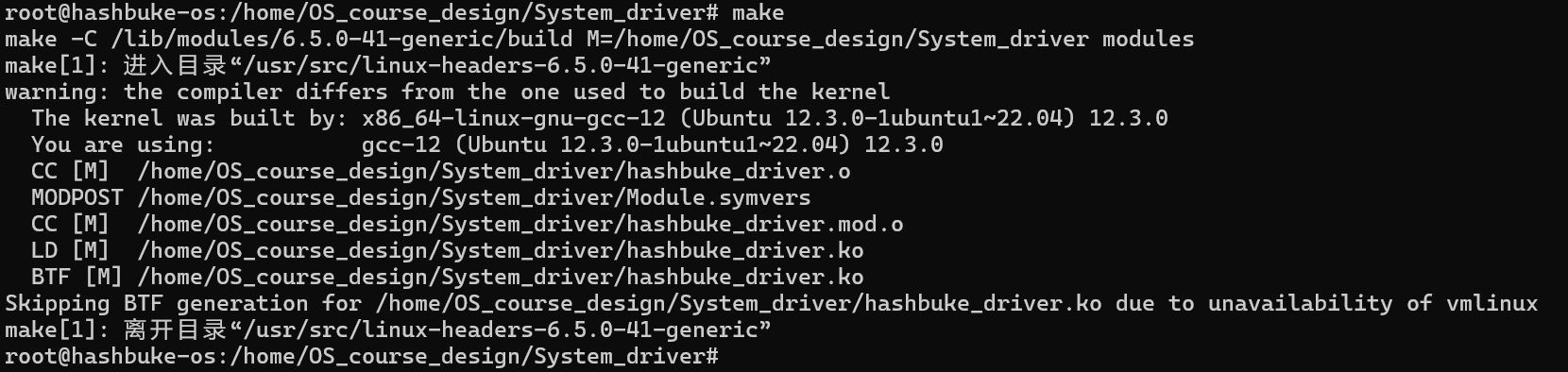
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

### 3.4 编译、运行过程

**编译内核模块**：打开终端，导航到包含hashbuke\_driver.c和Makefile的目录。使用以下命令编译内核模块：make all



**加载和卸载内核模块**：

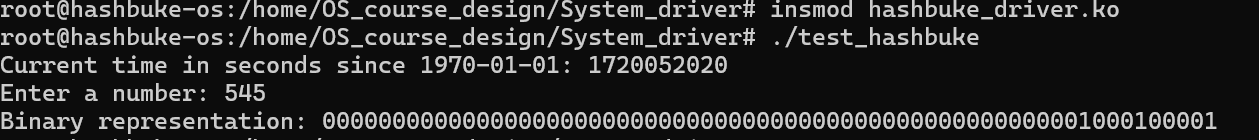
加载模块：sudo insmod hashbuke\_driver.ko

卸载模块：sudo rmmod hashbuke\_driver

**测试程序**：

编译用户空间测试程序：gcc -o test\_hashbuke test\_hashbuke.c

运行测试程序：./test\_hashbuke



程序将与字符设备驱动程序进行交互，测试读取当前时间和写入用户输入数字并显示其二进制表示。

### 3.5 问题以及解决方式

**遇到的问题与解决方案**

1. **设备注册失败**：在初始化模块时，驱动程序需要向系统注册一个字符设备。然而，我第一次运行程序时，设备注册失败，导致无法创建设备文件。我通过检查内核日志，发现是因为设备号的分配出了问题。为了避免硬编码设备号，我改用了动态分配设备号的方法，成功解决了这个问题。
2. **用户空间数据传输**：实现读写操作时，内核空间与用户空间的数据传输是一个关键点。在实现read和write接口时，我遇到了数据传输失败的情况。通过仔细查阅文档，我了解到copy\_to\_user和copy\_from\_user这两个函数是处理数据传输的关键。我在代码中正确使用了这些函数，并添加了错误检查机制，确保数据能够顺利传输。
3. **时间获取与格式化**：在read操作中，我需要将当前系统时间返回给用户空间程序。获取时间并格式化输出也是一个挑战。我使用了ktime\_get\_real\_ts64函数获取当前时间，并通过snprintf将时间格式化为字符串。经过几次调试，最终实现了将时间正确传递给用户空间程序。
4. **输入数据处理**：在write操作中，我需要处理用户输入的数据，并将其转换为长整型。最初的实现中，我对输入数据的处理不够严谨，导致输入数据无法正确解析。后来，我使用了kstrtol函数来转换字符串为长整型，并添加了错误处理，确保输入数据的有效性。

