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：内存管理实验

姓　　名　柳骁恩

班　　级　网安 2304 班

学　　号　U202314510

联系方式　13635763585

分　　数

评 分 人

2026 年 1 月 8 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	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 用户界面实验

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。

1.3 实验环境

Linux 6.8.12

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

1.5 实验要求

1,2 必做。寝室提前做完，老师机房检查和答疑。

2 实验过程

2.1 总体设计

2.1.1 程序演示页面淘汰算法任务

2.1.1.1 模块框图

系统模块框图如图 2-1 所示。

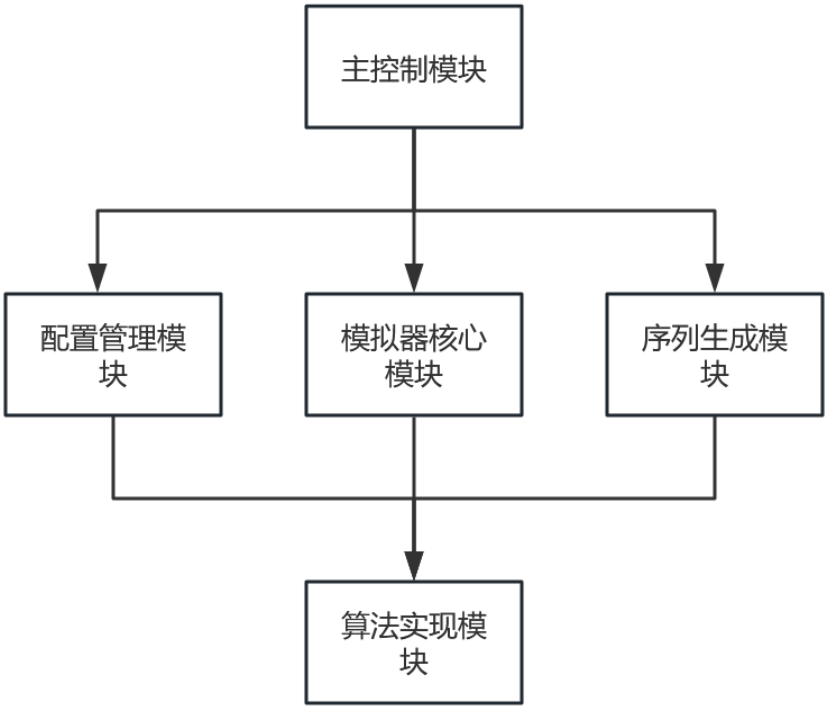

图 2-1 页面淘汰系统模块框图

2.1.1.2 模块功能说明

1.主控制模块 (main.c)

功能职责：

1）程序入口点

2）命令行参数解析和验证

3）模块初始化和协调

4）资源管理和清理

5）帮助信息显示

2.配置管理模块

（1）功能职责：

1）配置参数的存储和管理

2）配置参数的验证

3）提供配置访问接口

（2）关键数据结构：

```
15      // 配置参数
16      typedef struct {
17          int page_size;          // 页面大小（指令数）
18          int num_frames;         // 页框数量
19          int total_instructions; // 总指令数
20          int num_pages;          // 总页数
21          Algorithm algorithm;    // 使用的算法
22          int access_pattern;     // 访问模式：0-顺序，1-跳转，2-分支，3-循环，4-局部性随机
23          int *access_sequence;   // 访问序列
24          int seq_length;         // 访问序列长度
25          float locality_factor;  // 局部性因子（0-1），越大局部性越强
26      } Config;
```

图 2-2 配置管理数据结构

3.模拟器核心模块

（1）功能职责：

1）模拟器状态管理

2）内存资源分配和管理

3）页面访问模拟流程控制

4）缺页统计和结果收集

（2）关键数据结构：

```
28      // 页框结构
29      typedef struct {
30          int page_id;     // 页号
31          int last_used;   // 最近使用时间（LRU用）
32          int load_time;   // 加载时间（FIFO用）
33          bool valid;      // 是否有效
34      } Frame;
35
36      // 模拟器结构
37      typedef struct {
38          Config config;
39          int *large_array;    // 大数组A模拟进程
40          Frame *frames;       // 页框数组
41          int *page_table;     // 页表
42          int page_faults;     // 缺页次数
43          int current_time;    // 当前时间
44      } Simulator;
```

图 2-3 模拟器数据结构

4.访问序列生成模块

（1）功能职责：

1）根据访问模式生成指令访问序列

2）实现不同的局部性访问模式

3）提供灵活的访问模式配置

（2）支持模式：

1）顺序访问（强局部性）

2）跳转访问（中度局部性）

3）分支访问（模拟 if-else 模式）

4）循环访问（强局部性）

5）局部性随机访问

5.算法实现模块

功能职责：

1）实现 FIFO 页面置换算法

2）实现 LRU 页面置换算法

3）实现 OPT 页面置换算法

4）提供统一的算法接口

6.辅助工具模块

功能职责：

1）进度显示

2）调试信息输出

3）页框状态显示

2.1.1.3 模块之间接口说明

1.主控制模块与其他模块的接口

（1）与配置管理模块：

1）传入：命令行参数

2）传出：Config 结构体

3）调用关系：main() → Config 初始化/验证

（2）与模拟器核心模块：

```
203         Simulator sim;
204         init_simulator(&sim, &config);
205
206         // 生成访问序列
207         generate_access_sequence(&sim);
208
209         // 运行模拟
210         simulate(&sim);
211
212         // 打印结果
213         print_results(&sim);
214
215         // 清理资源
216         cleanup(&sim);
```

图 2-4 主控制模块与模拟器模块接口

2. 模拟器核心模块与其他模块的接口

（1）与访问序列生成模块：

```
206         // 生成访问序列
207         generate_access_sequence(&sim);
```

图 2-5 模拟器模块与访问序列模块接口

（2）与算法实现模块：

```
171         switch (config.algorithm) {
172             case ALG_FIFO:
173                 printf("使用算法: FIFO\n");
174                 break;
175             case ALG_LRU:
176                 printf("使用算法: LRU\n");
177                 break;
178             case ALG_OPT:
179                 printf("使用算法: OPT\n");
180                 break;
181         }
```

图 2-6 模拟器模块与算法实现模块接口

（3）数据接口：

1）通过 Simulator 结构体共享所有数据

2）算法模块直接访问 Simulator 中的 frames、config 等

2.1.1.4 数据处理流程

1.整体处理流程

（1）启动程序，解析命令行参数

（2）验证配置参数，初始化配置

（3）初始化模拟器

（4）生成访问序列

（5）模拟执行

（6）输出结果

（7）清理资源

2.页面访问详细流程

```
// simulate()函数中的核心循环
for (int i = 0; i < config->seq_length; i++) {
    // 步骤 1: 获取当前访问的页面
    int instruction_index = seq[i];
    int page_id = instruction_index / page_size;

    // 步骤 2: 检查页面是否在内存中
    int frame_index = find_page_in_frames(sim, page_id);

    if (frame_index == -1) {
        // 步骤 3: 缺页处理
        sim->page_faults++;

        // 步骤 4: 寻找可用页框（空闲或置换）
        if (没有空闲页框) {
            // 调用相应的置换算法
            frame_index = 算法替换函数(sim);
        }

        // 步骤 5: 加载页面
        load_page(sim, page_id, frame_index);
    } else {
        // 步骤 6: 页面命中，更新信息（LRU 需要）
        sim->frames[frame_index].last_used = i;
    }
}
```

