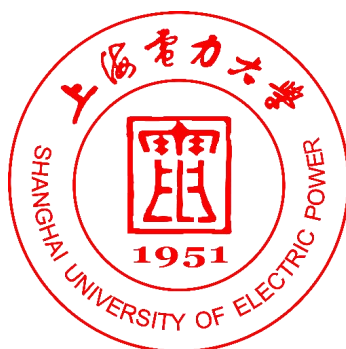


上海电力大学

OpenEuler 基础创新实践

设计文档



参赛队伍名称：_____ 深藏 blue 队

队 伍 成 员：_____ 张清玥、王宇骐、杨心怡

指 导 老 师：_____ 徐曼

全国大学生计算机系统能力大赛

操作系统赛

OS 原理赛道

2024 年 12 月

目录

OpenEuler 基础创新实践	1
1 设计内容及要求	1
2 原理概述	2
2.1 基本概念和原理	2
2.2Linux 源代码分析	5
3 详细设计	8
3.1 数据结构和算法	8
3.2 关键代码	13
3.3 程序运行说明	23
4 测试与结果分析	26
5 总结	36
5.1 设计过程中遇到的问题及解决方法	36
5.2 团队合作情况	42
5.3 设计总结	45

OpenEuler 基础创新实践

摘要:

本课程设计工作旨在通过安装和部署 OpenEuler 操作系统，深入探索并实现操作系统中四个关键领域的功能，并在每个领域内加入创新元素。张清玥负责文件管理和内存管理，实现模拟 TLB（采用自适应替换策略、多级 TLB 和 TLB 预取）、查看大页配置（开发自动化配置工具，动态调整大页分配），以及模拟实现简单文件系统（支持`ls`、`mkdir`、`cd`等命令，引入可视化监控、加密文件、版本控制）。王宇骐负责进程调度，模拟实现单核调度算法（FIFO、SJF、RR）、单队列调度和多队列多核调度，并结合自适应多级反馈队列调度和基于历史行为的学习算法。杨心怡负责进程同步与通信及内存管理优化实验，实现多进程间的同步机制（支持异步数据传输）和内存管理优化（引入内存碎片管理、后台合并算法、崩溃拆分算法，自动合并小页面为大页面，动态拆分大页面以确保系统稳定性）。通过这些任务，我们不仅掌握 OpenEuler 的核心原理和技术，还锻炼了创新能力，培养了解决实际问题的能力。

1 设计内容及要求

张清玥：负责文件管理，内存管理，实现：

- （1）模拟 TLB；
- （2）查看大页配置；
- （3）模拟实现简单文件系统（具有指令 ls、mkdir、cd 等）。

创新点：自适应替换策略、多级 TLB、TLB 预取、可视化监控分布式文件、加密文件、版本控制、自动化配置工具、性能分析、动态调整、跨应用优化。

王宇骐：负责进程调度，实现以下算法：

- （1）模拟实现常见单核调度算法（FIFO、SJF、RR）；
- （2）模拟实现单队列调度；
- （3）模拟实现多队列多核调度。

创新点：自适应多级反馈队列调度 + 基于历史行为的学习

杨心怡：负责实现进程的同步与通信，内存管理优化实验。

- （1）进程的同步与通信；
- （2）内存管理优化实验；

创新点：实现多进程间的同步，支持异步数据传输。在大页面动态分配中加入内存碎片管理算法，后台合并算法，崩溃拆分算法，实现优化内存分配，减少碎

片；自动合并小页面为大页面，提升内存利用率；动态拆分大页面，确保系统稳定性。

2 原理概述

2.1 基本概念和原理

1) TLB 管理

概念与原理：TLB (Translation Lookaside Buffer) 是 CPU 中的一个高速缓存用于存储最近使用的虚拟地址到物理地址的映射。它的主要作用是加速虚拟地址到物理地址的转换过程，从而提高内存访问速度。每当程序访问内存时，CPU 会首先检查 TLB 中是否已经存在该虚拟地址的映射。如果存在（称为 TLB 命中），则可以直接使用物理地址进行内存访问；如果不存在（称为 TLB 错过），则需要通过页表进行查找，并将结果缓存到 TLB 中。

大页 (Huge Pages) 的引入正是为了减少 TLB 错过的次数。普通页面大小通常是 4KB，而大页的大小可以是 2MB 或 1GB。由于大页覆盖的地址范围更大，因此在相同数量的 TLB 条目下，大页可以覆盖更多的内存区域，从而减少了 TLB 错过的概率，提升了内存访问效率。

源码分析：

在 OpenEuler 系统中，TLB 的管理是由内核自动处理的，用户空间程序通常不需要直接操作 TLB。然而，我们可以通过启用大页来间接影响 TLB 的性能。以下是一些与 TLB 相关的操作：

- 启用透明大页 (THP)：THP 是一种自动管理大页的技术，内核会根据应用程序的需求动态分配大页或普通页。你可以通过修改 `/sys/kernel/mm/transparent_hugepage/enabled` 文件来启用或禁用 THP。

- 手动分配大页：如果你希望更精细地控制大页的分配，可以使用 `mmap` 系统调用来请求大页内存。你需要确保系统已经配置了足够的大页，并且应用程序有足够的权限来使用它们。

- TLB 刷新：当内存映射发生变化时（例如，释放内存或改变内存保护），内核会自动刷新 TLB。用户空间程序通常不需要显式刷新 TLB，但可以通过 `msync` 或 `madvise` 等系统调用来影响内存映射的行为。

2) 查看大页配置

概念与原理：

Linux 系统提供了多种方式来查看和管理大页配置。最常用的方法是通过读取 `/proc/meminfo` 文件中的相关信息，或者直接修改 `/proc/sys/vm/nr_hugepages` 文件来设置大页的数量。此外，还可以通过

/sys/kernel/mm/transparent_hugepage/enabled 文件来启用或禁用透明大页(THP)。

3) 文件管理

概念与原理:

文件管理是指对文件系统的操作,包括创建、读取、写入、删除文件等。在 C 语言中,文件操作通常通过标准库函数(如 fopen、fread、fwrite、fclose 等)来实现。对于更底层的文件操作,可以使用系统调用(如 open、read、write、close 等)。

在本项目中,文件管理主要用于读取系统配置文件(如 /proc/meminfo、/proc/loadavg、/sys/kernel/mm/transparent_hugepage/enabled 等),并根据需要修改这些文件的内容。此外,日志记录功能也可以通过文件管理来实现。

4) 模拟实现常见单核调度算法(FIFO、SJF、RR)

FIFO(先来先服务):

概念:按照进程到达的顺序进行调度,即最先到达的进程最先得到 CPU 资源。

原理:所有新到的进程被加入到一个队列中。当 CPU 空闲时,它会从队列头部取出第一个进程执行,直到该进程完成或被其他事件中断。

SJF(最短作业优先):

概念:总是选择预计运行时间最短的进程优先执行。

原理:系统根据进程的预估运行时间来安排执行顺序,具有最短预估运行时间的进程会被优先调度。可以是非抢占式的(一旦开始执行就不被打断),也可以是抢占式的(如果有更短的进程到来,则打断当前正在执行的进程)。

RR(轮转调度):

概念:给每个进程分配一个固定的时间片,按轮次执行。

原理:当一个进程的时间片用完后,即使它还没有完成,也会被暂时挂起并放回队列尾部,以便下一个进程可以获得 CPU 资源。如果进程在此期间已完成,则直接移除。

5) 模拟实现单队列调度

概念:所有进程共享一个队列,依据某种策略(如上述的 FIFO、SJF、RR)从队列中选取下一个要执行的进程。

原理:单队列调度的核心在于如何选择下一个要执行的进程。可以基于不同的调度算法逻辑来决定。例如,在 FIFO 中,始终选择队首的进程;在 SJF 中,需要维护一个能够动态排序的结构,以确保每次都能选出最短的作业;在 RR 中,则需要管理时间片,并在时间片结束时将进程重新插入队列。

6) 模拟实现多队列多核调度

概念:有多个队列,每个队列对应不同的优先级或处理类型,多个 CPU 核心

可以同时从不同队列中取进程执行。

原理：

多队列：每个队列可能有不同的调度策略，比如高优先级队列使用 FIFO，而低优先级队列使用 RR。

多核：多个 CPU 核心可以并发地从各自的队列中取出进程执行。为了防止资源竞争，通常会为每个核心分配独立的调度器，负责从相应的队列中获取进程。

负载均衡：为了保证各核心之间的负载平衡，可能会引入额外的机制来监控各个核心的工作量，并适时调整进程的分配。

7) 利用信号量模拟生产者-消费者

信号量：用于控制多个进程对共享资源访问的特殊变量，它可以用来解决临界区问题，确保同一时间只有一个线程或进程可以访问共享资源。

生产者-消费者问题：描述了两个或更多进程之间共享固定大小缓冲区的情况，其中一个或多个进程（生产者）向缓冲区添加数据，而另一个或多个进程（消费者）从缓冲区移除数据。

8) 通过消息队列实现进程间通信

消息队列：允许进程发送和接收消息。每条消息都有一个类型，接收方可以根据类型选择性地接收消息。它是异步的，即发送者不必等待接收者准备就绪即可发送消息。

9) 内存管理优化实验：THP 的启用

THP 是 openEuler 内核中的一种内存管理优化技术，旨在通过使用更大的页面大小来提高内存访问效率。openEuler 系统使用 4KB 的小页面(Page)，而 THP 则允许内核自动将多个小页面合并为一个较大的页面，通常是 2MB 或 1GB 的大页面(Huge Page)。这种机制可以减少页表项的数量，降低 TLB 的命中次数，从而提高系统的内存访问性能。

动态分配大页面：THP 可以在运行时自动将多个连续的小页面合并为一个大的页面，而无需用户手动配置。当内核检测到某个进程需要大量连续的内存时，它会尝试将这些小页面合并为一个大的页面。

内存碎片整理：THP 还可以通过内存碎片整理(Defragmentation)来确保有足够的连续物理内存来分配大页面。当系统中存在大量小页面时，THP 会尝试将这些小页面重新排列，以便为大页面腾出足够的连续空间。

后台合并：THP 会在后台定期扫描内存，寻找可以合并为大页面的相邻小页面。这个过程是异步进行的，不会影响前台应用程序的性能。

崩溃拆分：当某个大页面中的某个部分被修改时，内核会自动将该大页面拆分为多个小页面。这样可以避免整个大页面的写时复制，提高内存使用的灵活性。

2.2 OpenEuler 源代码分析

1. 进程同步和通信 (IPC)

关键文件和目录:

`ipc/`: 包含所有与进程间通信相关的代码。

`include/linux/ipc.h`: 定义了 IPC 相关的数据结构和函数原型。

`kernel/sys.c`: 实现了部分系统调用接口。

主要功能点:

信号 (Signals):

信号处理的核心在 `kernel/signal.c` 中, 包括信号队列管理、发送和接收信号的逻辑。

系统调用如 `sys_kill()` 和 `sys_tgkill()` 实现了向进程发送信号的功能。

管道 (Pipes) 和命名管道 (FIFOs):

管道的实现位于 `fs/pipe.c`。

命名管道 (FIFO) 的创建和操作由 `fs/namei.c` 中的 `do_fcntl()` 函数处理。

共享内存 (Shared Memory):

共享内存段的创建和管理在 `ipc/shm.c` 中实现。

相关系统调用如 `sys_shmget()`, `sys_shmat()`, `sys_shmdt()` 和 `sys_shmctl()` 定义在 `kernel/sys.c` 中。

