华 中 科 技 大 学

网络空间安全学院

本科：《操作系统原理实验》

实验报告

题目：内存管理和设备管理实验

姓　　名 _____

班　　级 _____

学　　号 _____

联系方式 _____

分　　数 _____

评 分 人 _____

2024 年 12 月 10 日

报告要求

1. 报告不可以抄袭，发现雷同者记为 **0** 分。

2. 报告中不可以只粘贴大段代码，应是文字与图、表结合的，需要说明流程的时候，也应该用流程图或者伪代码来说明；如果发现有大段代码粘贴者，报告需重写。

3. 报告格式严格按照要求规范，并作为评分标准。

课程目标评价标准

课程目标	评价环节	评价标准		
		高于预期（优良）	达到预期（及格）	低于预期(不及格)
目标 1.能够依据实际工程问题中软硬件的约束，应用操作系统的基础理论、抽象和分层设计思想，对特定操作系统进行评价、分析，并能进行合理的优化和裁减。具备设计、实现、开发小型或简化的操作系统的能力，并能体现一定程度的创新性。能熟练应用操作系统的开发和调试技术和工具。	当堂验收	能很合理地设计程序结构、算法和主要数据结构；完成 80%以上的预定功能。	能比较合理地设计程序结构、算法和关键的数据结构；完成 80%以上的预定功能。	程序结构、算法和主要数据结构设计不很合理；仅完成不到50%的预定功能。
	实验报告	预定功能全部完成；截止日期前提交；源代码完整，且可编译；源代码注释清晰可读。	预定功能 80%以上完成；截止日期前提交；源代码完整，且可编译；源代码注释比较清晰可读。	预定功能仅完成不到 50%以上完成；截止日期以后提交；内容单薄；图文排版杂乱无章；源代码缺失或无法编译；源代码无注释或注释不清晰。
目标 2.具备操作系统国产化和自主创新意识，掌握国产操作系统，例如麒麟操作系统，鸿蒙操作系统的应用和开发环境，支持和推广国产操作系统，积极建设国产操作系统的技术生态。	当堂验收	完全使用国产操作系统；代码规范；能正确回答老师的绝大部分提问。	完全使用国产操作系统；代码比较规范；能正确回答老师的半数以上提问。	仅部分功能使用国产操作系统；代码不够规范，注释缺乏；仅能正确回答老师的极少数提问。
	实验报告	技术框架非常合理高效；符合用户硬件环境和约束参数。报告内容充实；图文排版规范；使用国产操作系统完成全部实验。	技术框架基本合理高效；符合用户硬件环境和约束参数。报告内容基本充实；图文排版基本规范；大部分实验过程使用国产操作系统。	技术框架不合理，代码臃肿；不完全符合用户硬件环境和约束参数。报告内容单薄；图文排版杂乱无章；仅小部分实验过程使用国产操作系统，或完全没用。

报告评分表

评 分 项 目		满分	得分	评分标准
内存管理和设备管理	总体设计（目标 1）	15		15-11：能够给出明确需求，系统功能完整、正确和适当。 10-6：能够给出需求，但不够完整，能够阐述系统的设计，但不够完整、恰当和准确。 5-0：需求不够明确，系统设计不够完整、正确和恰当。
	详细设计（目标 1）	15		15-11：函数和数据结构描述完整，关系清晰，流程设计正确规范。 10-6：函数和数据结构描述基本完整，流程设计基本正确。 5-0：函数和数据结构描述不完整，流程设计有错误。
	代码实现（目标 1，目标 2）	10		10-8：代码能够实现设计的功能要求，考虑错误处理和边界条件。有充分的注释，代码格式规范。 7-5：代码能够实现基本的功能要求，但可能缺少错误处理和边界条件的考虑。关键部分有简单注释，代码格式较为规范。 4-0：代码未能完全实现功能要求，缺少错误处理和边界条件的考虑。注释不足，代码格式不够规范。
	测试及结果分析（目标 1，目标 2）	20		20-14：测试方法科学、完整，结果分析准确完备。 13-8：测试方法描述基本正确、完整，结果分析准确完备。 7-0：测试仅针对数据集的通过性进行描述。
	问题描述及解决方案（目标 1）	10		10-8：遇到的问题及解决方案真实具体 7-5：遇到描述不够详细，解决方案不够具体 4-0：没有写什么内容。
感想（含思政）（目标 2）		10		10-8：感想真实具体。 7-5：感想比较空洞。 4-0：没有写什么感想。
遇到的问题和解决方案，及意见和建议（目标 1）		10		10-8：意见和建议有的放矢。 7-5：意见和建议不够明确。 4-0：没有写什么内容。
文档格式（段落、行间距、缩进、图表、编号等）（目标 1）		10		基本要求：目录、标题、行间距、缩进、正文字体字号按照模板要求执行，图、表清晰且有标号。 10-8：格式规范美观，满足要求。 7-5：基本满足要求。 4-0：格式较为混乱。
总分		**100**		
教师签名			日期	