3.算法置换流程（以 FIFO 为例）

（1）FIFO 页面置换流程：

1）遍历所有页框

2）查找 load_time 最小的页框

3）从页表中删除该页框的映射

4）返回该页框索引供新页面使用

（2）数据结构更新：

- frames[].load_time: 新页面加载时设置为当前时间

- page_table[]: 旧页面设为-1，新页面设为 frame_index

2.1.2 验证 Linux 虚拟地址转化为物理地址的机制任务

2.1.2.1 模块框图

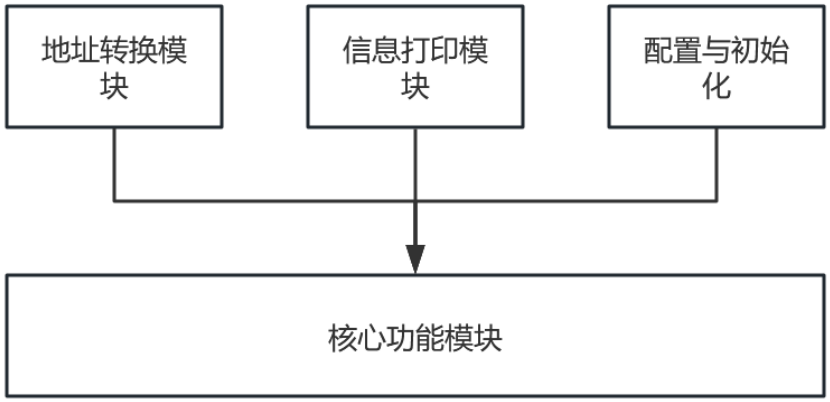

图 2-7 任务二模块框图

2.1.2.2 模块功能说明

1.地址转换模块

地址转换模块是程序的核心功能组件，专门负责将指定进程的虚拟内存地址转换为对应的物理内存地址。该模块通过直接访问 Linux 内核提供的/proc 文件系统中的 pagemap 接口来实现地址转换。其工作原理是首先根据目标进程的 PID 构建 pagemap 文件路径，然后打开该文件并定位到虚拟地址对应的页表项偏移位置。模块会读取 8 字节的页表项数据，检查页面的存在标志位以确认该页面当前是否驻留在物理 RAM 中。如果页面有效存在，模块会从页表项中提取物理帧号(PFN)，结合页内偏移量计算出完整的物理地址。如果页面不存在或无法访问，模块会安全地返回 0 值表示转换失败。这个模块实现了从用户空间直接查询页表映射的低级操作，为后续的地址分析提供了基础数据支持。

```
16  // 通用函数：通过 /proc/pid/pagemap 获取物理地址
17  uint64_t get_phys_addr(pid_t pid, void *virt_addr) {
18      char pagemap_path[64];
19      snprintf(pagemap_path, sizeof(pagemap_path), "/proc/%d/pagemap", (int)pid);
20      int fd = open(pagemap_path, O_RDONLY);
21      if (fd < 0) {
22          perror("open pagemap");
23          return 0;
24      }
25
26      uint64_t virt_page = (uintptr_t)virt_addr / page_size;
27      uint64_t entry;
28      off_t offset = virt_page * sizeof(entry);
29
30      if (pread(fd, &entry, sizeof(entry), offset) != sizeof(entry)) {
31          perror("pread pagemap");
32          close(fd);
33          return 0;
34      }
35      close(fd);
36
37      if (!(entry & (1ULL << 63))) {
38          // printf("Page not present in RAM.\n");
39          return 0;
40      }
41
42      uint64_t pfn = entry & ((1ULL << 55) - 1);
43      uint64_t page_offset = (uintptr_t)virt_addr % page_size;
44      return (pfn << (__builtin_ctzll(page_size))) + page_offset;
45  }
```

图 2-8 地址转换模块

2.信息打印模块

信息打印模块负责将地址转换结果以清晰易读的格式呈现给用户。该模块接收符号名称、虚拟地址和进程 PID 作为输入参数，调用地址转换模块获取物理地址后，进行全面的地址信息格式化输出。模块会计算并显示虚拟地址对应的页号和页内偏移量，同样也会计算物理地址的页号和偏移信息。特别重要的是，模块会自动验证虚拟地址的页内偏移与物理地址的页内偏移是否保持一致，这是验证地址转换正确性的关键检查点。如果发现偏移不匹配的情况，模块会输出明确的警告信息提示用户可能存在异常。通过这种结构化的输出方式，用户可以直观地理解虚拟地址到物理地址的映射关系，包括地址的空间分布和页对齐特性。

```
47      // 打印地址信息
48      void print_info(const char *name, void *addr, pid_t pid) {
49          uint64_t vaddr = (uint64_t)addr;
50          uint64_t phys = get_phys_addr(pid, addr);
51          uint64_t vpage = vaddr / page_size;
52          uint64_t voffset = vaddr % page_size;
53
54          printf("=== %s ===\n", name);
55          printf("Symbol name:       %s\n", name);
56          printf("Virtual address:   0x%016lx\n", vaddr);
57          printf("Virtual page #:    %lu (0x%lx)\n", vpage, vpage);
58          printf("Offset in page:    0x%03lx (%lu bytes)\n", voffset, voffset);
59
60          if (phys) {
61              uint64_t ppage = phys / page_size;
62              uint64_t poffset = phys % page_size;
63              printf("Physical address:  0x%016lx\n", phys);
64              printf("Physical page #:   %lu (0x%lx)\n", ppage, ppage);
65              printf("Offset in page:    0x%03lx (%lu bytes)\n", poffset, poffset);
66
67              // 验证转换是否正确
68              if (voffset != poffset) {
69                  printf("WARNING: Virtual and physical offsets don't match!\n");
70              }
71          } else {
72              printf("Physical address:  (not in RAM or inaccessible)\n");
73          }
74          printf("\n");
75      }
```

图 2-9 信息打印模块

3.配置与初始化模块

配置与初始化模块负责整个程序的启动准备和运行环境配置工作。在程序开始执行时，该模块首先调用系统接口获取当前系统的内存页大小，这是后续所有地址计算的基础参数。接着模块解析用户通过命令行传入的参数选项，支持指定运行模式和显示帮助信息等功能。当用户请求帮助或输入无效参数时，模块会显示详细的使用说明文档，指导用户正确使用程序的各项功能。根据解析得到的模式参数，该模块将程序执行流分派到对应的功能模块，实现不同实验场景的切换。这个模块作为程序的入口控制器，确保了程序能够根据用户需求灵活地运行在不同模式下，同时提供了友好的用户交互体验。

4.默认模式

默认模式展示了程序最基本的功能，即分析当前进程内部静态定义的变量和函数的地址映射关系。该模式首先获取当前进程的 PID 标识符，然后输出基本的系统信息包括内存页大小和使用的 pagemap 文件路径。接着程序会分析全局变量 global_var 的地址转换情况，这个变量在编译时被初始化为特定的十六进制值，用于演示数据段的地址映射。同时程序也会分析 dummy_function 函数的地址转换，这个简单的空函数用于演示代码段的地址特性。通过这种对静态地址空间的分析，用户可以直观地理解进程内存布局中不同区域的映射特点，包括数据段和代码段在虚拟地址空间和物理内存中的对应关系。

```
127     // ========== 默认模式 ==========
128     void run_default() {
129         pid_t pid = getpid();
130         printf("=== Default Mode: Current Process (%d) ===\n", pid);
131         printf("Page size: %ld bytes\n", page_size);
132         printf("Pagemap file: /proc/%d/pagemap\n\n", pid);
133 
134         print_info("global_var", &global_var, pid);
135         print_info("dummy_function", (void*)dummy_function, pid);
136     }
```

图 2-10 默认模式

（1）模式 1

模式 1 设计用于演示在多进程环境中相同虚拟地址可能映射到不同物理地址的现象。该模式使用 mmap 系统调用在固定的虚拟地址 0x10000000 处分配一个内存页，并通过 MAP_FIXED 标志确保地址的确定性。为了区分不同进程的映射内容，程序将当前进程的 PID 写入该内存区域作为标识。模式特别设置了 2 秒的延迟等待，为用户提供足够的时间启动另一个相同的进程实例进行对比观察。当两个进程都运行时，它们会在相同的虚拟地址上建立内存映射，但由于进程空间的独立性，这些虚拟地址会映射到各自不同的物理内存页面。这个模式生动地展示了操作系统虚拟内存管理的基本原理，即每个进程拥有独立的地址空间，相同的虚拟地址在不同进程中具有不同的物理含义。

```
84      // ========== 模式1: 固定虚拟地址 mmap ==========
85      void run_mode1() {
86          const void *fixed_virt = (void*)0x10000000; // 固定虚拟地址
87          void *mem = mmap((void*)fixed_virt, page_size,
88                           PROT_READ | PROT_WRITE,
89                           MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED,
90                           -1, 0);
91          if (mem == MAP_FAILED) {
92              perror("mmap for mode1");
93              exit(1);
94          }
95
96          pid_t mypid = getpid();
97          memcpy(mem, &mypid, sizeof(mypid)); // 写入 PID 作为标识
98
99          printf("[Mode 1] Process %d using fixed virtual address %p\n", mypid, mem);
100         printf("Pagemap file: /proc/%d/pagemap\n\n", mypid);
101
102         sleep(2); // 留时间给另一个进程启动
103
104         print_info("Fixed mmap region", mem, mypid);
105         munmap(mem, page_size);
106     }
107
```

图 2-11 模式 1

（2）模式 2

模式 2 专注于分析共享库在内存中的映射特性，特别是动态链接库函数的地址转换行为。该模式首先使用 dlsym 函数获取标准 C 库中 printf 函数的虚拟地址，这个地址在程序运行期间通过动态链接器确定。为了确保目标页面被加载到物理内存中，程序主动调用 printf 函数输出提示信息，触发缺页中断和页面加载机制。然后程序分析 printf 函数地址的转换情况，显示其在物理内存中的实际位置。模式设置了 5 秒的较长延迟，方便用户同时运行多个进程实例进行比较观察。当多个进程都加载了相同的共享库时，它们可能共享相同的物理内存页面，这个模式帮助用户理解操作系统的共享库机制和内存共享优化技术，展示了只读代码段在多个进程间的物理共享特性。

```
108     // ========== 模式2: 共享库物理地址 ==========
109     void run_mode2() {
110         pid_t mypid = getpid();
111 
112         // 强制解析 printf 地址
113         void *printf_addr = dlsym(RTLD_DEFAULT, "printf");
114         if (!printf_addr) {
115             fprintf(stderr, "dlsym failed to find printf\n");
116             exit(1);
117         }
118 
119         // 触发页面加载
120         printf("[Mode 2] Hello from PID %d\n", mypid);
121         printf("Pagemap file: /proc/%d/pagemap\n\n", mypid);
122 
123         print_info("printf (from libc)", printf_addr, mypid);
124         sleep(5); // 方便对比
125     }
```

图 2-12 模式 2

2.1.2.3 模块之间接口说明

1. 地址转换模块接口

调用接口：被 print_info()调用

输入接口：get_phys_addr(pid_t pid, void *virt_addr)

输出接口：返回物理地址或 0

依赖接口：

文件系统：open()、pread()、close()

全局变量：page_size

2. 信息打印模块接口

调用接口：被三个模式函数调用

输入接口：print_info(const char *name, void *addr, pid_t pid)