**收获体会**

通过这次字符设备驱动程序的开发，我深刻体会到了系统编程的复杂性和严谨性。内核编程不同于用户空间编程，它需要更高的稳定性和安全性。每一个细节都需要仔细考虑和处理，稍有不慎，就可能导致系统崩溃或无法正常工作。

## 4.内容四

### 4.1 设计思想

该程序实现了一个简单的文件管理工具，提供文件的创建、读取、写入、权限修改和查看功能。通过用户交互菜单，用户可以选择不同的操作来管理文件，程序会根据用户输入执行相应的文件操作。

**创建新文件**：使用open系统调用创建新文件，如果文件不存在则创建，存在则清空。

**写文件**：使用open和write系统调用将用户输入的内容追加到指定文件中。

**读文件**：使用open和read系统调用读取指定文件的内容并输出到终端。

**修改文件权限**：使用chmod系统调用修改指定文件的权限。

**查看文件权限**：使用stat系统调用获取指定文件的权限并打印详细信息。

**用户交互**：主函数提供一个菜单，用户可以选择不同的文件操作，程序会根据用户输入调用相应的函数执行操作。

### 4.2 数据结构

该程序主要使用C标准库和POSIX库函数进行文件操作，没有复杂的数据结构。

### 4.3 源代码

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <unistd.h>

*// 创建新文件*

void create\_file(const char \*filename) {

int fd = open(filename, O\_CREAT | O\_WRONLY, 0644);

if (fd == -1) {

perror("文件创建失败");

} else {

printf("文件创建成功: %s\n", filename);

close(fd);

}

}

*// 写文件*

void write\_file(const char \*filename, const char \*content) {

int fd = open(filename, O\_WRONLY | O\_APPEND);

if (fd == -1) {

perror("打开文件失败（写入）");

} else {

if (write(fd, content, strlen(content)) == -1) {

perror("写入文件失败");

} else {

printf("写入文件成功: %s\n", filename);

}

close(fd);

}

}

*// 读文件*

void read\_file(const char \*filename) {

int fd = open(filename, O\_RDONLY);

if (fd == -1) {

perror("打开文件失败（读取）");

} else {

char buffer[1024];

ssize\_t bytesRead;

while ((bytesRead = read(fd, buffer, sizeof(buffer) - 1)) > 0) {

buffer[bytesRead] = '\0';

printf("%s", buffer);

}

if (bytesRead == -1) {

perror("读取文件失败");

}

close(fd);

}

}

*// 修改文件权限*

void change\_permissions(const char \*filename, mode\_t mode) {

if (chmod(filename, mode) == -1) {

perror("修改文件权限失败");

} else {

printf("文件权限修改成功: %s\n", filename);

}

}

*// 查看文件权限*

void view\_permissions(const char \*filename) {

struct stat fileStat;

if (stat(filename, &fileStat) == -1) {

perror("获取文件状态失败");

} else {

printf("文件 %s 的权限: \n", filename);

printf((S\_ISDIR(fileStat.st\_mode)) ? "目录" : "文件");

printf("\n");

printf("用户权限：");

printf((fileStat.st\_mode & S\_IRUSR) ? " 读取" : " -");

printf((fileStat.st\_mode & S\_IWUSR) ? " 写入" : " -");

printf((fileStat.st\_mode & S\_IXUSR) ? " 执行" : " -");

printf("\n");

printf("组权限：");

printf((fileStat.st\_mode & S\_IRGRP) ? " 读取" : " -");

printf((fileStat.st\_mode & S\_IWGRP) ? " 写入" : " -");

printf((fileStat.st\_mode & S\_IXGRP) ? " 执行" : " -");

printf("\n");

printf("其它人权限：");

printf((fileStat.st\_mode & S\_IROTH) ? " 读取" : " -");

printf((fileStat.st\_mode & S\_IWOTH) ? " 写入" : " -");

printf((fileStat.st\_mode & S\_IXOTH) ? " 执行" : " -");

printf("\n");