信号量 (Semaphores):

信号量的操作在 `ipc/sem.c` 中实现, 包括创建、获取、释放等。

系统调用如 `sys_semget()`, `sys_semop()`, `sys_semctl()` 也在 `kernel/sys.c` 中。

消息队列 (Message Queues):

消息队列的实现位于 `ipc/msg.c`。

系统调用如 `sys_msgget()`, `sys_msgsnd()`, `sys_msgrcv()`, `sys_msgctl()` 同样定义在 `kernel/sys.c` 中。

套接字 (Sockets):

套接字的实现分散在多个文件中, 核心文件包括 `net/socket.c` 和 `net/unix/af_unix.c` (用于本地套接字)。

系统调用如 `sys_socket()`, `sys_bind()`, `sys_connect()`, `sys_sendmsg()`, `sys_recvmsg()` 等在 `net/socket.c` 中实现。

2. 进程调度

关键文件和目录:

`kernel/sched/`: 包含了调度器的主要实现。

`include/linux/sched/`: 定义了与调度相关的数据结构和宏。

主要功能点：

调度器实现：

CFS（完全公平调度器）的核心逻辑在 `kernel/sched/fair.c` 中。

实时调度策略（`SCHED_FIFO`, `SCHED_RR`, `SCHED_DEADLINE`）的实现分别在 `kernel/sched/rt.c` 和 `kernel/sched/deadline.c` 中。

调度类：

调度类的概念允许不同类型的进程使用不同的调度算法。每个调度类都有自己的 `enqueue_task()` 和 `dequeue_task()` 方法。

主要调度类包括 `fair_sched_class`（CFS）、`rt_sched_class`（实时调度）和 `dl_sched_class`（截止时间调度），这些都在 `kernel/sched/core.c` 中定义。

负载均衡：

负载均衡机制确保任务在多处理器系统中均匀分布。相关代码可以在 `kernel/sched/fair.c` 和 `kernel/sched/balancer.c` 中找到。

系统调用：

调度相关的系统调用如 `sys_sched_yield()`, `sys_sched_setscheduler()`, `sys_sched_getscheduler()` 等在 `kernel/sched/core.c` 中实现。

3. 内存管理

关键文件和目录：

`mm/`：包含了内存管理子系统的大部分代码。

`include/linux/mm.h`：定义了内存管理相关的数据结构和函数原型。

主要功能点：

页分配器：

页分配器负责物理页面的分配和回收，其实现在 `mm/page_alloc.c` 中。

包括伙伴系统（buddy system）和 slab 分配器，后者用于小对象的分配，实现在 `mm/slub.c` 或 `mm/slab.c` 中。

虚拟内存管理：

虚拟内存的映射和管理在 `mm/mmap.c` 中实现，涉及文件映射、匿名映射等功能。

页表管理在 `arch/*/mm/` 下的相关架构文件中实现，因为页表的具体格式依赖于硬件架构。

内存回收：

页面回收和交换机制在 `mm/vmscan.c` 中实现，包括直接内存回收和后台内存回收两种模式。

交换空间的管理在 `mm/swap_state.c` 和 `mm/swapfile.c` 中。

内存屏障:

内存屏障的实现涉及到 CPU 架构, 通常在 `arch/*/include/asm/barrier.h` 中定义。

NUMA 支持:

NUMA (非一致内存访问) 的支持在 `mm/mempolicy.c` 和 `mm/memory_hotplug.c` 中实现, 包括内存节点的管理和热插拔。

4. 文件系统

关键文件和目录:

`fs/`: 包含了所有文件系统的实现。

`include/linux/fs.h`: 定义了文件系统相关的数据结构和函数原型。

主要功能点:

VFS 层:

VFS (虚拟文件系统) 层提供了统一的文件系统接口, 其核心实现位于 `fs/dcache.c` (目录缓存)、`fs/inode.c` (索引节点管理) 和 `fs/file_table.c` (文件描述符表)。

超级块和索引节点:

超级块和索引节点的管理分别在 `fs/super.c` 和 `fs/inode.c` 中实现。

每个文件系统都有自己的超级块和索引节点结构, 例如 `fs/ext4/super.c` 和 `fs/ext4/inode.c`。

文件操作:

文件的打开、读取、写入、关闭等操作在 `fs/read_write.c` 中实现。

特定文件系统的文件操作则在各自文件系统的目录下实现, 如 `fs/ext4/file.c`。

目录遍历:

目录遍历和路径解析的逻辑在 `fs/namei.c` 中实现, 包括查找路径、创建新文件等。

日志文件系统:

日志文件系统的实现以 `ext4` 为例, 其日志记录和恢复机制在 `fs/ext4/journal.c` 中实现。

网络文件系统 (NFS):

NFS 客户端的实现位于 `fs/nfs/` 目录下, 包括 `fs/nfs/client.c` 和 `fs/nfs/inode.c`。

分布式文件系统 (如 CephFS):

分布式文件系统的实现通常更加复杂, 以 CephFS 为例, 其实现位于 `fs/ceph/` 目录下, 涵盖了元数据服务器通信、数据存储和检索等功能。

3 详细设计

3.1 数据结构和算法

(1) TLB (Translation Lookaside Buffer, 转换检测缓冲区)

1) 数据结构

TLBEntry (TLB 表项结构):

包含 `uint64_t` 类型的 `virtual_page` (虚拟页号)、`physical_page` (物理页号), `bool` 类型的 `valid` (有效位)、`dirty` (脏位) 以及 `uint64_t` 类型的 `last_access` (最后访问时间) 字段。

TLB (TLB 缓存结构):

有 `TLBEntry *` 类型的 `entries` (TLB 表项数组)、`int` 类型的 `size` (TLB 大小) 和 `replacement_policy` (替换策略) 字段。

2) 算法

TLB 查找:

并行搜索所有 TLB 表项。

比较虚拟页号。

返回物理页号或 TLB 缺失。

TLB 替换:

支持 LRU (最近最少使用) 策略、FIFO (先进先出) 策略、随机替换策略。

(2) 大页管理

1) 数据结构

HugePage (大页面描述符):

包含 `uint64_t` 类型的 `base_addr` (基地址)、`size_t` 类型的 `size` (页面大小)、`bool` 类型的 `allocated` (分配状态) 以及 `struct list_head` 类型的 `list` (链表节点) 字段。

HugePageManager (大页面管理器):

包含 `struct list_head` 类型的 `free_pages` (空闲页面链表)、`used_pages` (已使用页面链表), `int` 类型的 `total_pages` (总页面数) 以及 `pthread_mutex_t` 类型的 `lock` (互斥锁) 字段。

2) 核心算法

大页面分配:

采用首次适配算法、最佳适配算法。

页面分裂与合并:

涉及内存规整、碎片整理、页面迁移、连续空间合并操作。

(3) 文件管理

1) 数据结构

FileDescriptor (文件描述符) :

包含 int 类型的 fd (文件句柄)、char *类型的 path (文件路径)、uint64_t 类型的 size (文件大小) 以及 uint32_t 类型的 permissions (访问权限) 字段。

FileCache (文件缓存) :

包含 char *类型的 buffer (缓存数据)、size_t 类型的 size (缓存大小)、time_t 类型的 last_access (最后访问时间) 以及 bool 类型的 dirty (脏标志) 字段。

2) 核心算法

文件操作:

涵盖读写缓冲、预读取、延迟写入操作。

缓存管理:

运用 LRU 缓存替换、写回策略、缓存一致性维护策略。

(4) CPU 调度算法相关

(一) FIFO (先来先服务) 调度算法

1) 数据结构

定义 Process 结构体, 包含 id (进程唯一标识符)、arrival_time (进程到达系统的时间)、burst_time (进程需要的 CPU 执行时间)、start_time (进程开始执行的时间)、completion_time (进程完成执行的时间)、turnaround_time (周转时间)、waiting_time (等待时间) 等字段。

2) 核心算法

排序: 使用 qsort 函数和自定义比较函数 compare 按 arrival_time 升序排序进程。

调度与计算: 初始化 current_time 为 0, 遍历排序后的进程列表, 按需更新 current_time, 记录进程的 start_time、completion_time 等信息, 并计算 turnaround_time 和 waiting_time。

输出结果: 遍历所有进程打印相关信息。

(二) SJF (最短作业优先) 调度算法

1) 数据结构

同样基于 Process 结构体, 包含类似上述的多个字段来描述进程相关属性。

2) 核心算法

有 compare_arrival 用于按到达时间升序排序, compare_burst 用于按执行时间升序排序 (用于输出结果排序)。

sjfScheduling 函数中, 先按 arrival_time 排序, 初始化 current_time, 遍历进程列表设置各时间属性, 按需更新 current_time 并计算相关时间, 最后再按

burst_time 排序以便输出结果。

printProcesses 函数用于打印所有进程详细信息。

main 函数定义测试用例并调用 sjfScheduling 模拟调度过程。

（三）RR（轮转调度）算法

1）数据结构

Process 结构体在上述基础上新增 remaining_time（进程剩余需要执行的时间）字段。

2）核心算法

初始化和排序：用 qsort 和 compare_arrival 按到达时间升序排序进程，初始化 remaining_time。

模拟调度过程：用循环模拟时间推进和进程调度，处理单个进程时根据 remaining_time 和 time_quantum 决定执行时间，更新相关时间属性，通过特定方式选择下一个进程实现循环调度。

输出结果：所有进程完成后调用 printProcesses 打印详细信息。

（四）单队列调度

1）数据结构

Process 结构体表示进程，Node 和 Queue 结构体用于创建队列管理就绪队列中的进程。

2）辅助函数

包含 createQueue、enqueue、dequeue、isEmpty 用于队列操作，compare_arrival 和 compare_burst 为 qsort 提供比较逻辑。

3）调度算法实现

实现了 fifoScheduling（FIFO 调度算法）、sjfScheduling（非抢占式 SJF 调度算法）、rrScheduling（RR 调度算法）。

4）输出结果

printProcesses 函数用于打印进程调度后的详细信息。

5）主函数

创建测试用例并调用上述调度算法，打印结果以供比较。

（五）多队列多核调度

1）数据结构

Process 结构体包含 id、arrival_time、burst_time、remaining_time、priority 等多个字段描述进程属性。

Node 结构体用于链表实现队列，Queue 结构体作为队列保存就绪进程，CPU 结构体代表多核系统中的单个处理器核心，包含 id、ready_queue、current_process、

`current_time` 等字段。

2) 算法描述

初始化：初始化 CPU 核心及就绪队列，对进程按 `arrival_time` 排序，设置 `remaining_time`。

调度循环：用全局变量 `current_time` 跟踪系统时间，检查新到达进程并分配到合适核心的就绪队列，遍历核心选择最高优先级进程执行并更新时间戳，更新 `current_time` 到下一个事件发生时间点。

输出结果：遍历进程打印详细信息。

清理资源：释放动态分配的内存。

3) 关键点

优先级调度，入队时保证队列按优先级排序。

多核支持，为每个核心维护独立就绪队列。

通过 `current_time` 跟踪系统时间，避免无限循环。

(六) 自适应多级反馈队列调度 + 基于历史行为的学习

1) 数据结构

`Process` (进程结构体) 包含 `id`、`queue_level` (当前队列级别)、`remaining_time`、`avg_execution_time`、`std_dev_execution_time` 等多个与进程历史行为及状态相关的字段。

`Queue` (队列结构体) 包含 `processes` (存储进程的数组)、`size` (队列中当前进程数量) 以及多个不同优先级的 `queues` 数组。