内部调用：调用 get_phys_addr()

输出接口：printf()系列输出到控制台

3. 配置与初始化模块接口

主入口：main(int argc, char *argv[])

系统接口：sysconf(_SC_PAGESIZE)获取页大小

参数解析：getopt()处理命令行

模式分发：根据 mode 变量调用对应函数

帮助接口：show_usage()显示帮助信息

4. 核心功能模块接口

（1）默认模式接口

函数声明：run_default()

系统调用：getpid()获取进程 ID

分析调用：调用 print_info()分析静态地址

（2）模式 1 接口

函数声明：run_mode1()

内存映射：mmap()固定地址分配

进程标识：memcpy()写入 PID

延迟同步：sleep(2)等待其他进程

清理释放：munmap()释放内存

（3）模式 2 接口

函数声明：run_mode2(

动态链接：dlsym()获取 printf 地址

页面触发：printf()调用加载页面

长时延迟：sleep(5)便于对比

5. 全局数据接口

页大小变量：page_size 在 main 初始化，各模块使用

测试变量：global_var 和 dummy_function()用于默认模式

2.1.2.4 数据处理流程

1. 程序启动与初始化流程

（1）程序启动：操作系统加载可执行文件，传递命令行参数

（2）页大小获取：调用 sysconf(_SC_PAGESIZE)获取系统内存页大小

（3）参数解析：使用 getopt()解析-m 选项，确定运行模式

（4）模式分发：根据模式值调用对应功能函数（run_default/run_mode1/run_mode2）

2. 物理地址转换核心流程

（1）输入接收：接收目标进程 PID 和虚拟地址作为输入参数

（2）文件访问：构造/proc/<pid>/pagemap 文件路径并打开

（3）偏移计算：计算虚拟地址对应的页表项文件偏移

（4）数据读取：从 pagemap 文件读取 8 字节页表项数据

（5）状态检查：检查页表项第 63 位，确认页面是否驻留 RAM

（6）地址计算：提取物理帧号，结合页内偏移计算物理地址

（7）结果返回：返回物理地址或 0（页面不存在）

3. 信息显示标准流程

（1）参数接收：接收符号名、虚拟地址、进程 PID

（2）地址转换：调用 get_phys_addr()获取物理地址

（3）分页计算：计算虚拟/物理页号和页内偏移

（4）数据验证：比较虚拟与物理偏移是否一致

（5）格式化输出：按标准格式输出所有地址信息

（6）警告提示：偏移不一致时输出警告信息

4. 默认模式执行流程

（1）进程识别：调用 getpid()获取当前进程 ID

（2）基本信息输出：显示页大小和 pagemap 文件路径

（3）全局变量分析：分析 global_var 的地址转换

（4）函数地址分析：分析 dummy_function 的地址转换

（5）结果显示：输出两处地址的虚拟-物理映射关系

5. 模式 1（固定虚拟地址）执行流程

（1）内存映射：在固定地址 0x10000000 处分配内存页

（2）数据标识：将当前 PID 写入映射区域作为内容标识

（3）进程信息显示：输出进程 ID 和使用地址信息

（4）延迟等待：sleep(2)等待其他进程启动

（5）地址分析：分析映射区域的地址转换

（6）资源释放：释放映射的内存区域

6. 模式 2（共享库分析）执行流程

（1）函数定位：使用 dlsym()获取 printf 函数地址

（2）页面加载：调用 printf 触发共享库页面加载

（3）进程信息显示：输出进程 ID 和提示信息

（4）地址分析：分析 printf 函数的地址转换

（5）观察延迟：sleep(5)提供对比观察时间

7. 错误处理流程

（1）系统调用错误：perror()输出错误信息并退出

（2）参数错误：显示使用说明并正常退出

（3）页面不存在：返回 0 值，print_info 显示"not in RAM"

（4）文件访问错误：关闭文件描述符，返回错误

8. 多进程对比实验流程（用户操作）

（1）第一次执行：运行程序选择模式 1 或模式 2

（2）第二次执行：在延迟时间内启动第二个进程实例

（3）数据对比：比较两个进程相同虚拟地址的物理映射

（4）结果分析：观察是否映射到相同/不同的物理地址

2.2 详细设计

2.2.1 程序演示页面淘汰算法任务

2.2.1.1 核心函数流程

1.主流程函数：main（）

```
int main(int argc, char *argv[]) {
    // 阶段 1: 初始化
    srand(time(NULL));              // 设置随机种子
    Config config;                  // 创建默认配置
    parse_arguments();              // 解析命令行参数
    validate_config();              // 验证配置有效性
```

```
    // 阶段 2: 模拟器初始化
    Simulator sim;
    init_simulator(&sim, &config);        // 分配内存，初始化数据结构

    // 阶段 3: 生成访问序列
    generate_access_sequence(&sim);      // 根据模式生成指令访问序列

    // 阶段 4: 执行模拟
    simulate(&sim);                       // 核心模拟循环

    // 阶段 5: 结果输出与清理
    print_results(&sim);                  // 输出统计结果
    cleanup(&sim);                        // 释放所有内存

    return 0;
}
```

2.核心模拟函数：simulate（）

```
void simulate(Simulator *sim) {
    Config *config = &sim->config;
    int *seq = config->access_sequence;
    int page_size = config->page_size;

    // 遍历所有指令访问
    for (int i = 0; i < config->seq_length; i++) {
        sim->current_time = i;   // 更新时间戳

        // 步骤 1: 计算当前访问的页号
        int instruction_index = seq[i];            // 指令地址
        int page_id = instruction_index / page_size;   // 页号

        // 步骤 2: 检查页面是否在内存中
        int frame_index = find_page_in_frames(sim, page_id);

        if (frame_index == -1) {
            // 步骤 3A: 缺页处理
            sim->page_faults++;   // 缺页计数增加

            // 步骤 4A: 寻找替换页框
            frame_index = find_free_or_replace(sim, i);

            // 步骤 5A: 加载新页面
```

```
            load_page(sim, page_id, frame_index);
        } else {
            // 步骤 3B: 页面命中处理
            sim->frames[frame_index].last_used = i;  // 更新 LRU 时间戳
        }

        // 可选：显示进度
        if (i % 200 == 0) print_progress(i, config->seq_length);
    }
}
```

3.页面查找函数：find_page_in_frames()

```
int find_page_in_frames(Simulator *sim, int page_id) {
    // 线性扫描所有页框，查找指定页号
    for (int i = 0; i < sim->config.num_frames; i++) {
        if (sim->frames[i].valid && sim->frames[i].page_id == page_id) {
            return i;  // 找到，返回页框索引
        }
    }
    return -1;  // 未找到，返回-1 表示缺页
}
```

4.页面置换决策函数

（1）FIFO 算法：fifo_replace()

```
int fifo_replace(Simulator *sim) {
    int oldest_index = 0;
    int oldest_time = sim->frames[0].load_time;

    // 查找加载时间最早的页框
    for (int i = 1; i < sim->config.num_frames; i++) {
        if (sim->frames[i].load_time < oldest_time) {
            oldest_time = sim->frames[i].load_time;
            oldest_index = i;
        }
    }

    // 清理旧页面的页表项
    int old_page = sim->frames[oldest_index].page_id;
    if (old_page >= 0) {
        sim->page_table[old_page] = -1;  // 标记为不在内存
    }

    return oldest_index;
```

```
}
```

（2）LRU 算法：lru_replace()

```
int lru_replace(Simulator *sim) {
    int lru_index = 0;
    int lru_time = sim->frames[0].last_used;

    // 查找最近使用时间最早的页框
    for (int i = 1; i < sim->config.num_frames; i++) {
        if (sim->frames[i].last_used < lru_time) {
            lru_time = sim->frames[i].last_used;
            lru_index = i;
        }
    }

    // 清理旧页面的页表项
    int old_page = sim->frames[lru_index].page_id;
    if (old_page >= 0) {
        sim->page_table[old_page] = -1;
    }

    return lru_index;
}
```

（3）OPT 算法：opt_replace()

```
int opt_replace(Simulator *sim, int current_index) {
    int farthest_index = 0;
    int farthest_distance = -1;
    int seq_length = sim->config.seq_length;
    int page_size = sim->config.page_size;
    int *seq = sim->config.access_sequence;

    for (int i = 0; i < sim->config.num_frames; i++) {
        if (!sim->frames[i].valid) {
            return i;   // 有空闲页框，直接使用
        }

        int page_id = sim->frames[i].page_id;
        int next_use = seq_length;   // 默认未来不再使用

        // 在后续访问序列中查找该页面的下一次使用
        for (int j = current_index + 1; j < seq_length; j++) {
            int accessed_page = seq[j] / page_size;
```

```
                if (accessed_page == page_id) {
                    next_use = j;
                    break;
                }
            }

            // 如果页面在未来不再使用，直接选择它
            if (next_use == seq_length) {
                sim->page_table[page_id] = -1;
                return i;
            }

            // 更新最远使用的页面
            if (next_use > farthest_distance) {
                farthest_distance = next_use;
                farthest_index = i;
            }
        }

        // 清理被选中页面的页表项
        int old_page = sim->frames[farthest_index].page_id;
        sim->page_table[old_page] = -1;

        return farthest_index;
    }
```