目　　录

1 实验概述

1.1 内存管理和设备管理实验

实验三：第 7 章 内存管理、第 8 章设备管理

1.2 实验目的

(1) 理解页面淘汰算法原理，编写程序演示页面淘汰算法；

(2) 验证 Linux 虚拟地址转化为物理地址的机制；

(3) 掌握 Linux 驱动程序的编写流程和基本编程技巧。

1.3 实验环境

操作平台：VMware Workstation Pro 16

操作系统：优麒麟 24.04，Linux 6.6.22，Window 11

编辑工具：gedit 46.2

1.4 实验内容

(1) Windows/Linux 模拟实现 OPT,FIFO,LRU 等淘汰算法；

(2) Linux 下利用/proc/pid/pagemap 计算变量或函数虚拟地址对应的物理地址等信息；

(3) 在 Linux 平台编写一个字符设备的驱动程序和测试用的应用程序；

(4) 在 Linux 平台编写一个字符设备的驱动程序和测试用的应用程序。驱动程序的功能：内部维护一个 32 字节的缓冲区。

1.5 实验要求

任务 1、2 任选一，3、4 任选 1。

现场检查：任意完成一个即可。

我将四个任务均已完成，下面我将更加具体展开阐述完成过程和结果。

2 实验过程

2.1 总体设计

2.1.1 任务 1 Windows 下模拟 OPT 、FIFO 和 LRU 算法

本任务旨在通过编写程序模拟 OPT、FIFO 和 LRU 三种页面替换算法，以展示不同算法在内存管理中的应用。首先，需要确定程序的整体框架，包括数据结构的定义、函数的声明以及主函数的逻辑。然后，针对每种算法，设计相应的实现逻辑，如 LRU 使用计时器数组记录页框使用情况，FIFO 借助队列跟踪页面入队出队，OPT 通过映射和堆栈记录页号未来访问顺序。最后，通过主函数实现用户交互，允许用户输入相关参数，调用不同算法并输出结果，以便直观地比较各算法的性能。

2.1.2 任务 2 Linux 下计算虚拟地址对应的物理地址

此任务主要是实现一个能够计算给定虚拟地址对应的物理地址的函数。整体设计围绕着如何利用 Linux 系统的/proc 文件系统中的 pagemap 文件来获取内存映射信息展开。先确定要读取的进程的相关信息，然后通过函数 fun 实现具体的读取和计算逻辑，包括获取系统页面大小、计算虚拟页索引和偏移量、读取并解析物理页索引以及最终计算物理地址等步骤。在主函数中，通过创建子进程并多次调用 fun 函数，传入不同类型的地址，来验证函数的正确性和通用性。

2.1.3 任务 3 Linux 字符设备驱动实例

该任务的总体设计是创建一个 Linux 字符设备驱动程序，并编写相应的测试程序。首先，在驱动程序设计方面，需要定义相关的函数，如设备打开、读取和写入函数，以实现对用户输入的简单数学算式的处理和结果返回。在测试程序设计上，要实现打开字符设备、接收用户输入的数学表达式、将其写入设备并读取计算结果，最后将结果输出显示，以此来验证驱动程序的功能是否正确实现，以及用户空间与内核空间之间的交互是否正常。

2.2 详细设计

2.2.1 任务 1 Windows 下模拟 OPT 、FIFO 和 LRU 算法

任务 1 的核心部分在于**页面替换算法**的实现。

- **LRU (Least Recently Used)**

使用一个计时器数组 timer 来记录每个页框的使用情况。在访问时，如果命中，则重置对应的计时器；如果未命中，根据计时器找到最久未使用的页进行替换。更新缺页次数和缺页率，然后输出访问结果。

- **FIFO (First In First Out)**

我们可以使用一个队列 fifoQueue 来跟踪页框中页面的入队和出队情况。对于每次访问，检查该页是否在页框中：如果命中，继续访问；如果未命中，增加缺页次数，若页框已满，则从队列中取出最旧的页面并替换；如果未满，直接插入新页面。更新每种情况后，输出访问次数、缺页次数和缺页率。

- **OPT (Optimal Page Replacement)**

创建一个映射 ms，键为页号，值为堆栈，堆栈中存储该页的未来访问顺序。使用逆序遍历 orderArr，为每个页号记录下其将来的访问时间。在主动访问时，若未命中，则查找页框中是否存在；若不存在，再找出在未来最长时间不被访问的页进行替换。最后更新缺页次数和缺页率，并输出结果。