}

}

*// 主函数，提供用户菜单进行交互*

int main() {

int choice;

char filename[256];

char content[1024];

mode\_t mode;

while (1) {

printf("\n文件工具菜单:\n");

printf("1. 创建新文件\n");

printf("2. 写文件\n");

printf("3. 读文件\n");

printf("4. 修改文件权限\n");

printf("5. 查看文件权限\n");

printf("6. 退出\n");

printf("请输入您的选择: ");

scanf("%d", &choice);

getchar(); *// 消耗缓冲区中的换行符*

switch (choice) {

case 1:

printf("请输入文件名: ");

fgets(filename, sizeof(filename), stdin);

filename[strcspn(filename, "\n")] = '\0'; *// 去除换行符*

create\_file(filename);

break;

case 2:

printf("请输入文件名: ");

fgets(filename, sizeof(filename), stdin);

filename[strcspn(filename, "\n")] = '\0';

printf("请输入内容: ");

fgets(content, sizeof(content), stdin);

content[strcspn(content, "\n")] = '\0';

write\_file(filename, content);

break;

case 3:

printf("请输入文件名: ");

fgets(filename, sizeof(filename), stdin);

filename[strcspn(filename, "\n")] = '\0';

read\_file(filename);

break;

case 4:

printf("请输入文件名: ");

fgets(filename, sizeof(filename), stdin);

filename[strcspn(filename, "\n")] = '\0';

printf("请输入新的权限（八进制，例如 0644 表示用户可读写，组和其他人可读）: ");

scanf("%o", &mode);

getchar(); *// 消耗缓冲区中的换行符*

change\_permissions(filename, mode);

break;

case 5:

printf("请输入文件名: ");

fgets(filename, sizeof(filename), stdin);

filename[strcspn(filename, "\n")] = '\0';

view\_permissions(filename);

break;

case 6:

printf("退出...\n");

exit(0);

default:

printf("无效选择，请重试。\n");

}

}

return 0;