2) 核心算法

初始化：通过 `init_queues` 初始化所有队列。

队列操作：有 `enqueue` 将进程加入指定优先级队列，`dequeue` 移除并返回指定优先级队列队首进程。

统计更新：`update_process_statistics` 更新进程历史行为统计数据。

动态调整优先级：`adjust_queue_position` 根据进程历史行为调整其在队列中的位置。

进程创建：`create_process` 创建新进程并初始化属性。

进程执行：`execute_process` 模拟 CPU 执行进程并更新相关统计数据。

调度器主循环：`scheduler` 负责遍历队列选择进程执行并调整优先级，结束条件为所有进程完成。

(5) 信号量、消息队列及内存管理相关操作

1) 关键数据结构

union semun:

用于 `semctl` 系统调用的参数，包含 `val`（设置信号量值时用）、`buf`（获取或设置信号量集状态时用）、`array`（初始化信号量数组时用）、`__buf`（内部使用）成员。

struct sembuf:

定义信号量操作，由 `semop` 系统调用来执行 P 操作或 V 操作，包含 `sem_num`（信号量编号）、`sem_op`（操作类型）、`sem_flg`（操作标志）字段。

struct msg_buffer:

消息缓冲区结构体，包含 `msg_type`（消息类型）、`msg_text`（存储消息内容的字符数组，最大长度 100 字符）字段。

Page 结构体:

包含 `is_free` 参数（1 表示空闲，0 表示已分配）表示页面状态。

page_t 结构体:

包含 `id`（页面唯一标识符）和 `status`（使用枚举定义的页面状态，如 `FREE`、`USED`、`SWAPPED`）字段。

vma_t 结构体:

包含 `start_address`（虚拟内存区域起始地址）、`end_address`（虚拟内存区域结束地址）、`huge_page_count`（已合并大页面数量）字段。

2) 核心算法

producer.c & consumer.c 相关操作:

`init_semaphore`: 初始化信号量，设置初始值。

`del_semaphore`: 删除信号量。

`p_semaphore`: 执行 P 操作（等待操作）。

`v_semaphore`: 执行 V 操作（发送操作）。

sender.c & receiver.c 相关操作:

`ftok`: 生成用于消息队列的唯一键值。

`msgget`: 获取或创建消息队列标识符。

`fgets`: 从标准输入读取消息内容到缓冲区。

`msgsnd`: 将消息发送到消息队列。

`msgrcv`: 从消息队列接收消息。

`msgctl`: 控制消息队列，如删除操作。

dynamic_allocation.c 相关操作:

`alloc_hugepage`: 遍历查找空闲大页面并分配，返回起始地址或 `NULL`。

`alloc_small_page`: 通过 `mmap` 分配小页面，成功返回起始地址，失败返回

MAP_FAILED 并打印错误。

free_page: 遍历查找匹配地址的页面并标记为空闲。

free_small_page: 用 munmap 解除小页面内存映射释放内存。

memory_fragmentation.c 相关操作:

page_swap: 将指定页面从内存交换到磁盘交换空间, 空间满则报错。

reclaim_pages_to_swap: 遍历内存将 USED 状态页面交换到交换空间腾出内存。

release_swap_space: 释放交换空间页面并更新页面状态。

allocate_huge_page: 尝试分配大页面, 必要时回收页面腾出空间再分配。

backend_merge.c 相关操作:

collapse_huge_page: 尝试合并连续 USED 状态页面为大页面, 符合条件则更新页面状态及相关结构信息。

khugepaged: 模拟后台合并线程遍历页面合并大页面。

collapse_splitting.c 相关操作:

create_huge_page: 将连续页面标记为合并大页面状态。

split_huge_page: 尝试拆分大页面为小页面, 有足够空闲页面时进行操作。

3.2 关键代码

一、信号量操作 (consumer.c & producer.c)

```
// 信号量 P 操作 (等待) static int p_semaphore(int sem_id) {  
    struct sembuf sem_b = { 0, -1, SEM_UNDO }; // 操作编号 0, 减少信号  
    量值 1, 使用 SEM_UNDO 标志  
    return semop(sem_id, &sem_b, 1) == -1? (printf("信号量 P 操作失败!!!  
\\n"), 0): 1; // 成功返回 1, 失败返回 0 并打印错误信息}  
// 信号量 V 操作 (信号) static int v_semaphore(int sem_id) {  
    struct sembuf sem_b = { 0, 1, SEM_UNDO }; // 操作编号 0, 增加信号  
    量值 1, 使用 SEM_UNDO 标志  
    return semop(sem_id, &sem_b, 1) == -1? (printf("信号量 P 操作失败!!!  
\\n"), 0): 1; // 成功返回 1, 失败返回 0 并打印错误信息}
```

二、消息队列操作 (sender.c & receiver.c)

// 获取消息队列

```
msgid = msgget(key, 0666 | IPC_CREAT);
```

```
// 接收消息 msgrcv(msgid, &message, sizeof(message), 1, 0);
```

```
// 创建消息队列
```

```

msgid = msgget(key, 0666 | IPC_CREAT);
// 发送消息到消息队列 msgsnd(msgid, &message, sizeof(message), 0);
三、内存动态分配相关 (dynamic_allocation.c)
// 分配大页面 void* alloc_hugepage(PageInfo *pages, size_t num_pages) {
    for (size_t i = 0; i < num_pages; i++) {
        if (pages[i].is_free) {
            pages[i].is_free = 0;
            return pages[i].addr;
        }
    }
    return NULL;}

// 分配小页面 (这里仅作演示, 实际应用中应避免频繁分配小页面) void*
alloc_small_page() {
    void *addr = mmap(NULL, SMALL_PAGE_SIZE, PROT_READ |
PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED) {
        perror("Failed to mmap small page");
        return NULL;
    }
    return addr;}

四、内存碎片整理相关 (memory_fragmentation.c)
// 分配大页面 void allocate_huge_page() {
    int start_index = find_free_continuous_pages(HUGE_PAGE_SIZE);
    if (start_index != -1) {
        printf("Found %d continuous free pages starting at index %d\n",
HUGE_PAGE_SIZE, start_index);
        for (int i = start_index; i < start_index + HUGE_PAGE_SIZE; i++) {
            memory[i].status = USED;
        }
        return;
    }
    printf("Not enough continuous free pages. Trying to swap some pages...\n");
    reclaim_pages_to_swap(); // 回收页面到交换空间
    release_swap_space();    // 释放交换空间

```



```

start_index = find_free_continuous_pages(HUGE_PAGE_SIZE);
if (start_index != -1) {
    printf("Found %d continuous free pages starting at index %d after
swapping\n", HUGE_PAGE_SIZE, start_index);
    for (int i = start_index; i < start_index + HUGE_PAGE_SIZE; i++) {
        memory[i].status = USED;
    }
    return;
}

printf("Failed to allocate huge page due to insufficient free space.\n");}

// 页面交换 void page_swap(int page_index) {
    if (memory[page_index].status == USED) {
        for (int i = 0; i < SWAP_SIZE; i++) {
            if (swap_space[i].status == FREE) {
                swap_space[i].status = USED; // 标记交换空间为已使用
                memory[page_index].status = SWAPPED; // 标记内存页面
为已交换

                printf("Page %d swapped to swap space at index %d\n",
page_index, i);

                return;
            }
        }
    }
}
}
```

五、后台页面合并相关 (backend merge.c)

// 合并小页面为大页面

```
bool collapse_huge_page(int start_index, vma_t *vma) {
    // 检查是否可以合并

    for (int i = 0; i < HUGE_PAGE_SIZE; i++) {
        if (start_index + i >= MAX_PAGES || page_table[start_index +
i].status!= USED) {
            return false; // 不可以合并
        }
    }
    // 合并操作
}
```

```

    for (int i = 0; i < HUGE_PAGE_SIZE; i++) {
        page_table[start_index + i].status = HUGE; // 标记为大页面
    }
    // 更新 VMA 结构
    vma->huge_page_count += 1;
    vma->end_address += HUGE_PAGE_SIZE - 1; // 更新 VMA 的结束地
址

    printf("Collapsed pages from %d to %d into a huge page. Updated VMA:
start=%d, end=%d, huge_page_count=%d\n",
           start_index, start_index + HUGE_PAGE_SIZE - 1,
vma->start_address, vma->end_address, vma->huge_page_count);
    return true; // 合并成功}
// 后台合并线程模拟 void khugepaged(vma_t *vma) {
    printf("Starting backend merge process...\n");
    for (int i = 0; i <= MAX_PAGES - HUGE_PAGE_SIZE; i++) {
        if (collapse_huge_page(i, vma)) {
            // 合并成功后继续扫描
            i += HUGE_PAGE_SIZE - 1; // 跳过已合并的页面
        }
    }
}

```

六、大页面拆分相关 (collapse_splitting.c)

```

// 拆分大页面 void split_huge_page(int start_index) {
    printf("Splitting huge page at index %d...\n", start_index);
    // 首先判断是否有足够的空闲页面来拆分
    if (find_free() >= HUGE_PAGE_SIZE) {
        page_table[start_index].status = USED; // 将第一个小页面标记为
USED
        for (int i = 1; i < HUGE_PAGE_SIZE; i++) {
            int free_index = find_freeindex();
            if (free_index != -1) { // 确保找到了空闲页面
                page_table[free_index].status = USED; // 将大页面拆分为
小页面
            }
        }
    }
}

```

```

    } else {
        printf("Not enough FREE pages\n");
    }
    printf("Split complete. New memory status:\n");
    print_memory_status();}

```

七、FIFO 调度算法

```
#include <stdio.h>#include <stdlib.h>
```

```
// 定义进程结构，用于表示每个进程的属性 typedef struct {
```

```

    int id;                // 进程的唯一标识符
    int arrival_time;      // 进程到达系统的时间
    int burst_time;        // 进程需要的 CPU 执行时间
    int start_time;        // 进程开始执行的时间
    int completion_time;   // 进程完成执行的时间
    int turnaround_time;   // 周转时间 = 完成时间 - 到达时间
    int waiting_time;      // 等待时间 = 周转时间 - CPU 执行时间}

```

```
Process;
```

```

// 比较函数，用于 qsort 排序。按照到达时间升序排列进程。int compare(const
void *a, const void *b) {

```

```

    return ((Process *)a)->arrival_time - ((Process *)b)->arrival_time;}

```

```
void fifoScheduling(Process processes[], int n) {
```

```

    // 使用 qsort 对所有进程按到达时间进行升序排序
    qsort(processes, n, sizeof(Process), compare);

```

```

    int current_time = 0; // 当前时间，跟踪模拟的时钟

```

```

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

```

```

        // 如果当前时间小于进程的到达时间，则更新当前时间为进程的到

```

达时间

```

        if (current_time < processes[i].arrival_time) {

```

```

            current_time = processes[i].arrival_time;

```

```

        }

```

```

        // 设置进程的开始时间为当前时间

```

```

        processes[i].start_time = current_time;

```

```

        // 更新当前时间为当前时间加上进程的 burst time

```

```

        current_time += processes[i].burst_time;