下面依次是 LRU、FIFO 和 OPT 算法的具体代码内容。

```
// LRU算法实现
void LRU() {
    int timer[pageFrameCnt];
    memset(timer, 0, sizeof(timer));
    for (int i = 0; i < orderCnt; ++i) {
        auto pageNo = orderArr[i] / pageSize; // 页号
        auto offset = orderArr[i] % pageSize; // 页内偏移
        int j;

        // 命中
        for (j = 0; j < cnt; ++j) {
            if (pageIdx[j] == pageNo) {
                timer[j] = 0; // 更新使用时间
                break;
            }
        }
        // 未命中
        if (j == cnt) {
            ++LRUpageMissCnt;
            if (cnt == pageFrameCnt) {
                auto maxT = 0;
                for (int k = 0; k < pageFrameCnt; ++k) {
                    if (timer[k] > timer[maxT])
                        maxT = k;
                }
                copyPage(maxT, pageNo); // 替换页
                timer[maxT] = 0; // 重置时间
            } else {
                copyPage(cnt, pageNo); // 直接添加新页面
                ++cnt;
            }
        }
        for (int j = 0; j < cnt; ++j)
            ++timer[j];
    }
    cout << "visit times: " << orderCnt << " miss times: " << LRUpageMissCnt
        << " LRU miss rate: " << float(LRUpageMissCnt) / orderCnt * 100 << "%" << endl;
}
```

```cpp
// FIFO算法实现
void FIFO() {
    queue<int> fifoQueue; // 使用队列来实现 FIFO 页替换算法
    for (int i = 0; i < orderCnt; ++i) {
        auto pageNo = orderArr[i] / pageSize; // 页号
        int j;

        // 查找当前页号在页框中
        for (j = 0; j < cnt; ++j) {
            if (pageIdx[j] == pageNo) {
                break; // 命中
            }
        }

        // 如果未命中
        if (j == cnt) {
            ++FIFOpagMissCnt; // 缺页次数+1

            // 如果页框已满
            if (cnt == pageFrameCnt) {
                int oldPage = fifoQueue.front(); // 获取最旧的页面
                fifoQueue.pop(); // 移除最旧的页面

                // 查找该页面在页框中的索引并替换
                for (int k = 0; k < cnt; ++k) {
                    if (pageIdx[k] == oldPage) {
                        copyPage(k, pageNo); // 替换为新页面
                        break;
                    }
                }
            } else {
                copyPage(cnt, pageNo); // 直接添加新页面
                ++cnt;
            }
            fifoQueue.push(pageNo); // 将新页面入队
        }
    }
    cout << "visit times: " << orderCnt << " miss times: " << FIFOpagMissCnt
        << " FIFO miss rate: " << float(FIFOpagMissCnt) / orderCnt * 100 << "%" << endl;
}
```

```cpp
// OPT算法实现
void OPT() {
    map<int, stack<int>> ms;
    for (int i = orderCnt - 1; i >= 0; --i) {
        auto pageNo = orderArr[i] / pageSize;
        if (ms.count(pageNo) == 0) {
            stack<int> tmp;
            tmp.push(i);
            ms.insert(pair<int, stack<int>>(pageNo, tmp));
        } else {
            ms.at(pageNo).push(i);
        }
    }
    for (int i = 0; i < orderCnt; ++i) {
        auto pageNo = orderArr[i] / pageSize;
        int j;
        if (ms.at(pageNo).size())
            ms.at(pageNo).pop();
        for (j = 0; j < cnt; ++j) {
            if (pageIdx[j] == pageNo) {
                break; // 命中
            }
        }
        if (j == cnt) {
            ++OPTpageMissCnt; // 缺页次数+1
            if (cnt == pageFrameCnt) {
                auto maxT = 0;
                for (int k = 0; k < pageFrameCnt; ++k) {
                    if (ms.at(pageIdx[k]).size() == 0) {
                        maxT = k; // 找到不再使用的页
                        break;
                    } else if (ms.at(pageIdx[k]).top() > ms.at(pageIdx[maxT]).top()) {
                        maxT = k; // 找到最长时间不使用的页
                    }
                }
                copyPage(maxT, pageNo); // 替换页
            } else {
                copyPage(cnt, pageNo); // 添加新页
                ++cnt;
            }
        }
    }
```

2.2.2 任务 2 Linux 下计算虚拟地址对应的物理地址

任务 2 的核心部分在于**函数 fun** 的实现。

此函数的主要功能是通过读取特定进程的 **/proc/<pid>/pagemap** 文件来获取给定虚拟地址对应的物理地址。使用 getpagesize() 获取系统的页面大小，然后通过虚拟地址计算虚拟页索引和偏移量。打开对应进程的 pagemap 文件，通过 lseek 定位到正确的偏移处读取数据。解析出物理页索引，并计算出最终的物理地址。最后通过 printf 输出相关信息。

```
// 函数 fun，用于获取虚拟地址对应的物理地址
void fun(char* str, UL pid, UL viraddress, UL* phyaddress)
{
    U64 temp = 0;                           // 用于存储读取的物理页信息
    int pageSize = getpagesize();           // 获取系统页面大小
    UL vir_pageIndex = viraddress / pageSize; // 计算虚拟页索引
    UL vir_offset = vir_pageIndex * sizeof(U64); // 计算虚拟页偏移量
    UL page_offset = viraddress % pageSize; // 虚拟地址在页面中的偏移量

    sprintf(buf, "%s%lu%s", "/proc/", pid, "/pagemap"); // 构建 pagemap 文件路径
    int fd = open(buf, O_RDONLY);                       // 以只读方式打开 pagemap 文件
    lseek(fd, vir_offset, SEEK_SET);                     // 将文件指针移动到虚拟页的偏移量
    read(fd, &temp, sizeof(U64));                        // 读取对应项的值
    U64 phy_pageIndex = (((U64)1 << 55) - 1) & temp;    // 提取物理页号
    *phyaddress = (phy_pageIndex * pageSize) + page_offset; // 计算物理地址

    // 输出信息
    printf("<%s>  进程ID = %lu, 虚拟地址 = 0x%lx, 页码 = %lu, 物理页框架号 = %lu, 物理地址 = 0x%lx\n",
           str, pid, viraddress, vir_pageIndex, phy_pageIndex, *phyaddress);

    sleep(1); // 暂停1秒
    return;
}
```