}

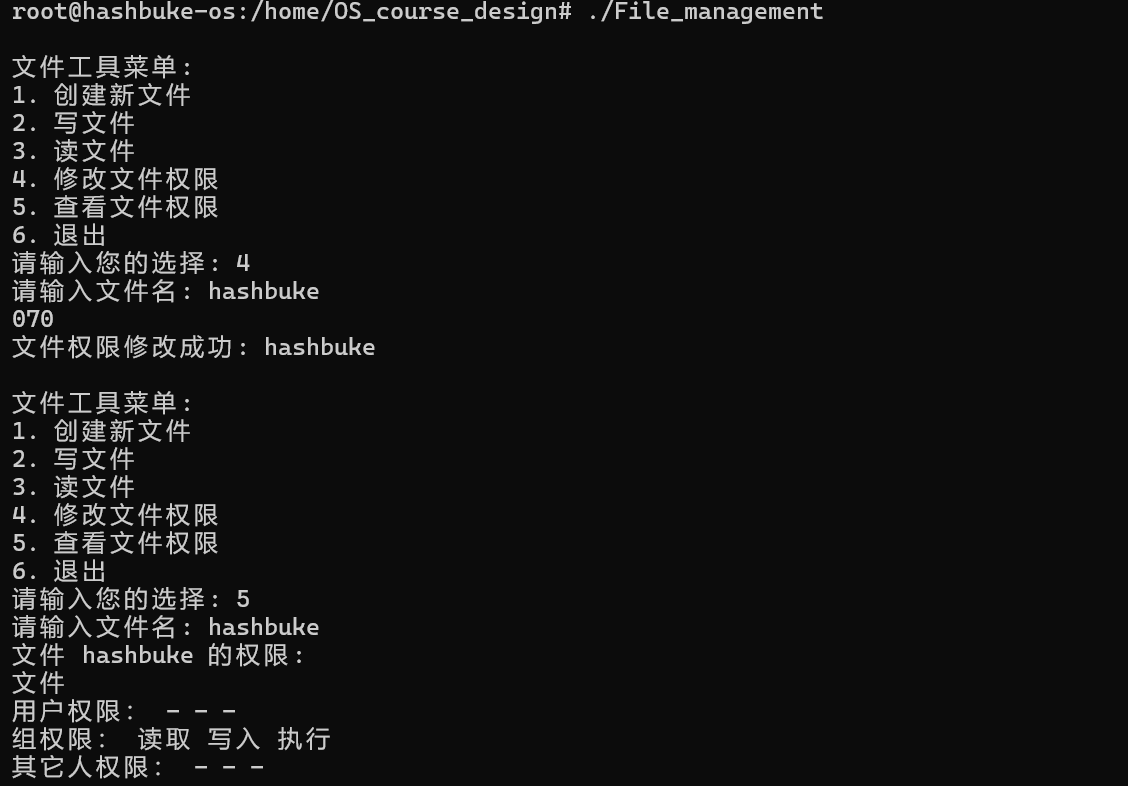
### 4.4 编译、运行过程

**编译程序**：打开终端或命令提示符。导航到包含源代码文件的目录。使用以下命令编译程序：

gcc -o File\_management File\_management.c

**运行程序**：在终端或命令提示符中运行编译后的可执行文件：

./File\_management

**

程序将启动并显示文件工具菜单，用户可以选择创建新文件、写文件、读文件、修改文件权限、查看文件权限或退出程序。根据用户的选择，程序将执行相应的文件操作。

### 4.5 问题以及解决方式

**遇到的问题与解决方案**

在文件系统操作中，权限设置是一个关键点。我决定使用open函数来创建文件，并且指定了权限为0644，这意味着文件所有者有读写权限，而其他人只有读取权限。这种权限设置既保证了安全性，也确保了文件的可访问性。然而，在实际操作中，我发现有时文件创建会失败，这可能是因为权限问题或路径不正确。为此，我增加了错误处理，使用perror函数来输出具体的错误信息，从而帮助用户了解问题所在。

在实现写入文件功能时，我面临的主要问题是如何高效地处理用户输入的内容并写入文件。我选择了使用O\_WRONLY | O\_APPEND标志来打开文件，以确保内容被追加到文件末尾而不是覆盖原有内容。这一设计决策的背后，是我对数据完整性的重视，特别是在日志文件等场景中，数据的连续性和完整性尤为重要。

读取文件的功能看似简单，但为了保证用户体验，我设计了一个循环读取机制，通过read函数每次读取固定大小的缓冲区，确保即使是大文件也能顺利读取并输出到屏幕上。在实际测试中，我发现当文件非常大时，单次读取可能无法处理所有数据，因此通过循环读取解决了这个问题。

文件权限的修改和查看功能涉及到系统调用chmod和stat。在实现过程中，我体会到Unix权限模型的精妙之处。通过对这些系统调用的使用，我不仅加深了对权限模型的理解，还掌握了如何在程序中灵活地处理和展示权限信息。在用户输入新的权限值时，我选择使用八进制来表示权限，这是Unix系统中惯用的表示方法，但为了用户友好性，我增加了详细的提示，帮助用户理解如何输入正确的权限值。

**收获体会**

整个实验的过程中，我最大的收获是对文件系统操作的深入理解和对错误处理的重要性的认识。每一个小细节的处理，都是为了提升用户体验和程序的健壮性。我也深刻体会到，一个好的程序不仅要实现功能，还要易于使用和维护。