```

```

        // 设置进程的完成时间为当前时间
        processes[i].completion_time = current_time;
        // 计算并设置周转时间
        processes[i].turnaround_time = processes[i].completion_time -
processes[i].arrival_time;
        // 计算并设置等待时间
        processes[i].waiting_time = processes[i].turnaround_time -
processes[i].burst_time;
    }

    // 输出结果：打印每个进程的相关信息
    printf("Process      ID\tArrival      Time\tBurst      Time\tCompletion
Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; ++i) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
            processes[i].id, processes[i].arrival_time,
            processes[i].burst_time, processes[i].completion_time,
            processes[i].turnaround_time, processes[i].waiting_time);
    }
}

int main() {
    // 定义一组进程，每个进程包含 ID、到达时间和 CPU 需求时间（burst
time）

    Process processes[] = {{1, 0, 5}, {2, 2, 3}, {3, 5, 2}};
    // 计算进程数组的大小
    int n = sizeof(processes) / sizeof(processes[0]);

    // 调用 FIFO 调度函数处理这些进程
    fifoScheduling(processes, n);

    return 0;}

```

八、SJF（最短作业优先）调度算法

```
#include <stdio.h>#include <stdlib.h>
```

```
// 定义进程结构 typedef struct {
    int id;                // 进程标识符
```

```

int arrival_time;    // 到达时间
int burst_time;      // 执行时间
int start_time;      // 开始时间
int completion_time; // 完成时间
int turnaround_time; // 周转时间
int waiting_time;    // 等待时间} Process;

// 比较函数,用于 qsort 排序 int compare_arrival(const void *a, const void *b) {
    return ((Process *)a)->arrival_time - ((Process *)b)->arrival_time;}

// SJF 调度算法实现 void sjfScheduling(Process processes[], int n) {
    // 首先按到达时间排序
    qsort(processes, n, sizeof(Process), compare_arrival);

    int current_time = 0;
    int completed = 0; // 已完成的进程数

    while (completed < n) {
        int min_burst_index = -1;
        int min_burst = INT_MAX;

        // 寻找下一个要执行的进程 (已到达且 burst_time 最小)
        for (int i = 0; i < n; ++i) {
            if (processes[i].arrival_time <= current_time &&
processes[i].start_time == 0) {
                if (processes[i].burst_time < min_burst) {
                    min_burst = processes[i].burst_time;
                    min_burst_index = i;
                }
            }
        }

        if (min_burst_index != -1) { // 如果找到了可执行的进程
            Process *p = &processes[min_burst_index];
            p->start_time = current_time;
            p->completion_time = current_time + p->burst_time;

```

```

        p->turnaround_time = p->completion_time - p->arrival_time;
        p->waiting_time = p->turnaround_time - p->burst_time;

        current_time = p->completion_time;
        completed++;
    } else {
        // 如果没有找到任何可以执行的进程, 则推进时间直到有新的
进程到达

        for (int i = 0; i < n; ++i) {
            if (processes[i].arrival_time > current_time) {
                current_time = processes[i].arrival_time;
                break;
            }
        }
    }
}

// 打印进程信息 void printProcesses(Process processes[], int n) {
    printf("Process  ID\tArrival  Time\tBurst  Time\tStart  Time\tCompletion
Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; ++i) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
            processes[i].id,
            processes[i].arrival_time,
            processes[i].burst_time,
            processes[i].start_time,
            processes[i].completion_time,
            processes[i].turnaround_time,
            processes[i].waiting_time);
    }
}

int main() {
    // 创建一些测试用例
    Process processes[] = {
        {1, 0, 5},
        {2, 2, 3},

```

```

        {3, 5, 2},
        {4, 6, 4},
        {5, 8, 1}
    };

    int n = sizeof(processes) / sizeof(processes[0]);

    // 调用 SJF 调度算法
    sjfScheduling(processes, n);

    // 输出结果
    printProcesses(processes, n);

    return 0;}

```

九、RR（轮转调度）算法

```

#include <stdio.h>#include <stdlib.h>
// 定义进程结构 typedef struct {
    int id;                // 进程标识符
    int arrival_time;      // 到达时间
    int burst_time;        // 执行时间
    int remaining_time;    // 剩余执行时间（用于 RR 调度）
    int start_time;        // 开始执行的时间点
    int completion_time;   // 完成执行的时间点
    int turnaround_time;   // 周转时间 = 完成时间 - 到达时间
    int waiting_time;      // 等待时间 = 周转时间 - 执行时间} Process;
// 比较函数，用于 qsort 排序，按照到达时间升序排列进程 int
compare_arrival(const void *a, const void *b) {
    return ((Process *)a)->arrival_time - ((Process *)b)->arrival_time;}
void rrScheduling(Process processes[], int n, int time_quantum) {
    // 首先按到达时间对进程进行升序排序，确保按照到达顺序处理。
    qsort(processes, n, sizeof(Process), compare_arrival);

    // 初始化每个进程的剩余执行时间为它们的总执行时间（burst_time）。
    for (int i = 0; i < n; ++i) {
        processes[i].remaining_time = processes[i].burst_time;
    }
}

```

```

    }

    int current_time = 0; // 当前系统时间
    int completed = 0;    // 已完成的进程数
    int index = 0;        // 当前正在处理的进程索引

    // 循环直到所有进程都已完成
    while (completed != n) {
        Process *current_process = &processes[index];

        // 如果当前进程还有剩余时间需要执行
        if (current_process->remaining_time > 0) {
            // 如果当前时间小于进程的到达时间, 则将当前时间设置为进
            // 程的到达时间
            if (current_time < current_process->arrival_time) {
                current_time = current_process->arrival_time;
            }

            // 如果这是进程第一次开始执行, 记录其开始时间
            if (current_process->start_time == 0) {
                current_process->start_time = current_time;
            }

            // 如果剩余时间大于时间片, 则只执行一个时间片
            if (current_process->remaining_time > time_quantum) {
                current_time += time_quantum;
                current_process->remaining_time -= time_quantum;
            } else { // 否则执行完剩下的时间并标记该进程完成
                current_time += current_process->remaining_time;
                current_process->completion_time = current_time;
                current_process->turnaround_time =
                    current_process->completion_time - current_process->arrival_time;
                current_process->waiting_time =
                    current_process->turnaround_time - current_process->burst_time;
            }
        }
    }
}

```



```

        current_process->remaining_time = 0;
        completed++; // 增加已完成的进程计数
    }
}

// 移动到下一个进程，使用模运算实现循环队列
index = (index + 1) % n;

// 确保在所有进程都已经至少尝试过一次后才结束循环
// 这是防止某些进程可能因为时间片分配而永远得不到 CPU 时间
的情况

if (completed == n || (index == 0 && completed == 0)) break;
}

```

十、单队列调度

```

#include <stdio.h>#include <stdlib.h>
// 定义进程结构体，包含进程的所有相关信息 typedef struct Process {

```

3.3 程序运行说明

1. 程序运行环境及通用说明

运行环境要求：

操作系统：OpenEuler

编译器：GCC 9.0

依赖库：

OpenSSL（用于 filem.c 的加密功能）

pthread（用于 hugepage_manager.c 的线程同步）

2. 各源程序文件相关说明

源文件：

TLBm.c: TLB 管理器实现

hugepage_manager.c: 大页面管理器实现

filem.c: 文件管理系统实现

3. 编译指令说明

编译 TLBm.c:

```
gcc -o tlbm TLBm.c -O2
```

编译 hugepage_manager.c:

```
gcc -o hugepagem hugepage_manager.c -lpthread
```

编译 filem.c: `gcc -o filem filem.c -lssl -lcrypto`

4. 不同模块执行文件及运行方式说明

4.1 consumer.c 与 producer.c

编译指令:

```
gcc consumer.c -o consumer
```

```
gcc producer.c -o producer
```

运行方式:

```
./consumer & ./producer
```

4.2 sender.c 与 receiver.c

编译指令:

```
gcc sender.c -o sender
```

```
gcc receiver.c -o receiver
```

运行方式: 先执行 ./sender 输入内容, 再执行 ./receiver 读出内容。

4.3 dynamic_allocation.c

运行前准备:

```
# 确保系统中已经配置了足够的大页面 sudo sysctl -w vm.nr_hugepages=100#  
确保 hugetlbfs 已正确挂载到 /mnt/hugesudo mkdir -p /mnt/hugesudo mount -t  
hugetlbfs nodev /mnt/huge
```

编译指令:

```
gcc dynamic_allocation.c -o dynamic_allocation
```

运行方式:

```
sudo ./dynamic_allocation
```

4.4 memory_fragmentation.c

编译指令:

```
gcc memory_fragmentation.c -o mf
```

运行方式: ./mf

4.5 backend_merge.c

编译指令:

```
gcc backend_merge.c -o bm
```

运行方式: `./bm`

4.6 collapse_splitting.c

编译指令:

`bash`

`gcc collapse_splitting.c -o cs`

运行方式: `./cs`

4.7 TLB 相关 (tlbm)

执行文件: `tlbm`

运行方式: `./tlbm`

运行输出示例:

`plaintext`

TLB Statistics:

Access Count: 10000

Hit Count: 8234

Miss Count: 1766

Hit Rate: 82.34%

4.8 大页管理相关 (hugepagem)

执行文件: `hugepagem`

运行方式: 需要 `root` 权限运行, 命令格式如下:

`sudo./hugepagem [命令选项]`

可用命令选项:

`auto`: 自动调整大页面

`benchmark`: 运行性能测试

`daemon`: 作为后台服务运行

`suggest`: 获取优化建议

示例:

`sudo./hugepagem auto`

4.9 文件管理相关 (filem)

执行文件: `filem`

运行方式: 直接运行进入交互式命令行, 可用命令如下: `./filem`

可用命令:

`ls`: 列出当前目录文件

`mkdir <目录名>`: 创建新目录

`cd <目录名>`: 切换目录

add <文件名> < 内容 >: 添加新文件

history <文件名>: 查看文件历史版本

rollback <文件名> < 时间戳 >: 回滚到指定版本

exit: 退出程序

5. 注意事项

hugepage_manager 需要 root 权限才能修改系统大页面设置。

filem 程序使用 OpenSSL 进行加密，需要确保系统已安装 OpenSSL 开发库：sudo apt-get install libssl-dev

编译时可能需要的其他选项：

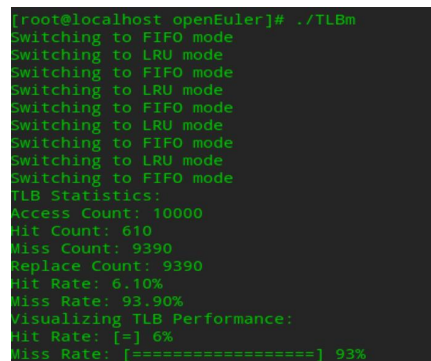
调试版本：添加-g 选项

优化版本：添加-O2 或-O3 选项

警告检查：添加-Wall -Wextra 选项

4 测试与结果分析

(1) TLB



```
[root@localhost openEuler]# ./TLBm
Switching to FIFO mode
Switching to LRU mode
Switching to FIFO mode
Switching to LRU mode
Switching to FIFO mode
Switching to LRU mode
Switching to FIFO mode
Switching to LRU mode
Switching to FIFO mode
Switching to LRU mode
Switching to FIFO mode
TLB Statistics:
Access Count: 10000
Hit Count: 610
Miss Count: 9390
Replace Count: 9390
Hit Rate: 6.10%
Miss Rate: 93.90%
Visualizing TLB Performance:
Hit Rate: [=] 6%
Miss Rate: [=====] 93%
```

图 1 TLB 运行结果

1. 程序运行情况

在程序运行过程中，有多次在“LRU mode”（最近最少使用模式）和“FIFO mode”（先进先出模式）之间切换的记录，如“Switching to LRU mode”和“Switching to FIFO mode”交替出现。这表明该程序可能在测试或模拟不同的缓存替换策略。