2.2.3 任务 3 Linux 字符设备驱动实例

该任务我实现了一个字符设备驱动程序，该程序能够接收用户输入的简单数学算式（如加法和减法），并将相应的运算结果返回给用户。通过这个实验，我们将深入理解 Linux 设备驱动的结构，字符设备的操作以及用户空间和内核空间之间的交互。

任务 3 关键部分是驱动程序的编写，其中核心函数是设备打开、设备读取、设备写入，下面我将对其代码的编写逻辑进行具体地讲解。

drv_open 函数是设备打开时的回调函数。它可以在打开设备时执行初始化操作，例如记录日志或分配资源。在此函数中，通过 MAJOR 和 MINOR 函数获取设备的主要和次要编号并记录到日志中。

```
static int drv_open(struct inode *nd, struct file *fp)
{
    int major;
    int minor;
    major = MAJOR(nd->i_rdev);
    minor = MINOR(nd->i_rdev);
    printk(KERN_EMERG "hello open, major = %d, minor = %d from hyt\n", major, minor);
    return 0;
}
```

drv_read 函数用于处理从设备读取数据的请求。这里逻辑是解析用户输入的数学表达式并计算结果。在读取之前，内核会将 hyt 中的内容解析为两个数字以及运算符（加法或减法），并执行相应的运算。

```
22 static ssize_t drv_read(struct file* fp, char __user* u, size_t sz, loff_t* loff)
23 {
24     char res[32];
25     int i = 0;
26     int num1 = 0;
27     int num2 = 0;
28     bool flag = true;
29     printk(KERN_EMERG "hello read from hyt.\n");
30     while (hyt[i] != '+' && hyt[i] != '-')
31     {
32         num1 = 10 * num1 + (hyt[i] - '0');
33         i++;
34     }
35     if (hyt[i] == '-') flag = false;
36     i++;
37     while (hyt[i] != '\0')
38     {
39         num2 = 10 * num2 + (hyt[i] - '0');
40         i++;
41     }
42     if (flag == true)
43     {
44         printk(KERN_EMERG "%d + %d = %d from hyt\n", num1, num2, num1 + num2);
45         sprintf(res, "%d + %d = %d\n", num1, num2, num1 + num2);
46         copy_to_user(u, res, strlen(res));
47     }
48     else
49     {
50         printk(KERN_EMERG "%d - %d = %d from hyt\n", num1, num2, num1 - num2);
51         sprintf(res, "%d - %d = %d\n", num1, num2, num1 - num2);
52         copy_to_user(u, res, strlen(res));
53     }
54     return 0;
55 }
```

drv_write 函数负责接收用户空间发送的数据。该函数将用户输入的字符串从用户空间复制到内核空间的 hyt 数组中。通过 copy_from_user 函数确保安全地将数据从用户空间传递至内核空间。

```
57 static ssize_t drv_write(struct file* fp, const char __user * u, size_t sz, loff_t* loff){
58     if (copy_from_user(hyt, u, sz)) {
59         return -EFAULT; // 处理错误
60     }
61     printk(KERN_EMERG "hello write from hyt.\n");
62     copy_from_user(hyt, u, sz);
63     return 0;
64 }
```

2.3 代码实现

2.3.1 任务 1 Windows 下模拟 OPT 、FIFO 和 LRU 算法

我在 Windows11 上使用 Visual Studio Code 编写 cpp 程序模拟 OPT、FIFO 和 LRU 的淘汰页面算法，下面我将对代码进行详细地讲解，页面替换算法实现的具体逻辑代码在上面已经进行了详细讲解，该部分便不再赘述。

（1） 初始化和定义

完成常量和全局变量的定义等，并初始化进程，用于随机生成进程数据，用于模拟访问的数据。以及初始化访问序列，根据用户选择的类型（随机或顺序），生成访问序列，随机方式是产生范围内的随机数，而顺序方式是直接生成从 0 到 orderCnt - 1 的数组。

```
#define MAX_NUM 10000
const int pageSize = 10;       // 页面大小
const int instrCnt = MAX_NUM; // 指令数目
const int pageFrameCnt = 3;    // 页框数

int orderCnt;                  // 访问次数
int cnt = 0;                   // 当前页框数
int LRUpageMissCnt = 0;        // LRU 缺页数
int OPTpageMissCnt = 0;        // OPT 缺页数
int FIFOpagMissCnt = 0;        // FIFO 缺页数
int op;                         // 选择页替换算法
int processArr[instrCnt];      // 进程数组
int pageFrame[pageFrameCnt][pageSize]; // 页框数组
int orderArr[MAX_NUM];         // 指令访问序列
int pageIdx[pageFrameCnt];     // 页框中存的页号
int arraykind;                 // 访问数组类型，0为随机，1为顺序，2为循环

// 初始化进程
inline void initProcess() {
    for (int i = 0; i < instrCnt; ++i) {
        processArr[i] = rand() % 1000; // 随机生成进程数据
    }
}

// 初始化访问序列
void initInstrOrder(int kind, int size) {
    orderCnt = size;
    if (kind == 1) {
        for (int i = 0; i < orderCnt; ++i) {
            orderArr[i] = rand() % size; // 随机访问
        }
    } else {
        for (int i = 0; i < orderCnt; ++i) {
            orderArr[i] = i; // 顺序访问
        }
    }
}
```