2.2.1.2 核心数据结构

1.Config 配置结构体

存储和管理所有模拟配置参数

```
15      // 配置参数
16      typedef struct {
17          int page_size;          // 页面大小（指令数）
18          int num_frames;         // 页框数量
19          int total_instructions; // 总指令数
20          int num_pages;          // 总页数
21          Algorithm algorithm;    // 使用的算法
22          int access_pattern;     // 访问模式：0-顺序，1-跳转，2-分支，3-循环，4-局部性随机
23          int *access_sequence;   // 访问序列
24          int seq_length;         // 访问序列长度
25          float locality_factor;  // 局部性因子（0-1），越大局部性越强
26      } Config;
```

图 2-13 config 配置结构体

page_size：决定虚拟内存分页的大小，影响页号计算

num_frames：物理内存容量，直接影响缺页率

access_sequence：动态分配的指令地址数组，模拟程序执行轨迹

locality_factor：控制访问的局部性强度，值越大局部性越强

2.Frame 页框结构体

表示物理内存中的一个页框

```
28      // 页框结构
29      typedef struct {
30          int page_id;     // 页号
31          int last_used;   // 最近使用时间（LRU用）
32          int load_time;   // 加载时间（FIFO用）
33          bool valid;      // 是否有效
34      } Frame;
```

图 2-14 Frame 页框结构

page_id：逻辑页号到物理页框的映射关键

last_used 和 load_time：不同算法的时态信息记录

valid：快速判断页框是否被占用

3.Simulator 模拟器结构体

整合所有模拟状态和数据

```
37      typedef struct {
38          Config config;
39          int *large_array;    // 大数组A模拟进程
40          Frame *frames;       // 页框数组
41          int *page_table;     // 页表
42          int page_faults;     // 缺页次数
43          int current_time;    // 当前时间
44      } Simulator;
```

图 2-15 Simulator 模拟器结构体

large_array：模拟进程的指令存储空间，大小=total_instructions

frames：物理内存页框数组，大小=num_frames

page_table：简化页表，数组索引=页号，值=页框索引（-1 表示不在内存）

page_faults：核心性能指标统计

2.2.2 验证 Linux 虚拟地址转化为物理地址的机制任务

2.2.2.1 核心函数流程

1.初始化阶段

程序启动时调用 sysconf(_SC_PAGESIZE) 获取系统页大小（通常为 4096 字节），并将其存储于全局变量 page_size，作为后续地址对齐与页号计算的基础。

2.命令行解析与模式选择

使用 getopt() 解析 -m 参数：若指定 -m 1，进入固定虚拟地址映射模式（Mode 1）；若指定 -m 2，进入共享库函数地址探测模式（Mode 2）；无参数时，执行默认模式，展示当前进程内全局变量与函数的地址信息。

3.默认模式（Default Mode）

调用 run_default()，依次对全局变量 global_var 与函数 dummy_function 调用 print_info()，输出其虚拟地址、所属虚拟页号、页内偏移，以及通过 pagemap 查询得到的物理地址（若存在）。

4.模式 1：固定虚拟地址映射（Fixed Virtual Address Mapping）

在 run_mode1() 中，程序使用 mmap() 配合 MAP_FIXED 标志，在固定虚拟地址 0x10000000 处映射一页可读写匿名内存，并写入当前进程 PID 作为标识。随后调用 print_info() 展示该区域的物理映射情况。此模式旨在演示：即使多个进程使用相同的虚拟地址，其对应的物理地址通常不同，反映操作系统的地址空间隔离机制。

5.模式 2：共享库物理地址探测（Shared Library Mapping）

run_mode2() 利用 dlsym(RTLD_DEFAULT, "printf") 获取标准库函数 printf 的虚拟地址，并主动调用一次以确保其代码页已加载至内存。随后通过 print_info() 输出其物理地址。该模式用于验证：只读共享库（如 libc）在多个进程中通常映射至相同的物理页帧，体现内存共享优化机制。

6.地址转换核心：get_phys_addr()

该函数是整个程序的关键组件，负责从 /proc/<pid>/pagemap 中读取指定虚拟页的 64 位页表项：首先校验第 63 位（Present 位），判断该页是否驻留于物理内存；若驻留，则提取

低 55 位作为物理页帧号（PFN）；结合原始虚拟地址的页内偏移，计算出完整的物理地址：

$$phys\_addr=(PFN\times page\_size)+(virt\_addr \bmod page\_size)$$

若页未驻留或读取失败，返回 0 表示不可用。

7.信息输出：print_info()

封装地址解析与格式化输出逻辑，同时校验虚拟与物理地址的页内偏移是否一致（理论上必须一致），若不一致则发出警告，辅助调试潜在映射错误。

2.2.2.2 核心数据结构

本程序未定义复杂的自定义数据结构，主要依赖系统原生类型与 Linux 内核暴露的虚拟内存抽象接口。关键数据元素如表 2-1：

表 2-1 核心数据结构

数据项	类型	作用说明
page_size	long（全局静态）	存储系统页大小（单位：字节），用于虚拟/物理地址的页对齐计算与页号提取。
virt_addr	void*	输入的用户空间虚拟地址，作为地址转换的起点。
pid	pid_t	目标进程 ID，用于构造 /proc/<pid>/pagemap 路径；本程序中通常为当前进程自身。
entry	uint64_t	从 /proc/<pid>/pagemap 读取的单个页表项，包含物理页帧号（PFN）及状态标志。
pfn	uint64_t	物理页帧号（Page Frame Number），由 entry 的低 55 位提取，代表物理内存中的页索引。
vpage / ppage	uint64_t	虚拟页号与物理页号，分别由 virt_addr / page_size 和 phys_addr / page_size 计算得出，用于页粒度分析。
voffset / poffset	uint64_t	虚拟与物理地址在各自页内的偏移量，理论上应完全相等，用于一致性校验。

2.3 代码实现

2.3.1 程序演示页面淘汰算法任务

1.模拟主循环核心代码

这段代码是模拟器的核心执行引擎，它实现了完整的页面访问模拟流程。程序首先从配置中获取访问序列和页面大小，然后开始遍历整个访问序列。对于序列中的每个指令访问，程序首先计算当前指令所属的页号，这是通过将指令索引除以页面大小得到的整数除法。接下来，程序调用 find_page_in_frames 函数在物理页框中查找该页面是否存在，这个函数会遍历所有页框，检查是否有页框存储着目标页号。

如果页面不在内存中（frame_index 为-1），则发生缺页异常，缺页计数器增加。此时程序需要为请求的页面分配物理页框，它首先检查是否有空闲页框，遍历所有页框寻找 valid 字段为 false 的页框。如果有空闲页框，就直接使用；如果没有空闲页框，就需要根据配置的算法选择牺牲页面进行置换。程序通过 switch 语句根据算法类型调用相应的置换函数：对于 FIFO 算法调用 fifo_replace，对于 LRU 算法调用 lru_replace，对于 OPT 算法调用 opt_replace 并传入当前访问索引以便分析未来访问模式。获得可用的页框索引后，调用 load_page 函数将请求的页面加载到该页框中，这个函数会更新页框的状态信息并建立页表映射。

如果页面已经在内存中（页面命中），则只需更新该页框的 last_used 字段为当前时间，这是为了 LRU 算法能够正确追踪页面最近使用的时间。在整个模拟过程中，每处理 200 条指令就会调用 print_progress 函数显示进度条，让用户了解模拟的进展情况。当所有指令访问处理完毕后，程序输出完成信息，此时缺页计数器中记录了整个模拟过程中发生的缺页次数，这是评估算法性能的关键指标。

```c
void simulate(Simulator *sim) {
    Config *config = &sim->config;
    int *seq = config->access_sequence;
    int page_size = config->page_size;

    printf("开始模拟...\n\n");

    for (int i = 0; i < config->seq_length; i++) {
        sim->current_time = i;
        int instruction_index = seq[i];
        int page_id = instruction_index / page_size;

        // 检查页面是否在内存中
        int frame_index = find_page_in_frames(sim, page_id);

        if (frame_index == -1) {
            // 缺页!
            sim->page_faults++;

            // 检查是否有空闲页框
            bool has_free_frame = false;
            for (int j = 0; j < config->num_frames; j++) {
                if (!sim->frames[j].valid) {
                    frame_index = j;
                    has_free_frame = true;
                    break;
                }
            }

            // 如果没有空闲页框，需要置换
            if (!has_free_frame) {
                switch (config->algorithm) {
                    case ALG_FIFO:
                        frame_index = fifo_replace(sim);
                        break;
                    case ALG_LRU:
                        frame_index = lru_replace(sim);
                        break;
                    case ALG_OPT:
                        frame_index = opt_replace(sim, i);
                        break;
                }
            }

            // 加载页面
            load_page(sim, page_id, frame_index);
        } else {
            // 页面命中，更新LRU信息
            sim->frames[frame_index].last_used = i;
        }

        // 每200条指令打印一次进度
        if (i % 200 == 0 && i > 0) {
            print_progress(i, config->seq_length);
        }
    }

    printf("\n模拟完成! \n\n");
}
```

图 2-16 模拟主循环核心代码

2.FIFO 页面置换算法实现

这段代码实现了先进先出（FIFO）页面置换算法，其核心思想是选择最早进入内存的页面进行置换。算法首先初始化 oldest_index 为 0，假设第一个页框中的页面是最早加载的，并将 oldest_time 设为该页框的 load_time 值。接着，算法遍历从第二个页框开始的所有页框，比较每个页框的 load_time 与当前记录的 oldest_time。load_time 记录了页面被加载到内存的时间戳，在 load_page 函数中设置，值越小表示页面越早被加载。如果发现某个页框的 load_time 小于当前记录的 oldest_time，说明这个页面比当前记录的最老页面还要早进入内存，于是更新 oldest_time 和 oldest_index 为这个更早页面的信息。