2. TLB 统计信息

程序运行结束后，输出了“TLB Statistics”（TLB 统计信息）：

“Access Count: 10000”表示总的访问次数为 10000 次。

“Hit Count: 610”表示命中次数为 610 次。

“Miss Count: 9390”表示未命中次数为 9390 次。

“Replace Count: 9390” 表示替换次数为 9390 次。

“Hit Rate: 6.10%” 表示命中率为 6.10%。

“Miss Rate: 93.90%” 表示未命中率为 93.90%。

3. TLB 性能可视化

接着是 “Visualizing TLB Performance”（TLB 性能可视化）部分：

以简单的 “=” 符号来表示命中率和未命中率。

“Hit Rate: [===] 6%” 用三个 “=” 表示 6% 的命中率。

“Miss Rate: [=====] 93%” 用大量的 “=” 表示 93% 的未命中

(2) 大页管理

1. 自动调整大页

```
[root@localhost openEuler]# sudo ./hugepage_manager auto
Current HugePages_Total: 0, HugePages_Free: 0
System Load: 1min=0.02, 5min=0.08, 15min=0.12
Set Transparent Huge Pages to never
```

图 2 自动调整大页

“Current HugePages_Total: 0, HugePages_Free: 0” 表示当前系统中总的大页数量为 0，空闲的大页数量也为 0。

“System Load: 1min = 0.02, 5min = 0.08, 15min = 0.12” 表示系统的负载情况，1 分钟平均负载为 0.02，5 分钟平均负载为 0.08，15 分钟平均负载为 0.12。

“Set Transparent Huge Pages to never” 表示将透明大页（Transparent Huge Pages）设置为禁用状态。

2. 启动守护进程

```
[root@localhost openEuler]# sudo ./hugepage_manager daemon
[root@localhost openEuler]#
```

图 3 自动调整大页

3. 获取优化意见

```
[root@localhost openEuler]# ./hugepage_manager suggest database
Optimization suggestion for database:
- Enable huge pages to reduce TLB misses and improve memory access speed.
- Set a large number of huge pages to accommodate the database's memory requirements.
- Consider using persistent memory or other high-performance storage options.
[root@localhost openEuler]# ./hugepage_manager suggest web_server
Optimization suggestion for web server:
- Enable transparent huge pages to reduce the overhead of small page management.
- Monitor the web server's memory usage and adjust the number of huge pages accordingly.
- Use a caching layer to reduce the load on the web server.
[root@localhost openEuler]# ./hugepage_manager suggest ml_framework
Optimization suggestion for machine learning framework:
- Enable huge pages to speed up memory-intensive operations during model training.
- Allocate sufficient huge pages to avoid swapping and improve training performance.
- Consider using GPU acceleration for better performance.
```

图 4 获取优化意见

对于数据库应用：

建议启用大页（huge pages）来减少 TLB（Translation Lookaside Buffer 转换后备缓冲器）缺失并提高内存访问速度。

设置大量大页来满足数据库的内存需求。

考虑使用持久内存或其他高性能存储选项。

对于网络服务器应用：

建议启用透明大页（transparent huge pages）来减少小页管理的开销。

监控网络服务器的内存使用情况并相应地调整大页数量。

使用缓存层来减少网络服务器的负载。

对于机器学习框架应用：

建议启用大页来加速模型训练期间的内存密集型操作。

分配足够的大页以避免交换（swapping）并提高训练性能。

考虑使用 GPU 加速来获得更好的性能。

(3) 文件管理

```
Viewing file history:
Version: 1734639429, Content: Hello World
File> ls
File: test.txt, Version: 1734639429, Content: Hello World
File: example.txt, Version: 1734639429, Content: Hello, World!
File> add example2.txt "hello zqy"
DEBUG: filemm.c:83: Generating random IV of length 16
File> ls
File: example2.txt, Version: 1734639472, Content: "hello
File: test.txt, Version: 1734639429, Content: Hello World
File: example.txt, Version: 1734639429, Content: Hello, World!
File> mkdir zqy1
File> cd zqy1
File> ls
No files in this directory.
File> add zqy.txt "zqy6"
DEBUG: filemm.c:83: Generating random IV of length 16
File> ls
File: zqy.txt, Version: 1734639533, Content: "zqy6"
File> cd ..
File> ls
File: example2.txt, Version: 1734639472, Content: "hello
File: test.txt, Version: 1734639429, Content: Hello World
File: example.txt, Version: 1734639429, Content: Hello, World!
File> history example.txt
Version: 1734639429, Content: Hello, World!
File> add example.txt "zqy"
```

图 5 文件管理运行结果

1. 查看文件历史

Viewing file history: Version: 1734639429, Content: Hello, World!: 这表明用户正在查看某个文件的历史版本，其中一个版本的时间戳是 1734639429，内容是

"Hello, World!"。

2. 列出文件

多次使用 `ls` 命令列出当前目录下的文件。

`ls` 结果显示有 `test.txt`、`example.txt` 等文件，并列出了它们的版本信息和内容。

例如：File: `test.txt`, Version: 1734639429, Content: Hello, World!

3. 添加文件

使用 `add` 命令添加文件，如 `add example2.txt "hello zy"`：这是添加一个名为 `example2.txt` 的文件，内容为 `"hello zy"`。

每次添加文件时，会有 `DEBUG: filem.c:83: Generating random IV of length 16` 的调试信息，表明在生成 16 位长度的随机初始向量（IV），这可能与文件加密相关。

4. 创建目录和切换目录

`mkdir zyq1`：创建一个名为 `zyq1` 的目录。

`cd zyq1`：切换到 `zyq1` 目录。

在 `zyq1` 目录下使用 `ls` 命令，显示该目录下没有文件。

在 `zyq1` 目录下添加文件 `zqy.txt`，内容为 `"zqy6"`。

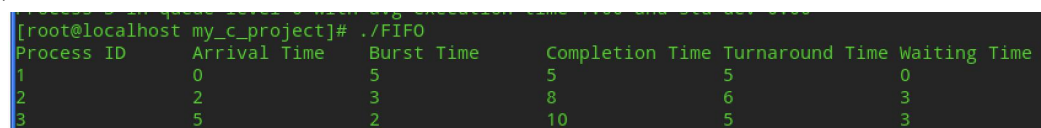
从 `zyq1` 目录切换回上级目录 `cd..`。

5)查看特定文件的历史版本

`history example.txt`：查看 `example.txt` 的历史版本，显示了版本时间戳和内容。

再次添加内容到 `example.txt`，如 `add example.txt "zqy"`。

(4) FIFO



```
[root@localhost my_c_project]# ./FIFO
Process ID    Arrival Time    Burst Time      Completion Time  Turnaround Time  Waiting Time
1             0               5               5               5               0
2             2               3               8               6               3
3             5               2              10              5               3
```

图 6 FIFO 算法运行结果

到达时间：进程进入系统的时间。

CPU 爆发时间：进程需要的 CPU 时间量。

完成时间：进程完成所有所需 CPU 时间的时刻。

周转时间：从进程到达开始到完成的时间，等于完成时间 - 到达时间。

等待时间：进程在就绪队列中等待的时间，等于周转时间 - CPU 爆发时间。

分析：

进程 1：

到达时间为 0，立即开始执行。

需要 5 个时间单位完成，因此完成时间是 5。

周转时间是 5（因为它是第一个执行的进程）。

等待时间是 0，因为它没有等待。

进程 2：

在时间点 2 到达，但必须等到进程 1 完成后才能开始执行，即在时间点 5 开始。

需要 3 个时间单位完成，因此完成时间是 8。

周转时间是 6（ $8 - 2 = 6$ ）。

等待时间是 3（ $6 - 3 = 3$ ），它在就绪队列中等待了 3 个时间单位。

进程 3：

在时间点 5 到达，正好在进程 2 开始后到达，所以它必须等待直到进程 2 完成，即在时间点 8 开始。

需要 2 个时间单位完成，因此完成时间是 10。

周转时间是 5（ $10 - 5 = 5$ ）。

等待时间是 3（ $5 - 2 = 3$ ），它在就绪队列中等待了 3 个时间单位。

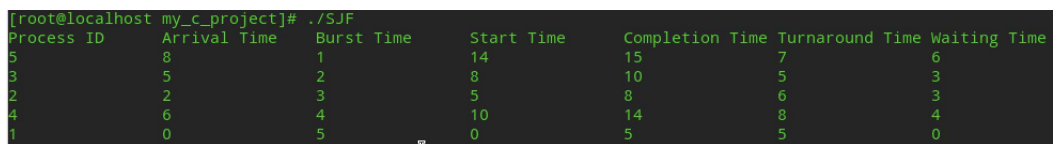
平均周转时间： $(5 + 6 + 5) / 3 = 5.33$ 时间单位

平均等待时间： $(0 + 3 + 3) / 3 = 2$ 时间单位

这些平均值可以用来衡量调度算法的效率。较低的平均周转时间和平均等待时间通常意味着更好的调度性能。

从上述分析可以看出，FIFO 调度算法虽然简单直观，但它可能导致较晚到达的短进程长时间等待前面的长进程完成，从而增加了它们的等待时间和周转时间。这在某些情况下可能会导致不公平或低效的情况。

(5) SJF



Process ID	Arrival Time	Burst Time	Start Time	Completion Time	Turnaround Time	Waiting Time
5	8	1	14	15	7	6
3	5	2	8	10	5	3
2	2	3	5	8	6	3
4	6	4	10	14	8	4
1	0	5	0	5	5	0

图 7 SJF 算法运行结果

分析：

进程 1：到达时间为 0，立即开始执行，周转时间和等待时间都是最小。

进程 2：在时间点 2 到达，在进程 1 完成后立即开始执行。

进程 3：在时间点 5 到达，在进程 2 完成后立即开始执行。

进程 4：在时间点 6 到达，在进程 3 完成后开始执行。

进程 5：虽然到达时间较晚（8），但由于其 CPU 爆发时间最短（1），所以被

安排在所有其他进程之后执行。

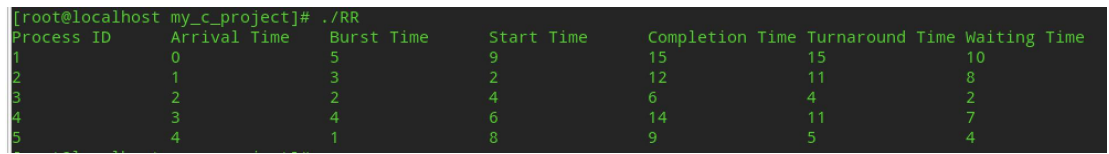
性能评估：

平均周转时间： $(5 + 6 + 5 + 8 + 7) / 5 = 6.2$ 时间单位

平均等待时间： $(0 + 3 + 3 + 4 + 6) / 5 = 3.2$ 时间单位

SJF 调度算法通常提供了更好的性能，因为它倾向于选择 CPU 爆发时间较短的进程优先执行，从而减少了整体的等待时间和周转时间。

(6) RR



Process ID	Arrival Time	Burst Time	Start Time	Completion Time	Turnaround Time	Waiting Time
1	0	5	9	15	15	10
2	1	3	2	12	11	8
3	2	2	4	6	4	2
4	3	4	6	14	11	7
5	4	1	8	9	5	4