## 5.内容五

### 设计思想

该实验的设计目的是通过创建一个自定义的系统调用来增强对Linux内核的理解和实践。实验包含以下几个部分：

**Python脚本**：用于获取内核中系统调用表的地址。

**内核模块**：实现一个自定义系统调用，并将其注入系统调用表。

**用户空间测试程序**：调用自定义的系统调用，验证其功能。

**获取系统调用表地址**：使用Python脚本读取/proc/kallsyms文件并过滤出系统调用表的地址。

**内核模块**：编写一个内核模块，替换系统调用表中的一个系统调用为自定义的系统调用。

使用clear\_cr0和setback\_cr0函数修改cr0寄存器，以允许对系统调用表的写操作。

**用户空间测试程序**：编写一个简单的C程序，使用syscall函数调用自定义的系统调用，并验证其是否成功执行。

### 5.2 数据结构

该实验没有复杂的数据结构，主要使用标准的内核模块编程接口和系统调用接口。

### 5.3 源代码

#### 获取系统调用表地址的Python脚本 get\_table.py

import subprocess

def run\_command():

try:

*# Running the command using subprocess*

result = subprocess.run(['cat', '/proc/kallsyms'], capture\_output=True, text=True)

*# Checking if the command was successful*

if result.returncode == 0:

*# Filtering the result to find lines containing 'sys\_call\_table'*

lines = result.stdout.splitlines()

filtered\_lines = [line for line in lines if 'sys\_call\_table' in line]

*# Printing the filtered lines*

for line in filtered\_lines:

print(line)

else:

print(f"Command failed with return code {result.returncode}")

except Exception as e:

print(f"An error occurred: {e}")

if \_\_name\_\_ == "\_\_main\_\_":

run\_command()

#### 内核模块 hashbuke\_syscall.c

#include <linux/kernel.h>

#include <linux/init.h>

#include <linux/module.h>

#include <linux/unistd.h>

#include <linux/sched.h>

MODULE\_LICENSE("Dual BSD/GPL");

#define SYS\_CALL\_TABLE\_ADDRESS 0xffffffffa9c00300 *// 确认该地址正确*

#define NUM 400 *// 系统调用号为400*

int orig\_cr0; *// 用来存储cr0寄存器原来的值*

unsigned long \*sys\_call\_table\_my = 0;

static int(\*anything\_saved)(void); *// 定义一个函数指针，用来保存一个系统调用*

*// 使cr0寄存器的第17位设置为0（内核空间可写）*

static int clear\_cr0(void)

{

unsigned int cr0 = 0;

unsigned int ret;

asm volatile("movq %%cr0, %%rax":"=a"(cr0));

ret = cr0;

cr0 &= 0xfffffffffffeffff; *// 将cr0变量值中的第17位清0*

asm volatile("movq %%rax, %%cr0"::"a"(cr0));

return ret;

}

*// 使cr0寄存器设置为内核不可写*

static void setback\_cr0(int val)

{

asm volatile("movq %%rax, %%cr0"::"a"(val));

}

*// 定义自己的系统调用*

asmlinkage long sys\_mycall(void)

{

printk(KERN\_INFO "By: HashBuke - sys\_mycall called\n");

return 1;

}

*// 初始化模块*

static int \_\_init call\_init(void)

{

sys\_call\_table\_my = (unsigned long\*)(SYS\_CALL\_TABLE\_ADDRESS);

printk(KERN\_INFO "HashBuke: Module loaded\n");

printk(KERN\_INFO "HashBuke: System call table address: %p\n", sys\_call\_table\_my);

anything\_saved = (int(\*)(void))(sys\_call\_table\_my[NUM]);

orig\_cr0 = clear\_cr0();

sys\_call\_table\_my[NUM] = (unsigned long)&sys\_mycall;

setback\_cr0(orig\_cr0);

printk(KERN\_INFO "HashBuke: System call %d replaced\n", NUM);

return 0;

}

*// 退出模块*

static void \_\_exit call\_exit(void)