（2） 页面替换算法的实现

- **LRU (Least Recently Used)**

使用计时器数组记录每个页框的使用情况，依据最久未使用的策略来替换页面，计数缺页次数并最终输出缺页率。

- **FIFO (First In First Out)**

事先计算未来的访问顺序，并选择在未来最长时间不会被访问的页面进行替换。

- **OPT (Optimal Page Replacement)**

采用队列实现 FIFO 策略，移除最早加入的页面。类似于 LRU，但不考虑最近使用情况，

仅关注加入顺序。

（3） 主函数

主函数为程序的起点，负责初始化随机数种子。允许用户输入访问次数、选择访问种类（随机或顺序）、选择页面替换算法等。根据用户的选择，调用相应的页面替换算法，输出结果。实现了循环和用户交互，使用 while (1) 创建一个无限循环，直到用户选择退出 (4.quit)。根据用户的输入以及输出缺页统计信息，帮助用户理解页面替换算法的效果。

```
185     srand(time(nullptr));
186     printf("array size:\n");
187     cin >> orderCnt; // 用户输入访问次数
188     while (1) {
189         initProcess(); // 初始化进程
190         printf("select array kind:\n1.Random\n2.Sequence\n");
191         cin >> arraykind; // 选择访问方式
192         initInstrOrder(arraykind, orderCnt); // 初始化访问序列
193 
194         printf("1.OPT\n2.FIFO\n3.LRU\n4.quit\n");
195         cin >> op; // 选择页替换算法
196         if (op == 1) {
197             OPTpageMissCnt = 0; // 重置缺页计数
198             OPT(); // 执行 OPT 算法
199         } else if (op == 3) {
200             LRUpageMissCnt = 0; // 重置 LRU 缺页计数
201             LRU(); // 执行 LRU 算法
202         } else if (op == 2) {
203             FIFOpagMissCnt = 0; // 重置 FIFO 缺页计数
204             FIFO(); // 执行 FIFO 算法
205         } else if (op == 4) {
206             break; // 选择退出，结束程序
207         } else {
208             cout << "无效的选择，请重新输入。\n"; // 无效选择提示
209         }
210     }
```

（4）其他

输出页框状态 (showPageFrame)，该函数在后续可供调试时使用，输出当前页框的内容和状态。同时结束部分调用 system("pause") 确保程序在运行结束前保持控制台窗口打开，便于查看输出。最后 return 0; 表示程序正常结束

综上，该程序模拟了页面替换算法的工作原理，用户可以通过简单交互测试不同算法的性能。每种算法都有其独特的逻辑与实现方式，能够有效展示内存管理中的页面替换策略。

2.3.2 任务 2 Linux 下计算虚拟地址对应的物理地址

代码借助 Linux 系统下/proc 文件系统中 pagemap 文件所记录的内存映射信息，尝试探索不同类型变量在内存中的物理地址情况，展示了进程内存管理相关的一种获取物理地址的实现方式，但需要注意的是，这种方式依赖于特定的 Linux 内核机制以及系统配置，并且在不同

的 Linux 发行版等环境下可能存在一些细微差异和需要注意的地方。

（1） 数据定义

如下所示定义了一些全局变量，这里我将字符串变量定义为我的名字" helloworld ----hyt "。

```
#define UL unsigned long      // 定义 UL 为 unsigned long 类型
#define U64 uint64_t          // 定义 U64 为 uint64_t 类型

char buf[100];                // 定义一个字符数组，用于存储路径
const int a = 20;             // 定义一个全局常量 a
const char* name = "helloworld! ----hyt";  // 定义一个字符串常量 name
```

（2） 函数 fun

此函数的主要功能是通过读取特定进程的 **/proc/<pid>/pagemap** 文件来获取给定虚拟地址对应的物理地址。使用 getpagesize() 获取系统的页面大小，然后通过虚拟地址计算虚拟页索引和偏移量。打开对应进程的 pagemap 文件，通过 lseek 定位到正确的偏移处读取数据。解析出物理页索引，并计算出最终的物理地址。最后通过 printf 输出相关信息。

（3） main 函数

在主函数种我创建了一个局部变量 b 和常量 d，并定义了一个物理地址变量 phy。fork() 创建子进程，这个过程会生成一个新的进程。然后调用 fun 函数四次，传入不同的地址（局部变量、常量、全局常量和 puts 函数的地址），以获取这些地址对应的物理地址。

```
    int b = 1;                  // 定义一个局部变量 b
    const int d = 3;            // 定义一个局部常量 d
    UL phy = 4;                 // 定义一个物理地址变量 phy

    puts(name);                 // 输出字符串常量 name
    int pid = fork();           // 创建子进程

    // 调用 fun 函数，传入不同的虚拟地址
    fun("局部变量", getpid(), (UL)&b, &phy);
    fun("局部常量", getpid(), (UL)&d, &phy);
    fun("全局常量", getpid(), (UL)&a, &phy);
    fun("动态函数", getpid(), (UL)&puts, &phy);
```