通过这样一次遍历，算法能够找到所有页框中 load_time 最小的页框，即最早加载到内存的页面。找到目标页框后，算法需要清理这个页框对应的页表项，首先获取该页框中存储的页号 old_page，然后检查这个页号是否有效（大于等于 0），如果有效就将页表中对应页号的映射设置为-1，表示该页面不再驻留在内存中。最后，函数返回被选中页框的索引，供调用者加载新的页面。这种实现方式简单高效，时间复杂度为 O(n)，其中 n 是页框数量，但它有一个著名的缺点就是可能出现 Belady 异常，即增加页框数量反而可能导致缺页率增加，因为它只考虑了页面进入内存的时间而没有考虑页面的使用频率。

```
368     int fifo_replace(Simulator *sim) {
369         int oldest_index = 0;
370         int oldest_time = sim->frames[0].load_time;
371 
372         for (int i = 1; i < sim->config.num_frames; i++) {
373             if (sim->frames[i].load_time < oldest_time) {
374                 oldest_time = sim->frames[i].load_time;
375                 oldest_index = i;
376             }
377         }
378 
379         // 从页表中删除旧页面的映射
380         int old_page = sim->frames[oldest_index].page_id;
381         if (old_page >= 0) {
382             sim->page_table[old_page] = -1;
383         }
384 
385         return oldest_index;
386     }
```

图 2-17 FIFO 页面置换算法实现

3.LRU 页面置换算法实现

这段代码实现了最近最少使用（LRU）页面置换算法，该算法基于程序访问的局部性原理，认为最近使用过的页面很可能在不久的将来再次被使用，因此应该置换最长时间没有被访问的页面。算法开始时，将第一个页框设为初始候选，假设它的 last_used 时间戳是最小的，即它是最久未被使用的页面。last_used 字段在页面每次被访问时（包括缺页加载和命中访问）都会更新为当前模拟时间，记录了页面最后一次被访问的时间。

然后算法遍历其余所有页框，比较每个页框的 last_used 值与当前记录的 lru_time。这里的关键是比较条件是 sim->frames[i].last_used < lru_time，因为 last_used 存储的是时间戳，数值越小表示访问时间越早，即越久没有被使用。当发现某个页框的 last_used 值更小时，说明这个页面比当前记录的最久未使用页面还要久未被访问，于是更新 lru_time 和 lru_index。完成遍历后，lru_index 指向的就是所有页框中 last_used 值最小的页框，即最久未被使用的页面。

找到目标页框后，算法需要清理相关数据结构，首先获取该页框中存储的页号，然后检查页号是否有效（有效的页框存储的页号大于等于 0），如果有效就将页表中该页号对应的映射项设置为-1，表示这个页面即将被移出内存，不再可以通过页表直接访问。最后函数返回被选中页框的索引，这样调用者就可以在这个位置加载新的页面。LRU 算法是对实际程序行为较

好的近似，通常能获得比 FIFO 更好的性能，但实现需要为每个页面维护时间戳，并在每次页面访问时更新时间戳，这带来了一定的开销。

```
388     int lru_replace(Simulator *sim) {
389         int lru_index = 0;
390         int lru_time = sim->frames[0].last_used;
391 
392         for (int i = 1; i < sim->config.num_frames; i++) {
393             if (sim->frames[i].last_used < lru_time) {
394                 lru_time = sim->frames[i].last_used;
395                 lru_index = i;
396             }
397         }
398 
399         // 从页表中删除旧页面的映射
400         int old_page = sim->frames[lru_index].page_id;
401         if (old_page >= 0) {
402             sim->page_table[old_page] = -1;
403         }
404 
405         return lru_index;
406     }
```

图 2-18 LRU 页面置换算法实现

4.OPT 页面置换算法实现

这段代码实现了最优置换（OPT）算法，这是一种理论上最优的页面置换算法，它通过预知未来的页面访问序列来选择被置换的页面，总是选择在未来最长时间内不会被访问的页面进行置换。由于在实际系统中无法预知未来的访问序列，OPT 算法通常作为其他算法的性能上界参考。函数接收当前访问序列的索引 current_index 作为参数，这样它就知道从哪个位置开始查找未来的访问。

算法首先检查是否有空闲页框，遍历所有页框，如果发现某个页框的 valid 字段为 false，表示这是一个空闲页框，那么直接返回这个页框的索引，因为有空闲页框时不需要置换现有页面。如果所有页框都被占用，就需要选择牺牲页面。对于每个被占用的页框，算法需要确定该页面对应的页面在未来何时会被再次访问。它初始化 next_use 为 seq_length（访问序列长度），这表示一个默认情况：如果页面在未来不再被访问，那么它的下一次使用时间就是序列末尾之后。

然后算法从 current_index + 1 开始遍历未来的访问序列，对于每个未来的指令访问，计算它访问的页号（指令索引除以页面大小），如果这个页号与当前页框中存储的页号相同，说明这个页面在未来会被访问，于是更新 next_use 为这个未来访问的位置并跳出循环。如果在整个未来序列中都没有找到对该页面的访问，那么 next_use 保持为 seq_length，表示页面在未来不再使用。这种情况下，算法会立即选择这个页面进行置换，因为置换一个永远不会再被使用的页面是最优的选择。它先更新页表，将该页面的映射删除，然后返回这个页框的索引。

如果所有页面在未来都会被再次访问，那么算法需要选择哪个页面在最远的将来才会被访问。它维护 farthest_distance 记录当前找到的最远访问距离，farthest_index 记录对应的页框索引。对于每个页面，如果它的下一次访问时间 next_use 大于当前记录的 farthest_distance，就更新这两个变量。遍历完所有页框后，farthest_index 指向的就是在未来最久才会被访问的页面，也就是 OPT 算法应该选择的置换页面。算法最后删除该页面在页表中的映射并返回页框索引。OPT 算法的时间复杂度较高，因为它需要对每个候选页面扫描未来的访问序列，在最坏情况下时间复杂度为 O(n×m)，其中 n 是页框数量，m 是剩余访问序列长度，但它的缺页率是最低的，为其他算法提供了性能比较的基准。

```
408     int opt_replace(Simulator *sim, int current_index) {
409         // 找到未来最长时间不会使用的页面
410         int *seq = sim->config.access_sequence;
411         int seq_length = sim->config.seq_length;
412         int page_size = sim->config.page_size;
413         int farthest_index = 0;
414         int farthest_distance = -1;
415 
416         for (int i = 0; i < sim->config.num_frames; i++) {
417             if (!sim->frames[i].valid) {
418                 return i; // 有空闲页框
419             }
420 
421             int page_id = sim->frames[i].page_id;
422             int next_use = seq_length; // 默认未来不再使用
423 
424             // 查找页面在未来何时被使用
425             for (int j = current_index + 1; j < seq_length; j++) {
426                 int accessed_page = seq[j] / page_size;
427                 if (accessed_page == page_id) {
428                     next_use = j;
429                     break;
430                 }
433             // 如果页面在未来不再使用，直接替换它
434             if (next_use == seq_length) {
435                 // 从页表中删除旧页面的映射
436                 sim->page_table[page_id] = -1;
437                 return i;
438             }
439 
440             // 更新最远使用的页面
441             if (next_use > farthest_distance) {
442                 farthest_distance = next_use;
443                 farthest_index = i;
444             }
445         }
446 
447         // 从页表中删除旧页面的映射
448         int old_page = sim->frames[farthest_index].page_id;
449         sim->page_table[old_page] = -1;
450 
451         return farthest_index;
452     }
```

图 2-19 OPT 页面置换算法实现

5.循环访问模式生成代码

这段代码生成了循环访问模式的指令序列，模拟了程序中常见的循环结构访问模式，这种模式通常表现出很强的空间局部性和时间局部性。程序首先定义了几个循环区域，设置了循环数量为 5，每个循环的大小为 3 个页面（config->page_size * 3），这意味着每个循环区域跨越

3 个连续的虚拟页面。然后为每个循环随机生成起始位置，确保起始位置加上循环大小不会超出总的指令范围，这是通过 rand() % (total_inst - loop_size)实现的，保证了循环完全在指令空间内。

初始化完成后，代码开始生成访问序列，使用 current_loop 记录当前所在的循环区域，pos_in_loop 记录在循环内的相对位置。对于序列中的每个访问，计算指令地址为当前循环起始位置加上循环内位置，这样生成的访问会在当前循环区域内顺序进行。每次访问后，pos_in_loop 增加 1 并对循环大小取模，这实现了循环内的环绕访问：当到达循环末尾时，下一个访问会回到循环开头。