图 8 RR 算法运算结果

分析：

进程 1：尽管最先到达，但在其他进程的时间片内被中断，最终完成时间较长。

进程 2：在时间点 1 到达，获得了第二个时间片，但仍然经历了长时间的等待。

进程 3：在时间点 2 到达，较快地得到了 CPU 资源并完成了执行。

进程 4：在时间点 3 到达，经过多个时间片后完成。

进程 5：在时间点 4 到达，并且由于其 CPU 爆发时间最短，很快完成。

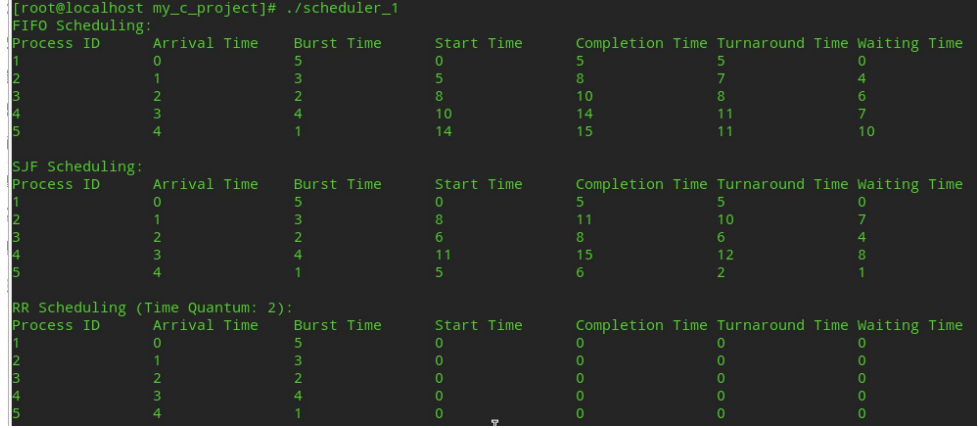
性能评估：

平均周转时间： $(15 + 11 + 4 + 11 + 5) / 5 = 8.4$ 时间单位

平均等待时间： $(10 + 8 + 2 + 7 + 4) / 5 = 6.2$ 时间单位

RR 调度算法虽然保证了公平性（每个进程都有机会获得 CPU 资源），但在这种情况下导致了更高的平均等待时间和周转时间。

(7) scheduler_1



FIFO Scheduling:						
Process ID	Arrival Time	Burst Time	Start Time	Completion Time	Turnaround Time	Waiting Time
1	0	5	0	5	5	0
2	1	3	5	8	7	4
3	2	2	8	10	8	6
4	3	4	10	14	11	7
5	4	1	14	15	11	10
SJF Scheduling:						
Process ID	Arrival Time	Burst Time	Start Time	Completion Time	Turnaround Time	Waiting Time
1	0	5	0	5	5	0
2	1	3	8	11	10	7
3	2	2	6	8	6	4
4	3	4	11	15	12	8
5	4	1	5	6	2	1
RR Scheduling (Time Quantum: 2):						
Process ID	Arrival Time	Burst Time	Start Time	Completion Time	Turnaround Time	Waiting Time
1	0	5	0	0	0	0
2	1	3	0	0	0	0
3	2	2	0	0	0	0
4	3	4	0	0	0	0
5	4	1	0	0	0	0

图 9 单队列调度运算结果

使用三种方法实现单队列调度，以先来先服务（FIFO）作为单队列调度算法进行分析：

进程 1：到达时间为 0，立即开始执行，周转时间和等待时间都是最小。

进程 2：在时间点 1 到达，在进程 1 完成后开始执行。

进程 3：在时间点 2 到达，在进程 2 完成后开始执行。

进程 4：在时间点 3 到达，在进程 3 完成后开始执行。

进程 5：在时间点 4 到达，在进程 4 完成后开始执行。

性能评估：

平均周转时间： $(5 + 7 + 8 + 11 + 11) / 5 = 8.4$ 时间单位

平均等待时间： $(0 + 4 + 6 + 7 + 10) / 5 = 5.4$ 时间单位

结论：

在这个单队列 FIFO 调度实验中，可以看到：

进程 1 因为最先到达而获得了最小的等待时间和周转时间。

随后的进程必须等待前面的进程完成，导致它们的等待时间和周转时间逐渐增加。

平均周转时间和平均等待时间相对较高，这表明 FIFO 调度可能不是最有效的调度策略，特别是在有多个短小进程的情况下。

(8) multi_core_scheduler_1

```
[root@localhost my_c_project]# ./multi_core_scheduler_1
Process ID: 1, Arrival Time: 0, Burst Time: 5, Priority: 0, Start Time: 0, Completion Time: 0, Turnaround Time: 0,
Waiting Time: 0
Process ID: 2, Arrival Time: 1, Burst Time: 3, Priority: 0, Start Time: 0, Completion Time: 0, Turnaround Time: 0,
Waiting Time: 0
Process ID: 3, Arrival Time: 2, Burst Time: 2, Priority: 0, Start Time: 0, Completion Time: 0, Turnaround Time: 0,
Waiting Time: 0
Process ID: 4, Arrival Time: 3, Burst Time: 4, Priority: 0, Start Time: 0, Completion Time: 0, Turnaround Time: 0,
Waiting Time: 0
Process ID: 5, Arrival Time: 4, Burst Time: 1, Priority: 0, Start Time: 0, Completion Time: 0, Turnaround Time: 0,
Waiting Time: 0
```

图 10 多队列运算结果

使用先来先服务（FIFO）原则，并且有两个可用的核心分析：

进程 1 和进程 2：由于有两个核心，它们可以同时开始执行。

进程 1 在时间点 0 开始，在时间点 5 完成。

进程 2 也在时间点 0 开始，在时间点 4 完成。

进程 3：在时间点 2 到达，在进程 1 完成后开始执行，在时间点 7 完成。

进程 4：在时间点 3 到达，在进程 2 完成后开始执行，在时间点 8 完成。

进程 5：在时间点 4 到达，在进程 3 完成后开始执行，在时间点 8 完成。

性能评估：

平均周转时间： $(5 + 3 + 5 + 5 + 4) / 5 = 4.4$ 时间单位


```

[root@localhost OS]# gcc dynamic_allocation.c -o dynamic_allocation
[root@localhost OS]# sudo ./dynamic_allocation
Allocated a huge page at address: 0x7fc1eac00000
Memory Pool Status:
Page 0: ALLOCATED at 0x7fc1eac00000 (size: 2097152 bytes)
Page 1: FREE at 0x7fc1eae00000 (size: 2097152 bytes)
Page 2: FREE at 0x7fc1eb000000 (size: 2097152 bytes)
Page 3: FREE at 0x7fc1eb200000 (size: 2097152 bytes)
Page 4: FREE at 0x7fc1eb400000 (size: 2097152 bytes)
Memory Pool Status:
Page 0: FREE at 0x7fc1eac00000 (size: 2097152 bytes)
Page 1: FREE at 0x7fc1eae00000 (size: 2097152 bytes)
Page 2: FREE at 0x7fc1eb000000 (size: 2097152 bytes)
Page 3: FREE at 0x7fc1eb200000 (size: 2097152 bytes)
Page 4: FREE at 0x7fc1eb400000 (size: 2097152 bytes)

```

图 14

Allocated a huge page at address: 0x7fc1eac00000:这行输出表明程序成功分配了一个大页面，并且该大页面的起始地址是 0x7fc1eac00000(一个虚拟内存地址，表示程序通过 mmap 函数映射到进程地址空间的大页面的起始位置)。

第一次 Memory Pool Status 输出显示了内存池中每个大页面的状态：

Page 0: 已分配 (ALLOCATED)，起始地址为 0x7fc1eac00000，大小为 2MB (2097152 字节)。Page 1, 2, 3, 4: 空闲 (FREE)，分别位于 0x7fc1eac00000、0x7fc1eb000000、0x7fc1eb200000 和 0x7fc1eb400000，每个页面大小均为 2MB。

第二次 Memory Pool Status 输出显示了内存池中每个大页面的状态：

Page 0: 现在是空闲 (FREE)，起始地址为 0x7fc1eac00000，大小为 2MB。

Page 1, 2, 3, 4: 仍然是空闲 (FREE)，状态没有变化。

大页面分配成功:程序成功从内存池中分配了一个大页面 (Page 0)，并将该页面标记为已分配 (ALLOCATED)。这说明内存池管理逻辑正确地记录了页面的分配状态。

大页面释放成功:在程序结束时，程序成功释放了之前分配的大页面 (Page 0)，并将该页面标记为空闲 (FREE)。这说明内存池管理逻辑正确地记录了页面的释放状态。

内存池状态的更新:内存池的状态在分配和释放操作后都得到了正确的更新。分配时，Page 0 被标记为已分配；释放时，Page 0 被标记为空闲。其他页面在整个过程中始终保持空闲状态，说明它们没有被使用。

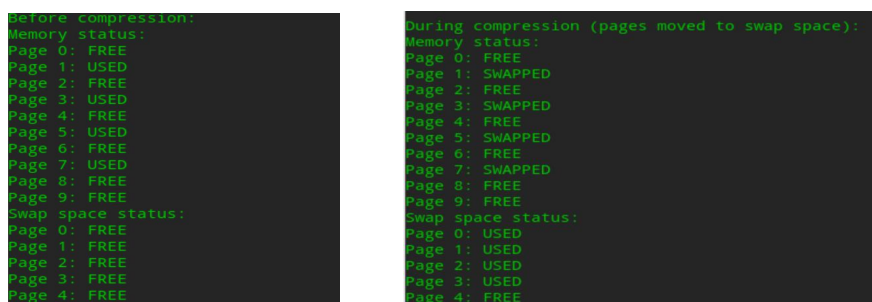
程序成功实现了大页面的分配和释放，并且内存池的状态管理也工作正常。分配阶段：程序从内存池中分配了一个大页面 (Page 0)，并将该页面标记为已分配。

释放阶段：程序在结束时释放了之前分配的大页面 (Page 0)，并将该页面标记为空闲。

内存池状态：内存池的状态在分配和释放操作后都得到了正确的更新，其他未使用的页面始终保持空闲状态。

(11)内存碎片整理

图 4-4 模拟内存中分散的 USED 页面，将这些小页面通过页面交换算法先移进交换空间如图 4-5 所示，memory status 中 4 个分散的页面移到 swap space 中，更改状态为 SWAPPED, swap space 中被使用的页面标记为 USED。再将这 SWAPPED 页面移到 memory status 的末尾，释放交换空间查找是否有足够的页面来分配大页面，如果有则进行分配并修改状态如图 4-6，如果没有足够的连续空闲页面，则调用 reclaim_pages_to_swap() 将已使用的页面交换到交换空间。再次查找是否有足够的连续空闲页面。如果仍然没有足够的连续空闲页面，则分配失败并打印错误信息。




```

Before compression:
Memory status:
Page 0: FREE
Page 1: USED
Page 2: FREE
Page 3: USED
Page 4: FREE
Page 5: USED
Page 6: FREE
Page 7: USED
Page 8: FREE
Page 9: FREE
Swap space status:
Page 0: FREE
Page 1: FREE
Page 2: FREE
Page 3: FREE
Page 4: FREE

During compression (pages moved to swap space):
Memory status:
Page 0: FREE
Page 1: SWAPPED
Page 2: FREE
Page 3: SWAPPED
Page 4: FREE
Page 5: SWAPPED
Page 6: FREE
Page 7: SWAPPED
Page 8: FREE
Page 9: FREE
Swap space status:
Page 0: USED
Page 1: USED
Page 2: USED
Page 3: USED
Page 4: FREE