2.3.3 任务 3 Linux 字符设备驱动实例

本节展示了实现字符设备驱动及其对应用户空间程序的代码。该实现不仅要满足基本的输入输出功能，还要展示内核模块的加载和用户空间的 API 调用。

（1） 驱动程序代码

此部分实现了一个基础的字符设备驱动，通过输入和输出简单的算术运算，增强了对 Linux 内核模块的理解。首先需要引入必要的 Linux 内核编程接口头文件，这些文件帮助我们进行模块编程、字符设备文件操作、字符串处理和用户内存访问。

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/string.h>
#include <linux/uaccess.h>

static struct cdev drv;
static dev_t ndev;
static char hyt[32];
```

由于核心函数函数在 2.2.3 进行了详细的讲解，此处便不再展示。然后 file_operations 结构体定义了与设备相关的操作（例如打开、读取和写入）。在这里，我们将相关函数指针指向先前定义的函数，实现对设备的操作。

```c
static struct file_operations drv_ops =
{
    .owner  =   THIS_MODULE,
    .open   =   drv_open,
    .read   =   drv_read,
    .write  =   drv_write,
};
```

接着是模块的初始化和卸载。hytDrive_init 函数是模块加载时被调用的初始化函数。它用于分配设备号、注册字符设备以及相关初始化操作。

```c
static int hytDrive_init(void)
{
    int ret;
    cdev_init(&drv, &drv_ops);
    ret = alloc_chrdev_region(&ndev, 0, 1, "hytDrive");
    if (ret < 0)
    {
        printk(KERN_EMERG " alloc_chrdev_region error.\n");
        return ret;
    }
    printk(KERN_EMERG "hytDrive_init(): major = %d, minor = %d\n", MAJOR(ndev), MINOR(ndev));
    ret = cdev_add(&drv, ndev, 1);
    if (ret < 0)
    {
        printk(KERN_EMERG " cdev_add error.\n");
        return ret;
    }
    return 0;
}

static void hytDrive_exit(void)
{
    printk("exit process from hyt!\n");
        cdev_del(&drv);
        unregister_chrdev_region(ndev, 1);
}
```

最后是模块宏，这些宏提供了模块的许可证、作者信息和描述，帮助识别和分类模块的来源和许可以及其功能。

```c
module_init(hytDrive_init);
module_exit(hytDrive_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("u202211915@hust.edu.cn");
MODULE_DESCRIPTION("hyt's Driver Demo");
```

（2） 测试程序代码

驱动程序编写完成后，接下来需要编写测试程序代码，首先是头文件与定义，然后主函数开始尝试打开字符设备，如果打开失败则输出错误信息并退出程序。这里使用了非阻塞模式（O_NDELAY），即使设备被其他操作忙封锁也不会阻塞当前进程。程序会提示用户输入一个数学表达式，要求格式为 number1+number2 或 number1-number2，并将用户输入写入设备，然后读取结果。写入操作将触发驱动程序中的 drv_write 函数，进行处理。再将从设备读取的计算结果打印到标准输出。

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#define CHAR_DEV_NAME "/dev/hytDrive"

char calc[32];
char res[32];
int main()
{
        int ret;
        int fd;
        extern int errno;
        fd = open(CHAR_DEV_NAME, O_RDWR | O_NDELAY);
        if(fd < 0)
        {
                printf("open failed! error %d\n", errno);
                return -1;
        }
        scanf("%s", calc);
        write(fd, calc, 32);
        read(fd, res, 32);
        printf("%s\n", res);
        close(fd);
        return 0;
}
```

（3） 编写 Makefile

该文件定义了编译内核模块和用户程序的规则。它首先检查是否在内核构建上下文中。如果不是，它将获取内核版本并指定构建目录，然后执行 make 命令构建模块并编译用户程序 test2。clean 目标用于清理构建过程中生成的文件。

```
ifneq ($(KERNELRELEASE),)
obj-m := hytDrive.o

else
KVER := $(shell uname -r)
KDIR := /lib/modules/$(KVER)/build
PWD  := $(shell pwd)
all:
        $(MAKE) -C $(KDIR) M=$(PWD) modules
        gcc -o test2 test2.c
clean:
        rm -rf .*.cmd *.o *.mod.c *.ko .tmp_versions modules.*  Module.*
        rm -rf test2
endif
```

3 测试与分析

3.1 系统测试及结果说明

3.1.1 任务 1 Windows 下模拟 OPT 、FIFO、LRU 算法

在 Windows 11 环境中编译并运行该程序时，观察到在访问序列的数量为 1000 且为随机情况时（左侧），三种算法的缺页率均较高，接近 90%。此时，OPT 算法的表现优于 FIFO 和 LRU。而在顺序访问的情况下（右侧），三者的表现几乎没有明显差异。

```
select array kind:
1.Random
2.Sequence
1
1.OPT
2.FIFO
3.LRU
4.quit
1
visit times: 1000 miss times: 864 OPT miss rate: 86.4%
```

```
select array kind:
1.Random
2.Sequence
2
1.OPT
2.FIFO
3.LRU
4.quit
1
visit times: 1000 miss times: 98 OPT miss rate: 9.8%
```