为了模拟真实程序中循环间的切换，代码在每次完成一个完整循环（即 pos_in_loop 回到 0 时）后，有 30%的概率切换到另一个随机选择的循环区域，这是通过 rand() % 100 < 30 的条件判断实现的。这种设计模拟了程序在不同循环体之间的跳转，比如嵌套循环或函数调用中的循环。如果未触发切换，程序继续在当前循环中执行下一个迭代。这种访问模式具有很强的局部性，因为大部分时间都在少数几个循环区域内顺序访问，偶尔在区域间跳转，这通常会导致较高的缓存命中率，因为同一循环内的访问倾向于集中在连续的几个页面中，并且循环的重复执行提供了时间局部性。这种模式对于测试页面置换算法如何处理具有强局部性的工作负载非常有代表性，特别是对于 LRU 这类基于访问历史的算法，循环访问模式通常能取得很好的效果。

```
301             case 3: // 循环访问（强局部性）
302                 {
303                     // 创建几个循环区域
304                     int num_loops = 5;
305                     int loop_size = config->page_size * 3; // 每个循环3个页面
306                     int loop_start[num_loops];
307                     for (int i = 0; i < num_loops; i++) {
308                         loop_start[i] = rand() % (total_inst - loop_size);
309                     }
310 
311                     int current_loop = 0;
312                     int pos_in_loop = 0;
313 
314                     for (int i = 0; i < config->seq_length; i++) {
315                         seq[i] = loop_start[current_loop] + pos_in_loop;
316                         pos_in_loop = (pos_in_loop + 1) % loop_size;
317 
318                         // 偶尔切换到其他循环
319                         if (pos_in_loop == 0 && rand() % 100 < 30) {
320                             current_loop = rand() % num_loops;
321                         }
322                     }
323                 }
324                 break;
```

图 2-20 循环访问模式生成代码

2.3.2 验证 Linux 虚拟地址转化为物理地址的机制任务

1.虚拟地址到物理地址的转换：get_phys_addr()

该函数是整个程序的技术核心，其实现基于 Linux 的 pagemap 机制。

（1）构造 pagemap 路径

根据目标进程 ID 构造路径字符串 /proc/<pid>/pagemap，并以只读方式打开该文件：

```
snprintf(pagemap_path, sizeof(pagemap_path), "/proc/%d/pagemap", (int)pid);
int fd = open(pagemap_path, O_RDONLY);
```

若打开失败，如权限不足或进程不存在，函数返回 0。

（2）计算虚拟页号与文件偏移

虚拟地址首先按页对齐，得到其所属的虚拟页号：

```
uint64_t virt_page = (uintptr_t)virt_addr / page_size;
```

由于每个页表项占 8 字节（sizeof(uint64_t)），对应在 pagemap 文件中的偏移为：

```
off_t offset = virt_page * sizeof(entry);
```

（3）读取页表项（Page Table Entry）

使用 pread() 从指定偏移处读取一个 64 位条目：

```
if (pread(fd, &entry, sizeof(entry), offset) != sizeof(entry)) { ... }
```

此操作避免了显式 lseek()，保证线程安全。

（4）解析页表项

Linux pagemap 条目的格式定义如下：

Bit 63：Present 位。若为 1，表示该虚拟页当前驻留在物理内存中。

Bits 0–54：物理页帧号（PFN, Page Frame Number）。

程序首先检查 Present 位：

```
if (!(entry & (1ULL << 63))) {
    return 0; // 页不在 RAM 中
}
```

若页存在，则提取 PFN：

```
uint64_t pfn = entry & ((1ULL << 55) - 1);
```

（5）计算物理地址

物理地址由两部分组成：物理页基址 + 页内偏移。其中：

页内偏移 = virt_addr % page_size

物理页基址 = pfn * page_size

程序使用位运算高效实现乘法：

```
uint64_t page_offset = (uintptr_t)virt_addr % page_size;
return (pfn << (__builtin_ctzll(page_size))) + page_offset;
```