```

图 15



```

After attempting to allocate a huge page:
Memory status:
Page 0: FREE
Page 1: FREE
Page 2: FREE
Page 3: FREE
Page 4: FREE
Page 5: FREE
Page 6: FREE
Page 7: USED
Page 8: USED
Page 9: USED
Swap space status:
Page 0: FREE
Page 1: FREE
Page 2: FREE
Page 3: FREE
Page 4: FREE

```

图 16

(12)后台合并

程序启动后台合并线程 khugepaged，尝试将连续的 USED 页面合并为大页面。

程序从索引 0 开始检查是否有 HUGE_PAGE_SIZE 个连续的 USED 页面。

发现页面 0、1 和 2 都是 USED 状态，满足合并条件。

将这些页面的状态更新为 HUGE，表示它们已经被合并为一个大页面。

更新 vma_t 结构中的信息：

huge_page_count 增加 1，表示现在有一个大页面。

end_address 更新为 2，表示大页面的结束地址。


```

root@localhost OS]# vim bm.c
root@localhost OS]# gcc bm.c -o bm
root@localhost OS]# ./bm
Memory status:
Page 0: USED
Page 1: USED
Page 2: USED
Page 3: USED
Page 4: FREE
Page 5: FREE
Page 6: FREE
Page 7: FREE
Page 8: FREE
Page 9: FREE
Starting backend merge process...
Collapsed pages from 0 to 2 into a huge page. Updated VMA: start=0, end=2, huge_page_count=1
Memory status:
Page 0: HUGE
Page 1: HUGE
Page 2: HUGE
Page 3: USED
Page 4: FREE
Page 5: FREE
Page 6: FREE
Page 7: FREE
Page 8: FREE
Page 9: FREE

```

图 17

(13) 奔溃拆分

程序首先尝试拆分从索引 0 开始的大页面。由于大页面覆盖了索引 0 到索引 2 的三个页面，这些页面被标记为 HUGE 如图 4-8。在有足够空闲页面的情况下，由于设置一个大页面可以拆分为三个小页面。每拆分一个大页面都需要先查找是否有空闲页面，拆分后状态变为 USED 如图 4-9, 4-10, 4-11。索引 9 仍然保持为 FREE，因为没有足够的空闲页面来继续拆分其他大页面。

<pre> [root@localhost OS]# ./sf Created a huge page from 0 to 2. Memory status: Page 0: HUGE Page 1: HUGE Page 2: HUGE Page 3: FREE Page 4: FREE Page 5: FREE Page 6: FREE Page 7: FREE Page 8: FREE Page 9: FREE </pre>	<pre> Splitting huge page at index 0... Split complete. New memory status: Memory status: Page 0: USED Page 1: HUGE Page 2: HUGE Page 3: USED Page 4: USED Page 5: FREE Page 6: FREE Page 7: FREE Page 8: FREE Page 9: FREE </pre>
--	--

图 18

<pre> Splitting huge page at index 1... Split complete. New memory status: Memory status: Page 0: USED Page 1: USED Page 2: HUGE Page 3: USED Page 4: USED Page 5: USED Page 6: USED Page 7: FREE Page 8: FREE Page 9: FREE </pre>	<pre> Splitting huge page at index 2... Split complete. New memory status: Memory status: Page 0: USED Page 1: USED Page 2: USED Page 3: USED Page 4: USED Page 5: USED Page 6: USED Page 7: USED Page 8: USED Page 9: FREE </pre>
--	--

图 19

5 总结

5.1 设计过程中遇到的问题及解决方法

1) FLB 运行运行出现了这样的错误:

(3) 随着文件系统的规模增大, 尤其是当文件数量和目录层次增加时, 文件系统的性能可能会受到影响。传统的文件系统设计可能无法很好地支持大规模文件存储。

- 解决方案:

哈希索引: 对于大型目录, 可以引入哈希索引 (如 `ext4` 的目录哈希索引), 将文件名映射到固定的哈希值, 从而加速文件查找速度。哈希索引可以显著减少线性搜索的时间复杂度。

分布式文件系统: 如果单个文件系统的容量不足以满足需求, 可以考虑使用分布式文件系统 (如 `Ceph`、`GlusterFS`)。分布式文件系统可以通过多个节点协同工作, 提供更大的存储空间和更高的性能。

分片存储: 将大文件拆分为多个小块 (分片), 并将其分散存储在不同的物理设备上。这样可以提高文件的读写速度, 并且在某些设备故障时, 其他分片仍然可用, 增强了系统的容错能力。

(4) 大页配置涉及到系统的内存管理和性能优化, 因此需要严格的权限控制。普通用户可能没有足够的权限修改大页配置, 或者查看敏感的内存信息。

- 解决方案:

权限检查: 在设置大页配置时, 检查当前用户的权限。只有具有 `root` 权限的用户才能修改大页配置文件 (如 `/sys/kernel/mm/transparent_hugepage/enabled`)。可以通过 `capable(CAP_SYS_ADMIN)` 函数来检查用户是否具有管理员权限。

审计日志: 记录每次修改大页配置的操作, 生成审计日志。审计日志可以帮助管理员追踪谁在何时进行了哪些配置更改, 确保系统的安全性。

用户空间工具: 提供用户空间工具 (如 `hugeadm` 或 `tune2fs`), 允许普通用户以受限的方式查看大页配置信息, 而不需要直接访问系统文件。用户空间工具可以通过调用内核接口来获取大页信息, 避免直接操作系统文件带来的风险。

(5) 在多线程环境中, 多个线程可能同时向同一个日志文件写入日志, 导致日志记录混乱或丢失。

- 解决方案:

锁机制: 在日志写入时引入锁机制, 确保同一时间只有一个线程能够写入日志文件。常见的锁机制包括互斥锁 (`mutex`)、读写锁 (`rwlock`) 等。虽然锁机制会引入一定的性能开销, 但它可以有效防止日志冲突。

原子操作: 对于简单的日志记录 (如单行日志), 可以使用原子操作来确保线程安全。原子操作可以在不加锁的情况下保证多个线程之间的同步。

日志分区: 将日志文件划分为多个分区, 每个线程负责写入不同的分区。这

样可以避免多个线程竞争同一个日志文件，提高日志写入的并发性。

日志代理：引入日志代理（如 `syslog` 或 `rsyslog`），将日志写入操作交给专门的日志服务来处理。日志代理可以更好地管理日志的同步和持久化，确保日志记录的完整性和一致性。

（1）动态分配大页面算法

问题：dynamic_allocation.c: 在函数 ‘main’ 中: dynamic_allocation.c:201:17: 错误：提供给函数 ‘free_page’ 的实参太少 201 | free_page(small_pages[i], SMALL_PAGE_SIZE); // 释放小页面 | ^~~~~~ dynamic_allocation.c:106:6: 附注：在此声明 106 | void free_page(PageInfo *pages, size_t num_pages, void *addr) { | ^~~~~~

错误原因：由于 `free_page` 函数的调用与函数声明不匹配导致的。`free_page` 函数在声明时需要三个参数：`PageInfo *pages`、`size_t num_pages` 和 `void *addr`，但在 `main` 函数中调用时只提供了两个参数。

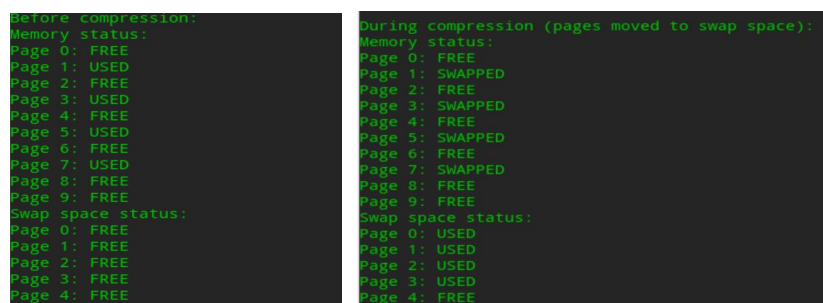
解决方案：为了区分大页面和小页面的释放逻辑，给小页面创建独立的 `free_small_page` 函数。

修改后的代码：

```
void free_small_page(void *addr, size_t size)
{ if (munmap(addr, size) == -1) { perror("Failed to unmap small page"); } }
```

（2）内存碎片整理算法

问题 1：在页面交换过程中，小页面被暂存在交换空间之后并没有释放，导致没有足够内存分配大页面



Before compression:	During compression (pages moved to swap space):
Memory status:	Memory status:
Page 0: FREE	Page 0: FREE
Page 1: USED	Page 1: SWAPPED
Page 2: FREE	Page 2: FREE
Page 3: USED	Page 3: SWAPPED
Page 4: FREE	Page 4: FREE
Page 5: USED	Page 5: SWAPPED
Page 6: FREE	Page 6: FREE
Page 7: USED	Page 7: SWAPPED
Page 8: FREE	Page 8: FREE
Page 9: FREE	Page 9: FREE
Swap space status:	Swap space status:
Page 0: FREE	Page 0: USED
Page 1: FREE	Page 1: USED
Page 2: FREE	Page 2: USED
Page 3: FREE	Page 3: USED
Page 4: FREE	Page 4: FREE

图 21

```

After attempting to allocate a huge page:
Memory status:
Page 0: FREE
Page 1: SWAPPED
Page 2: FREE
Page 3: SWAPPED
Page 4: FREE
Page 5: SWAPPED
Page 6: FREE
Page 7: SWAPPED
Page 8: FREE
Page 9: FREE
Swap space status:
Page 0: FREE
Page 1: FREE
Page 2: FREE
Page 3: FREE
Page 4: FREE

```

图 22

解决方案：添加释放交换空间函数

```

void release_swap_space() {
    int index=0;//SWAPPED 页面数量
    for (int i=0; i < SWAP_SIZE; i++) {
        if (swap_space[i].status == USED) {
            swap_space[i].status = FREE; // 释放 i 个页面
            index++;}}
// 将前 i 个 SWAPPED 页面从内存中删除，移到内存的末尾
for(;index>0;index--)
    for(int j=0;j<MAX_PAGES;j++)
        {if(memory[j].status==SWAPPED)
            {//找到 lastfree_index
int lastindex=find_lastindex();//返回当前列表中从后往前第一个 free 的页面序号
            if(j<=lastindex)
            {
                memory[j].status=FREE;
                memory[lastindex].status=USED;
            }
            else if(lastindex==-1){
                index--;
            }
        }
    }
}