图 3 - 1 任务 1 模拟 OPT 算法结果截图

```
select array kind:
1.Random
2.Sequence
1
1.OPT
2.FIFO
3.LRU
4.quit
2
visit times: 1000 miss times: 971 FIFO miss rate: 97.1%
```

```
select array kind:
1.Random
2.Sequence
2
1.OPT
2.FIFO
3.LRU
4.quit
2
visit times: 1000 miss times: 100 FIFO miss rate: 10%
```

图 3 - 2 任务 1 模拟 FIFO 算法结果截图

```
select array kind:
1.Random
2.Sequence
1
1.OPT
2.FIFO
3.LRU
4.quit
3
visit times: 1000 miss times: 963 LRU miss rate: 96.3%
```

```
select array kind:
1.Random
2.Sequence
2
1.OPT
2.FIFO
3.LRU
4.quit
3
visit times: 1000 miss times: 100 LRU miss rate: 10%
```

图 3 - 3 任务 1 模拟 LRU 算法结果截图

3.1.2 任务 2 Linux 下计算虚拟地址对应的物理地址

由图 3-4 可以看到，局部变量和全局变量在父进程与子进程之间所映射的物理地址存在差异。然而，全局常量的物理地址是相同的，这表明子进程会重新创建变量，但仍然使用父进程中的常量。至于动态链接库，在主函数中调用 puts 成功输出了字符串 " helloworld ----hyt "，并且父进程和子进程调用的 puts 函数共享同一物理地址。

```
hyt@hyt915-VMware:~/Desktop/lab3$ gcc addr.c -o  addr
hyt@hyt915-VMware:~/Desktop/lab3$ ./addr
helloworld! ----hyt
<局部变量>  进程ID = 273335, 虚拟地址 = 0x7ffc0c699074, 页码 = 34355594905, 物理页框架号 = 0, 物理地址 = 0x74
<局部变量>  进程ID = 273336, 虚拟地址 = 0x7ffc0c699074, 页码 = 34355594905, 物理页框架号 = 0, 物理地址 = 0x74
<局部常量>  进程ID = 273335, 虚拟地址 = 0x7ffc0c699078, 页码 = 34355594905, 物理页框架号 = 0, 物理地址 = 0x78
<局部常量>  进程ID = 273336, 虚拟地址 = 0x7ffc0c699078, 页码 = 34355594905, 物理页框架号 = 0, 物理地址 = 0x78
<全局常量>  进程ID = 273335, 虚拟地址 = 0x55dbf3233008, 页码 = 23047647795, 物理页框架号 = 0, 物理地址 = 0x8
<全局常量>  进程ID = 273336, 虚拟地址 = 0x55dbf3233008, 页码 = 23047647795, 物理页框架号 = 0, 物理地址 = 0x8
<动态函数>  进程ID = 273335, 虚拟地址 = 0x7a7c31487bd0, 页码 = 32879350919, 物理页框架号 = 0, 物理地址 = 0xbd0
<动态函数>  进程ID = 273336, 虚拟地址 = 0x7a7c31487bd0, 页码 = 32879350919, 物理页框架号 = 0, 物理地址 = 0xbd0
hyt@hyt915-VMware:~/Desktop/lab3$
```

图 3 - 4 任务 2 结果截图

3.1.3 任务 3 Linux 字符设备驱动实例

首先对驱动代码进行 make 编译，再通过执行**$ sudo insmod hytDriver.ko** 来加载模块。

```
hyt@hyt915-VMware:~/Desktop/lab3/cdev_driver$ make
make -C /lib/modules/6.6.22/build M=/home/hyt/Desktop/lab3/cdev_driver modules
make[1]: 进入目录“/usr/src/linux-6.6.22”
make[1]: 离开目录“/usr/src/linux-6.6.22”
gcc -o test2 test2.c
hyt@hyt915-VMware:~/Desktop/lab3/cdev_driver$ sudo insmod hytDrive.ko
[sudo] hyt 的密码：
hyt@hyt915-VMware:~/Desktop/lab3/cdev_driver$
```

图 3 - 5 编译驱动代码结果截图

使用 dmesg 命令查看内核日志，确认模块是否正确加载及打印信息，下图表示成功加载。

```
hyt@hyt915-VMware:~/Desktop/lab3/cdev_driver$ sudo -s
root@hyt915-VMware:/home/hyt/Desktop/lab3/cdev_driver# dmesg | tail
[62201.038644] write test
[62201.038698] demo release
[62616.982403] workqueue: hub_event hogged CPU for >10000us 32 times, consider s
witching to WQ_UNBOUND
[63903.375353] workqueue: e1000_watchdog [e1000] hogged CPU for >10000us 32 time
s, consider switching to WQ_UNBOUND
[67025.869399] usb 2-2.1: reset full-speed USB device number 4 using uhci_hcd
[67028.381679] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control:
None
[67031.806334] audit: type=1400 audit(1734977246.904:140): apparmor="DENIED" ope
ration="capable" class="cap" profile="/usr/sbin/cupsd" pid=303120 comm="cupsd" c
apability=12  capname="net_admin"
[67035.174925] workqueue: pcpu_balance_workfn hogged CPU for >10000us 8 times, c
onsider switching to WQ_UNBOUND
[68791.811301] hytDrive_init(): major = 239, minor = 0
[68791.819365] audit: type=1400 audit(1734979006.911:141): apparmor="DENIED" ope
ration="open" class="file" profile="rsyslogd" name="/run/systemd/sessions/" pid=
822 comm=72733A6D61696E20513A526567 requested_mask="r" denied_mask="r" fsuid=102
 ouid=0
root@hyt915-VMware:/home/hyt/Desktop/lab3/cdev_driver#
```