3 测试与分析

3.1 系统测试及结果说明

3.1.1 程序演示页面淘汰算法任务

1.实验环境：

虚拟机 Linux 6.8.12

2.测试结果：

（1）FIFO 淘汰算法实现，如图 3-1 所示为页框数为 4、分支访问模式时的缺页情况。

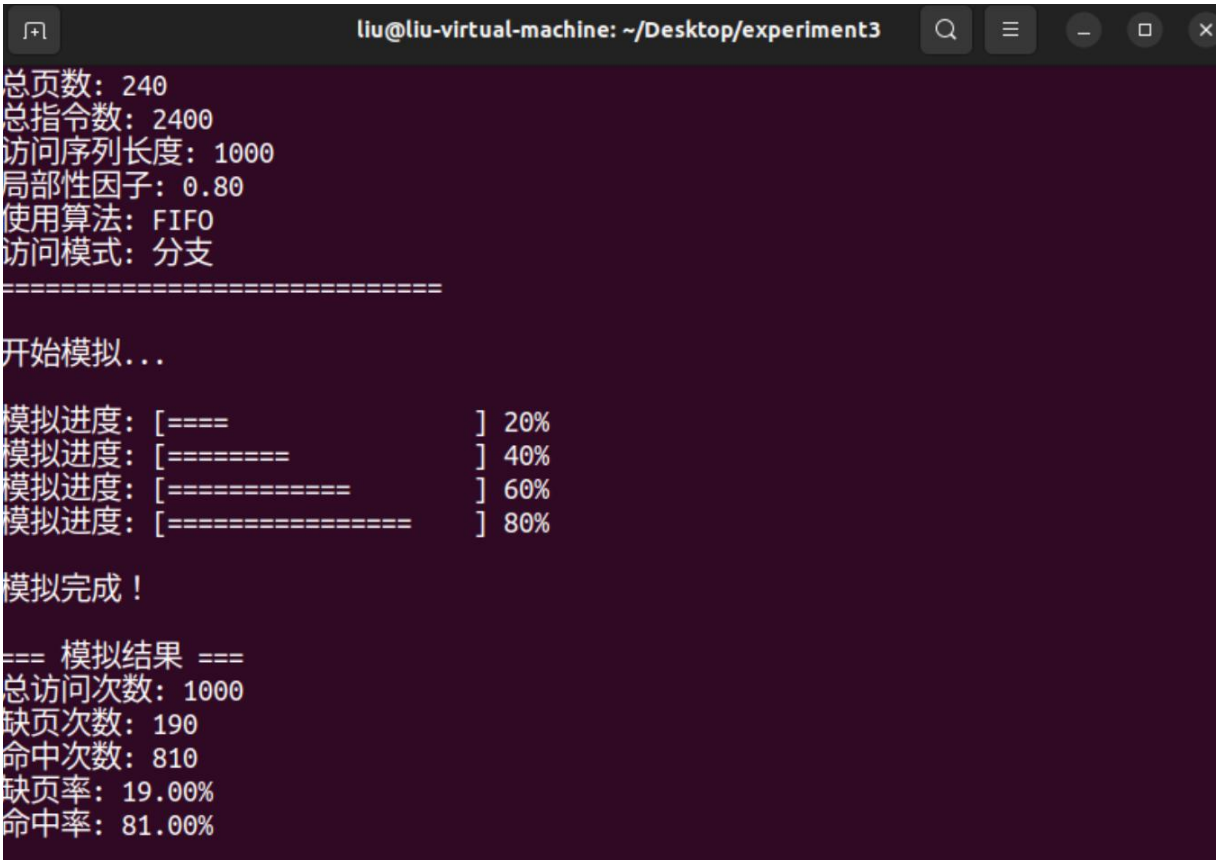

图 3-1 FIFO 淘汰算法

（2）LRU 淘汰算法实现，如图 3-2 所示为页框数为 4、分支访问模式时的缺页情况。

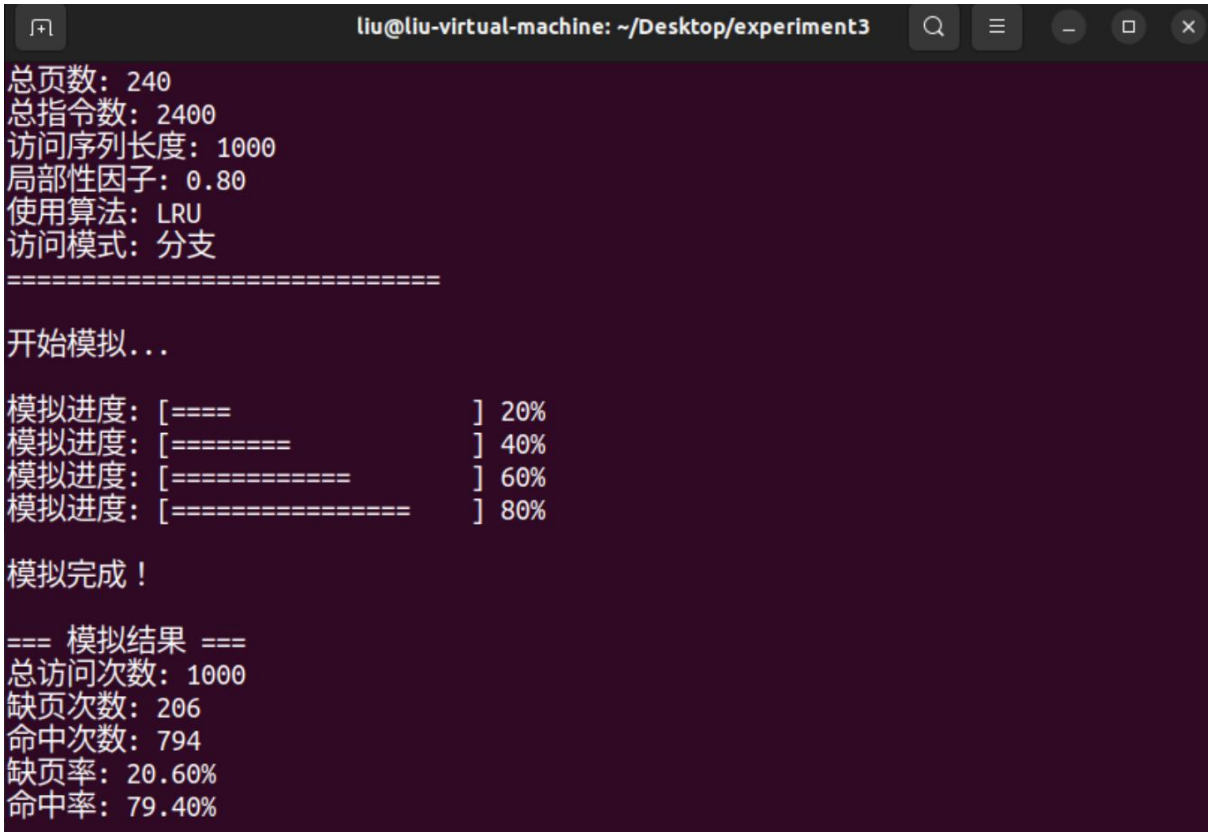

图 3-2 LRU 淘汰算法

（3）OPT 淘汰算法实现，如图 3-3 所示为页框数为 4、分支访问模式时的缺页情况。

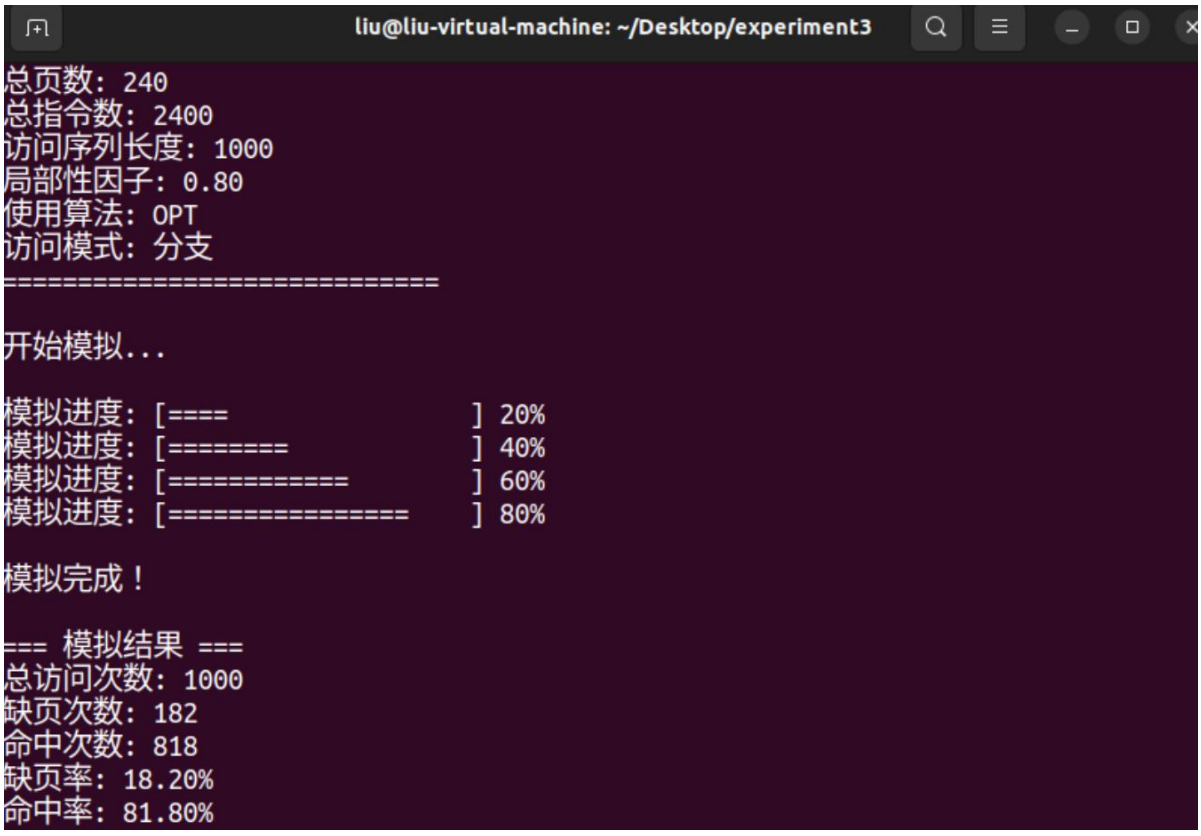

图 3-3 OPT 淘汰算法

3.1.2 验证 Linux 虚拟地址转化为物理地址的机制任务

（1）以 sudo 命令运行 vtop.c 程序，可以计算某个变量或函数虚拟地址对应的物理地址等信息，如图 3-4 所示。

```
liu@liu-virtual-machine:~/Desktop/experiment3$ sudo ./vtop
[sudo] password for liu:
=== Default Mode: Current Process (14735) ===
Page size: 4096 bytes
Pagemap file: /proc/14735/pagemap

=== global_var ===
Symbol name:        global_var
Virtual address:    0x000056f441622010
Virtual page #:     23341569570 (0x56f441622)
Offset in page:     0x010 (16 bytes)
Physical address:   0x0000000011967010
Physical page #:    72039 (0x11967)
Offset in page:     0x010 (16 bytes)

=== dummy_function ===
Symbol name:        dummy_function
Virtual address:    0x000056f44161f6c3
Virtual page #:     23341569567 (0x56f44161f)
Offset in page:     0x6c3 (1731 bytes)
Physical address:   0x000000003fede6c3
Physical page #:    261854 (0x3fede)
Offset in page:     0x6c3 (1731 bytes)
```