{

orig\_cr0 = clear\_cr0();

sys\_call\_table\_my[NUM] = (unsigned long)anything\_saved;

setback\_cr0(orig\_cr0);

printk(KERN\_INFO "HashBuke: Module unloaded\n");

}

module\_init(call\_init);

module\_exit(call\_exit);

MODULE\_LICENSE("GPL");

MODULE\_DESCRIPTION("一个简单的 Linux 字符驱动程序 HashBuke");

MODULE\_VERSION("1.0");

#### 用户空间测试程序 test\_hashbuke.c

#include <stdio.h>

#include <unistd.h>

#include <sys/syscall.h>

#define \_\_NR\_hashbuke 400 *// 使用您的实际系统调用号*

int main()

{

long res = syscall(\_\_NR\_hashbuke);

if (res == -1)

{

printf("系统调用失败！\n");

}

else

{

printf("系统调用成功！\n");

}

return 0;

}

#### Makefile

*# Makefile for compiling the hashbuke\_syscall kernel module*

*# 设置模块名*

obj-m += hashbuke\_syscall.o

*# 默认目标，编译模块*

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

*# 清理目标，删除编译生成的文件*

clean:

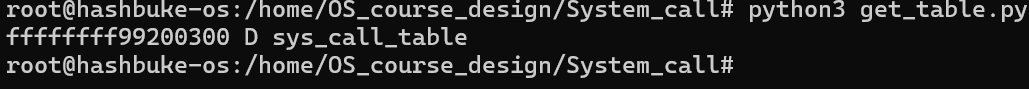
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