图 3 - 6 加载模块和设置截图

然后编译并运行测试程序 test2.c，注意可以使用 mknod 命令来手动创建设备文件，。

```
root@hyt915-VMware:/home/hyt/Desktop/lab3/cdev_driver# mknod /dev/hytDrive c 239
 0
root@hyt915-VMware:/home/hyt/Desktop/lab3/cdev_driver# ./test2
5+3
5 + 3 = 8
```

图 3 - 7 运行应用程序结果截图

最后可以使用 rmmod 卸载模块。

```
root@hyt915-VMware:/home/hyt/Desktop/lab3/cdev_driver# sudo rmmod hytDrive
root@hyt915-VMware:/home/hyt/Desktop/lab3/cdev_driver# make clean
rm -rf .*.cmd *.o *.mod.c *.ko .tmp_versions modules.*  Module.*
rm -rf test2
```

图 3 - 8 卸载模块截图

本实验大部分命令均需要 root 权限，可以使用**$ sudo -s** 在该文件目录获取 root 用户权限。

3.2 遇到的问题及解决方法

（1）问题一：对比不同算法性能时，由于测试数据的随机性，有时候很难直观地看出各算法在不同场景下的优劣差异。

解决方法：我设计了多组具有不同特征的页面访问序列作为测试数据，包括顺序访问、随机访问，对每组数据分别运行 OPT、FIFO 和 LRU 算法，并记录它们的页面置换次数、算法执行时间等关键指标。

（2）问题二：对/proc/pid/pagemap 文件的格式和数据含义理解不够准确，按照初始设想的方式解析数据后，得到的物理地址结果总是不符合预期。

解决方法：查阅内核文档和资料，发现不同版本的 Linux 系统在某些细节上存在差异，针对实验所使用的 Linux 系统版本调整数据解析代码，确保正确提取处理信息。

（3）问题三：尝试运行 ./test2 时仍然遇到 "open failed! error 2" 的错误。

解决方法：alloc_chrdev_region() 函数分配设备号，但它不会自动创建 /dev/hytDrive 这一设备文件。需要手动创建字符设备文件通过**$ sudo mknod /dev/hytDrive c 239 0** 命令。

3.3 设计方案存在的不足

驱动程序的功能设计可以更加多样。

4 实验总结

4.1 实验感想

4.1.1 任务 1 Windows 下模拟 OPT 、FIFO、LRU 算法实验感想

在 Windows 环境中对 OPT、FIFO 和 LRU 页面淘汰算法的模拟实验，不仅加深了我对内存管理页面置换机制的理解，更让我意识到其在操作系统安全层面的重要意义。设计良好的交互界面以选择随机或顺序访问方式，使我能够从多维度观察算法性能。从安全角度看，合理的页面置换策略可防止恶意程序通过占用大量内存页面来实施拒绝服务攻击，保障系统的稳定运行。这种实为我今后在网络安全领域优化资源管理和防范内存相关攻击筑牢了基础，使我明白内存管理不仅关乎性能，更是系统安全的关键防线之一。

4.1.2 任务 2 Linux 下计算虚拟地址对应的物理地址实验感想

深入探究/proc/pid/pagemap 文件以计算物理地址的过程，任何微小的偏差都可能导致目标地址的误判，进而引发严重的安全后果，如非法访问敏感内存区域或执行恶意代码。对地址转换机制的透彻理解，使我意识到在操作系统安全中，内存地址的准确映射是防止缓冲区溢出、代码注入等攻击的关键。这如同在网络安全中，确保数据包的正确路由和解析，防止黑客利用地址欺骗等手段进行入侵。此实验不仅提升了我对 Linux 内存管理的认知，更强化了我在操作系统安全领域防范内存相关攻击的意识和能力。

4.1.3 任务 3 Linux 字符设备驱动实例实验感想

通过该 Linux 字符设备驱动开发任务，让我深刻认识到驱动程序作为操作系统与硬件交互桥梁的关键作用，同时也凸显了其在安全方面的敏感性。遵循内核编程规范编写驱动代码，尤其是在内存管理方面使用内核专用函数，类似于在网络安全设备的驱动开发中，严格遵循特定的安全编程标准，防止因内存操作不当引发的漏洞。这使我明白，在操作系统安全领域，驱动程序的稳定与安全是保障硬件设备正常运行且不被恶意利用的核心要素，为构建安全可靠的操作系统环境提供了坚实保障，也为我未来从事网络安全相关的底层开发工作积累了宝贵经验。

4.2 意见和建议

原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

已阅读并同意以下内容。

判定为不合格的一些情形：

（1）请人代做或冒名顶替者；

（2）替人做且不听劝告者；

（3）实验报告内容抄袭或雷同者；

（4）实验报告内容与实际实验内容不一致者；

（5）实验代码抄袭者。

作者签名：