图 3-4 虚拟地址物理地址对应信息

（2）不同进程的同一虚拟地址对应不同的物理地址，如图 3-5 所示，在不同的进程中，可以存在同一虚拟地址对应不同的物理地址的情况。

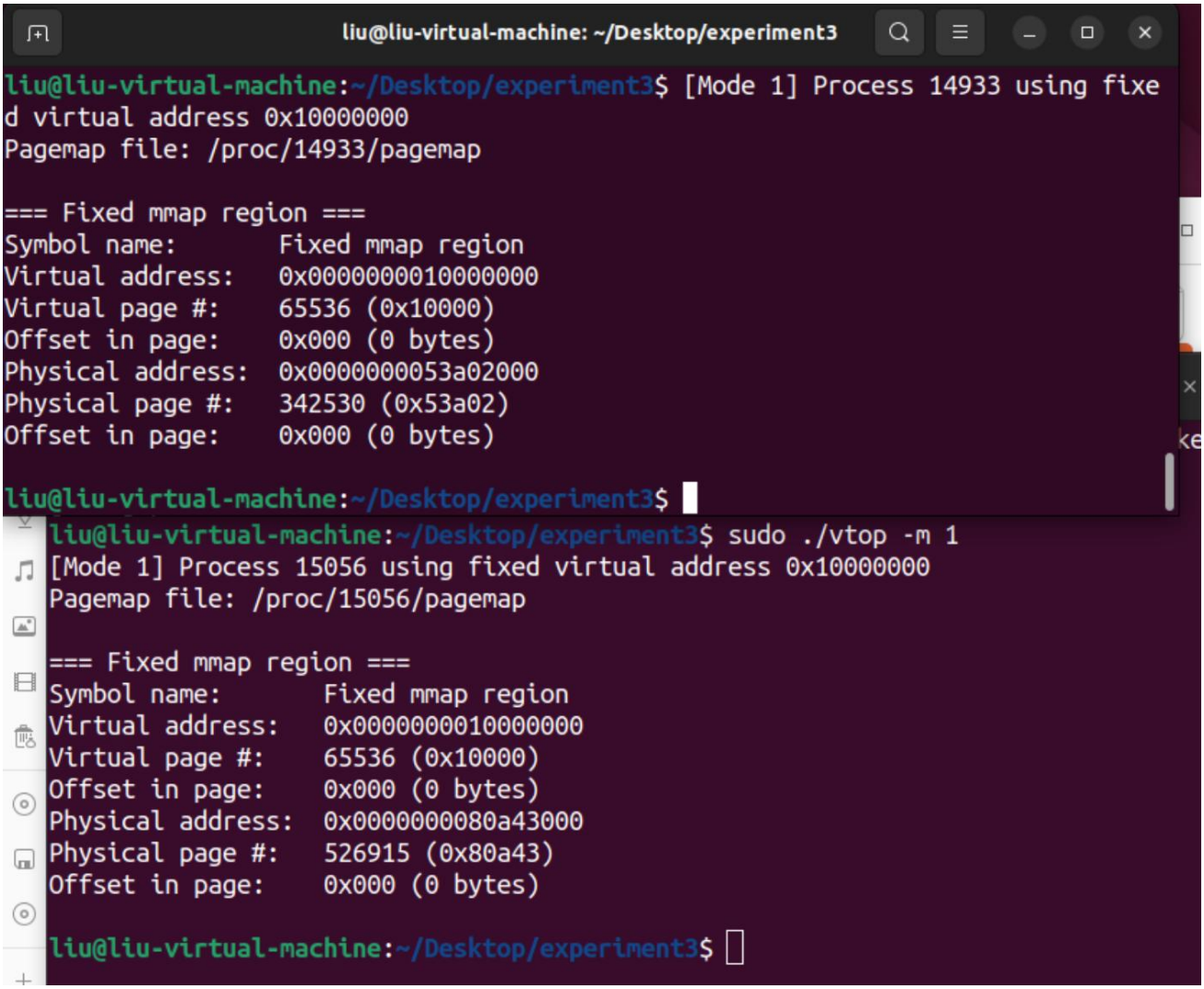

图 3-5 同一虚拟地址对应不同的物理地址

（3）验证不同进程的共享库具有同一的物理地址，如图 3-6 所示，不同进程的共享库可以具有同一的物理地址。

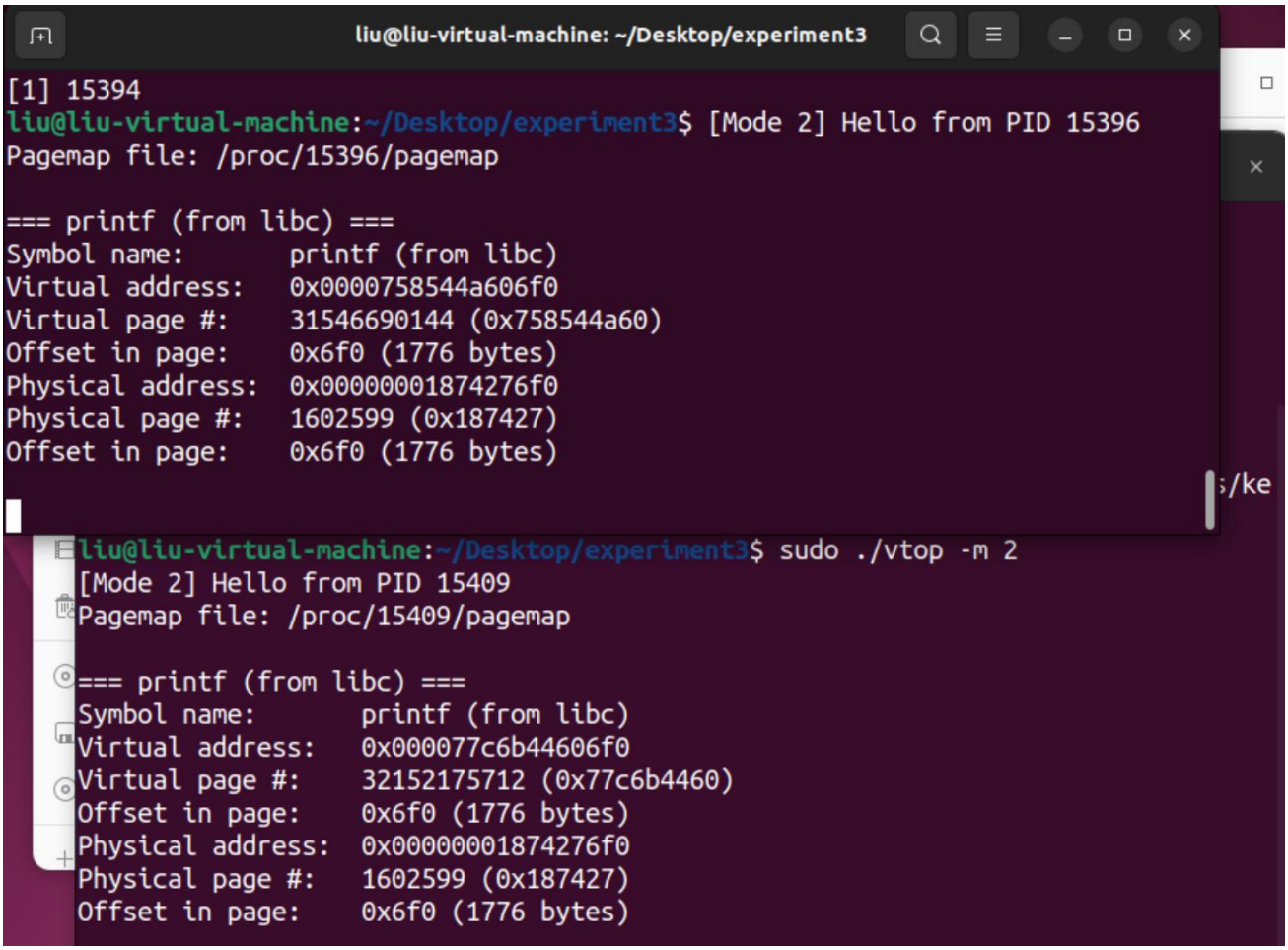

图 3-6 不同进程的共享库具有同一的物理地址

3.2 遇到的问题及解决方法

3.2.1 程序演示页面淘汰算法任务

问题：固定的页框数、页面大小、访问方法不好修改，并且有的会造成缺页率特别高。

解决方法：在代码中通过可调节的局部性因子和多样化的访问模式，实现了对程序局部性的灵活控制，从而避免了单一访问方式导致的极端高缺页率。

3.2.2 验证 Linux 虚拟地址转化为物理地址的机制任务

1.问题：虚拟地址和物理地址都无法显示真实值。

解决方法：用 sudo 运行程序，是由于权限不足造成的问题。

2.问题：扩充实验中的虚拟地址和物理地址显示出现问题，偶尔会有不能满足要求的情况。

解决方法：发现可能是由于如果 ASLR 开启，内核可能不允许在固定的 0x10000000 地址映射。所以在扩充实验任务一关闭 ASLR，即可正确显示。

3.3 设计方案存在的不足

3.3.1 程序演示页面淘汰算法任务

1.访问序列生成过于理想化，虽然通过五种模式模拟了不同程序的执行特征，但与真实应用程序的内存访问模式相比仍显简化，无法完全反映操作系统在实际复杂工作负载下的性能表现。

2.算法实现存在局限性，OPT 算法在实际系统中无法实现，而 LRU 的近似实现未能模拟硬件支持的时间戳或访问位机制，同时未考虑 Belady 异常、工作集模型等高级内存管理概念，使得模拟结果与真实系统存在差距。

3.3.2 验证 Linux 虚拟地址转化为物理地址的机制任务

1.权限依赖过强，必须使用 sudo 运行所有模式，然而实际开发中很少需要直接读取物理地址，限制了实验的可用性。

2.硬编码的 sleep 可能导致竞态条件，没有确保两个进程真正同时运行。

4 实验总结

4.1 实验感想

4.1.1 程序演示页面淘汰算法实验任务感想

通过本次页面置换算法模拟实验，我不仅深入理解了 FIFO、LRU 和 OPT 等经典内存管理策略的工作机制，更从网络安全专业视角重新审视了操作系统内存调度与系统安全之间的深层关联。页面置换算法虽属于传统操作系统范畴，但其设计直接影响系统的性能稳定性与侧信道攻击面。

在实验中，我们观察到：当程序具有强局部性时，LRU 等基于历史行为的算法能显著降低缺页率；而随机性强的访问模式则容易导致频繁换页，加剧系统抖动。这一现象在真实攻击场景中具有重要意义——恶意程序可通过精心构造的内存访问模式人为制造高缺页率，从而实施拒绝服务攻击或干扰关键进程的内存调度。更严重的是，现代高级持续性威胁常利用缓存和页表状态作为隐蔽信道，例如通过监控 LRU 链的变动推断其他进程的敏感操作，这正是“页表侧信道攻击” 的典型手法。

此外，OPT 算法虽在理论上最优，却因需预知未来访问序列而无法在现实中部署，这也揭示了一个重要安全原则：任何依赖全局信息或完美预测的机制都难以在开放、对抗环境中落地。相比之下，FIFO 和 LRU 虽简单，却因其实现透明、行为可预期，反而成为构建可信执行环境时内存管理模块的首选基础。

作为网络空间安全专业的学生，我们应意识到：内存管理不仅是性能优化问题，更是安全边界问题。未来的安全操作系统设计，必须将抗侧信道能力、访问模式混淆与页表隔离强化纳入页面置换机制的核心考量。唯有如此，才能在保障效率的同时，筑牢系统底层的安全防线。

4.1.2 验证 Linux 虚拟地址转化为物理地址的机制实验任务感想

通过本次实验，我深刻体会到虚拟内存机制不仅是操作系统实现多任务隔离与资源高效管理的核心技术，更是现代系统安全体系的重要基石。利用 /proc/<pid>/pagemap 接口将虚拟地址映射到物理地址的过程，表面上是内存管理的底层细节，实则揭示了操作系统在地址空间隔离与资源共享之间的精妙平衡。

从网络安全专业视角来看，这种隔离机制直接构成了进程沙箱的基础：即使恶意程序获知

某一虚拟地址，也无法直接访问其他进程的物理内存，从而有效遏制了跨进程信息窃取与内存破坏攻击。然而，实验中也观察到共享库在多个进程中映射至相同物理页的现象——这虽提升了内存效率，却也潜在引入侧信道攻击风险，攻击者可能借此推断其他进程的执行路径或密钥信息。

此外，若系统配置不当，则可能造成物理地址泄露，削弱 ASLR 等安全机制的有效性。因此，深入理解虚拟-物理地址映射机制，不仅有助于掌握操作系统原理，更对评估系统攻击面、设计内存安全防护策略具有重要现实意义。作为网络空间安全专业的学生，我们应始终以“攻防双重视角”审视底层机制，在理解其功能的同时，警惕其可能被滥用的安全隐患。

4.2 意见和建议

建议可以把这两个任务和实际例子结合起来，增添实际意义。

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：