### 5.4 编译、运行过程

**获取系统调用表地址**：运行get\_table.py脚本获取系统调用表的地址：

python3 get\_table.py

确认地址正确，并在hashbuke\_syscall.c文件中更新SYS\_CALL\_TABLE\_ADDRESS宏定义。



**编译内核模块**：

打开终端，导航到包含hashbuke\_syscall.c和Makefile的目录。

使用以下命令编译内核模块：

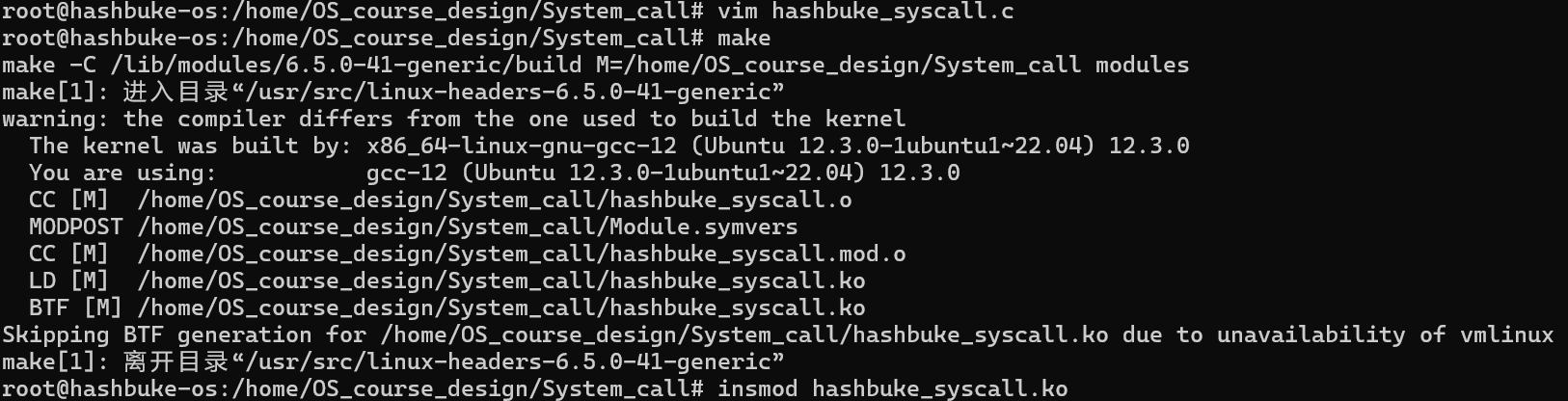
make all

**加载和卸载内核模块**：加载模块：

sudo insmod hashbuke\_syscall.ko

卸载模块：

sudo rmmod hashbuke\_syscall



**测试自定义系统调用**：

编译用户空间测试程序：

gcc -o test\_hashbuke test\_hashbuke.c

运行测试程序：

./test\_hashbuke

程序将调用自定义的系统调用并验证其是否成功执行，输出相应的结果。



### 5.5 问题以及解决方式

**遇到的问题与解决方案**

**1. 获取系统调用表地址**

**问题：** 需要从/proc/kallsyms文件中获取系统调用表的地址。

**解决方法：** 我使用Python的subprocess模块运行cat /proc/kallsyms命令，并过滤包含sys\_call\_table的行来获取地址。

**2. 编写内核模块**

**问题：** 修改系统调用表的条目，确保系统稳定性和安全性。

**解决方法：** 在hashbuke\_syscall.c中，我定义了指向系统调用表的指针，通过修改cr0寄存器临时允许内核空间写操作，然后将新的系统调用函数地址写入系统调用表。在卸载模块时，恢复原来的系统调用地址和cr0寄存器。

**3. 用户空间测试**

**问题：** 验证内核模块的工作，需要在用户空间调用新的系统调用。

**解决方法：** 我编写了test\_hashbuke.c程序，通过调用syscall函数并传入自定义的系统调用号来测试内核模块的功能，并根据返回值输出结果。

**4. 编写Makefile**

**问题：** 简化内核模块的编译和清理过程。

**解决方法：** 编写Makefile文件，设置内核源代码路径和模块名称，确保模块可以顺利编译和清理。

**收获体会**

通过这个实验，我对Linux内核的系统调用和模块开发有了更深入的理解。内核编程需要扎实的理论知识和细致的调试能力，安全性和稳定性至关重要。每一步操作都需要谨慎验证，以避免系统崩溃。

# 五、课程设计总结

在这门课程设计中，我经历了从理论到实践的完整过程，通过一系列的实验，深入理解了Linux操作系统的各项原理。每个实验都带来了新的挑战和收获，让我对系统编程有了更深刻的认识。

搭建开发环境是一个不小的挑战。我需要配置内核源代码，并确保所有必要的开发工具正确安装。虽然看似简单，但过程中遇到了一些版本兼容性的问题。解决这些问题不仅让我掌握了更多的工具使用方法，也让我对环境配置的重要性有了更深的体会。

编写获取系统调用表地址的脚本让我深入了解内核数据结构。使用Python的subprocess模块运行系统命令，并过滤获取目标数据，这个过程让我感受到脚本语言在系统管理中的强大作用。

内核模块的编写是整个课程设计中最具挑战性的部分。修改系统调用表条目，需要非常谨慎地处理cr0寄存器的读写权限。我不断调试和查阅资料，终于让模块成功加载并运行。看到自己的模块能够正确工作，那种成就感无以言表，这是对我编程能力的一次巨大提升。

实现自定义系统调用，进一步让我理解了内核的工作机制。通过asmlinkage关键字定义新的系统调用函数，并将其插入系统调用表中，这个过程不仅考验了我的编程技巧，更让我认识到内核开发的严谨性和复杂性。

用户空间的测试程序是验证内核模块正确性的关键。编写测试程序并调用自定义系统调用，让我切身体会到用户态与内核态的交互。测试成功时，我的努力得到了验证，内心的满足感油然而生。

编写Makefile虽然是最基础的工作之一，但它大大简化了内核模块的编译和清理过程。这个实验让我认识到自动化工具在开发中的重要性，提高了我的开发效率，也让我在实际操作中感受到规范化和自动化带来的便利。

这次课程设计不仅是一次知识的积累，更是一次实践能力的锻炼。从初期的环境配置，到最终的系统调用实现和测试，每一个实验都让我学到了宝贵的经验。通过解决一个个实际问题，我对Linux内核有了更深刻的理解，也增强了自己在系统编程方面的自信心。