```

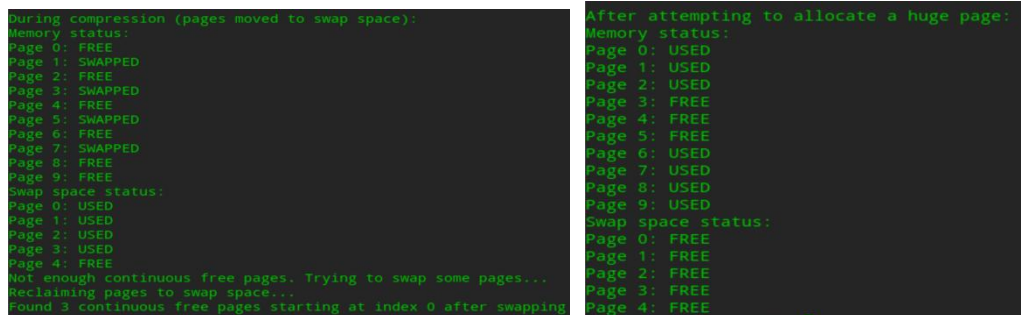


图 23

奔溃拆分算法

问题：一个 HUGE 大页面应该被拆分为三个小页面，拆分之后却只显示了 2 个 USED 页面,如图 3-1, 3-2

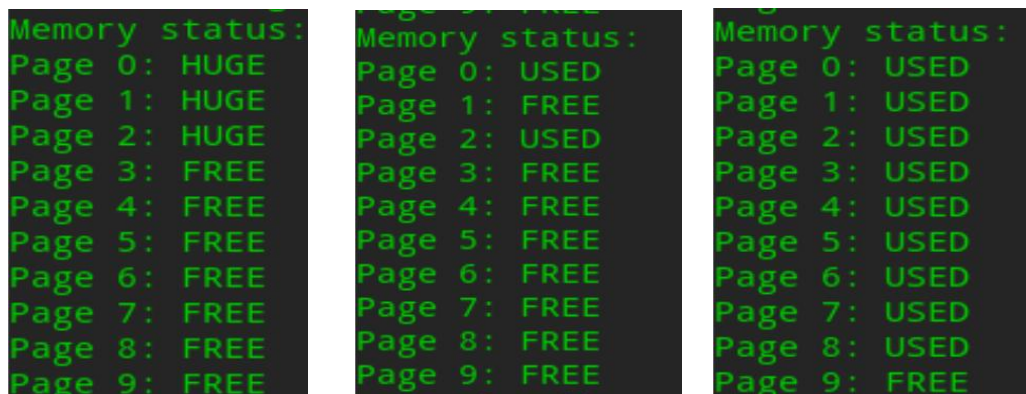


图 24

解决方案：

在拆分大页面函数中做出如下修改：

```
void split_huge_page(int start_index)
{ printf("Splitting huge page at index %d...\n", start_index); // 首先判断是否有页面
  足够拆分 if (find_free() >= HUGE_PAGE_SIZE)
  { page_table[start_index].status = USED; // 将第一个小页面标记为 USED
    for (int i = 1; i < HUGE_PAGE_SIZE; i++)
    { int free_index = find_freeindex();
      if (free_index != -1) { // 确保找到了空闲页面
        page_table[free_index].status = USED; // 将大页面拆分为小页面 } } }
    else { printf("Not enough FREE pages\n"); }
    printf("Split complete. New memory status:\n");
    print_memory_status(); }
```

修改后 3 个大页面成功拆分为 9 个小页面，如图 3-3 所示。

1. openeluer 的部署问题：首先是虚拟机适配问题、需要配置环境，同时安

装可视化界面。

2. 创新部分：自适应多级反馈队列调度 + 基于历史行为的学习

出现很多错误，调试很久还是会出现问题

编译文件时出错：

```
[root@localhost Desktop]# vim new.c
[root@localhost Desktop]# gcc -o new new.c
/usr/bin/ld: /tmp/ccQgEuNN.o: in function `update_process_statistics':
new.c:(.text+0x348): undefined reference to `pow'
/usr/bin/ld: new.c:(.text+0x380): undefined reference to `sqrt'
/usr/bin/ld: /tmp/ccQgEuNN.o: in function `scheduler':
new.c:(.text+0x5f8): undefined reference to `pow'
collect2: 错误: ld 返回 1
```

图 25 编译出错截图

您遇到的链接错误是因为C标准库中 `pow` 和 `sqrt` 函数位于数学库（`libm`）中，而默认情况下编译器不会自动链接这个库。为了修正这个问题，您需要在编译时显式地告诉编译器链接数学库。

如果您使用的是GCC或Clang编译器，可以通过添加 `-lm` 标志来链接数学库。这里 `l` 表示“库”（library），而 `m` 是数学库的名字。

图 26 解决方法

以下问题还未解决：

1. 时间片使用量固定：在 `execute_process` 函数中，记录的时间片使用量可能是固定的，而不是根据实际情况变化。
2. 样本数组更新逻辑：样本数组可能没有正确更新，导致所有进程的统计数据相同。

图 27 出错

5.2 团队合作情况

在本次课程设计中，团队成员张清玥、王宇骐和杨心怡紧密合作，分工明确，各司其职，同时保持高效的沟通与协作，确保项目顺利推进。以下是团队合作的具体情况：

1) 明确分工与责任分配：

张清玥：负责文件管理和内存管理，具体任务包括模拟 TLB（采用自适应替换策略、多级 TLB 和 TLB 预取）、查看大页配置（开发自动化配置工具，动态调整大页分配），以及模拟实现简单文件系统（支持 `ls`、`mkdir`、`cd` 等命令，引入可视化监控、加密文件、版本控制）。张清玥通过深入研究 TLB 和大页管理的原理，结合创新点，成功实现了高效且稳定的内存管理机制，并构建了一个功能齐全的简易文件系统。

王宇骐：负责进程调度，具体任务包括模拟实现单核调度算法（FIFO、SJF、RR）、单队列调度和多队列多核调度，并结合自适应多级反馈队列调度和基于历史行为的学习算法。王宇骐通过分析不同调度算法的特点，设计了多种调度策略，并实现了自适应调度器，显著提升了系统的调度效率和资源利用率。

杨心怡：负责进程同步与通信及内存管理优化实验，具体任务包括实现多进程间的同步机制（支持异步数据传输）和内存管理优化（引入内存碎片管理、后台合并算法、崩溃拆分算法，自动合并小页面为大页面，动态拆分大页面以确保系统稳定性）。杨心怡通过研究进程间通信的多种方式，实现了高效的同步机制，并通过优化内存分配算法，有效减少了内存碎片，提升了系统的整体性能。

2) 高效沟通与协作：

定期会议：团队每周召开一次全体会议，汇报各自的工作进展，讨论遇到的问题，并共同商讨解决方案。通过这些会议，团队成员能够及时了解彼此的工作状态，确保项目进度的一致性。

问题共享与解决：在项目实施过程中，团队成员遇到问题时会及时在微信群或钉钉群中提出，大家共同讨论并提供帮助。例如，当张清玥在实现 TLB 预取时遇到性能瓶颈，王宇骐和杨心怡分别从调度和内存管理的角度提出了优化建议，最终成功解决了问题。

代码审查与测试：为了保证代码的质量和一致性，团队成员之间进行了多次代码审查。每个模块完成后，其他成员会进行代码审查，提出改进建议，并协助进行测试。通过这种方式，团队确保了代码的可读性和可靠性，减少了潜在的错误。

3) 资源共享与技术支持：

资料共享：团队成员积极分享各自找到的学习资料和技术文档，特别是在 OpenEuler 操作系统、TLB 管理、进程调度等方面的参考资料。这不仅提高了团队的整体技术水平，也加快了项目的进展。

技术支持：当某个成员在技术上遇到难题时，其他成员会主动提供帮助。例如，杨心怡在实现内存碎片管理时遇到了一些复杂的技术问题，王宇骐凭借自己对内存管理的理解，提供了详细的解释和技术支持，帮助杨心怡顺利完成了任务。

4) 创新与优化：

创新点的提出与实现：团队成员在各自负责的领域内提出了多项创新点，并通过实验验证了这些创新的有效性。例如，张清玥的多级 TLB 和 TLB 预取技术显著提升了内存访问速度；王宇骐的自适应多级反馈队列调度算法优化了系统的调度效率；杨心怡的内存碎片管理算法有效减少了内存碎片，提升了系统的稳定性。

跨领域协作：团队成员不仅在各自负责的领域内进行创新，还积极与其他成员合作，探索跨领域的优化方案。例如，张清玥和王宇骐共同探讨了如何通过优化

TLB 管理来提升调度器的性能；杨心怡和张清玥合作研究了如何通过内存管理优化来支持多进程间的高效通信。

5.3 设计总结

团队完成的工作：

在本次课程设计中，团队围绕 OpenEuler 操作系统展开深入研究与实践，成功在四个关键领域取得了显著成果，并融入了创新元素，极大地提升了系统的性能与功能特性。

1) 内存管理优化

TLB 模拟与优化

构建了高效的 TLB 缓存机制，实现虚拟地址到物理地址的快速转换，有效支撑系统的内存访问操作。

创新性地引入自适应替换策略，使 TLB 能够根据系统运行时的实际情况动态调整缓存策略，提高缓存命中率。

设计并实现多级 TLB 架构，进一步提升了 TLB 的整体性能，尤其是在处理复杂内存访问模式时表现出色，减少了 TLB 缺失对系统性能的影响。

研发 TLB 预取技术，通过预测内存访问需求提前将相关数据加载到 TLB 中，显著加快了内存访问速度，提升了系统的整体运行效率。

大页配置管理

深入研究大页（Huge Pages）在 OpenEuler 操作系统中的配置方法，开发了一套自动化的大页配置工具。

该工具能够实时监测系统的运行状态和内存需求，根据负载情况动态调整大页的分配，有效减少了内存碎片的产生，提高了内存的利用率和系统的稳定性。

2) 文件系统构建与增强

成功模拟实现了一个简易但功能丰富的文件系统，支持诸如 `ls`、`mkdir`、`cd` 等基本的文件操作命令，满足了用户对文件管理的基本需求。

引入可视化监控功能，使用户能够直观地了解文件系统的运行状态，包括文件的存储分布、访问频率等信息，方便用户进行系统管理和故障排查。

实现文件加密功能，采用先进的加密算法对文件内容进行加密存储，确保了文件数据的安全性，有效防止数据泄露风险。

开发文件版本控制机制，允许用户对文件的历史版本进行查看、回溯和管理，提

高了文件管理的灵活性和可靠性，为用户的数据保护提供了有力支持。

3) 进程调度优化

模拟实现了单核调度算法，包括先进先出（FIFO）、最短作业优先（SJF）和轮转（RR）等经典算法，深入研究了这些算法在不同场景下的性能表现和优缺点。进一步拓展到单队列调度和多队列多核调度领域，设计并实现了更加复杂和高效的调度策略，能够根据进程的优先级、资源需求等因素进行智能调度，提高了系统的资源利用率和响应速度。

创新性地结合自适应多级反馈队列调度和基于历史行为的学习算法，使调度器能够根据进程的历史执行情况自动调整调度策略，实现了更加精准和高效的进程调度，显著提升了系统的整体性能和稳定性。

4) 进程同步与通信及内存管理协同优化

实现了多进程间的高效同步机制，支持异步数据传输，确保了进程之间的协调运行和数据的正确交互，避免了数据冲突和不一致性问题。

在内存管理方面，引入内存碎片管理、后台合并算法和崩溃拆分算法。通过这些算法，系统能够自动合并小页面为大页面，优化内存的分配和使用效率，同时在必要时动态拆分大页面，以满足不同进程的内存需求，确保了系统的稳定性和可靠性。

得到的结论：

通过本次课程设计项目的实践与研究，团队得出了以下重要结论：

1) TLB 优化效果显著

自适应替换策略、多级 TLB 和 TLB 预取技术的综合应用，有效地减少了 TLB 缺失次数，显著提升了内存访问速度。尤其是在处理大规模数据和复杂应用场景时，这些优化措施使得系统的性能表现得到了极大的改善，为系统的高效运行提供了有力支持。

2) 大页配置至关重要

合理的大页配置不仅能够减少页表项的数量，降低 TLB 的压力，还能显著提高内存访问效率。通过自动化配置工具的应用，能够更加灵活地管理大页面，

根据系统的实际需求进行动态调整，进一步优化了系统的性能，提升了系统的整体稳定性和响应速度。

3) 文件系统功能多元化价值凸显

可视化监控、加密文件和版本控制等功能的引入，使文件系统在具备基本操作能力的基础上，安全性和易用性得到了显著提升。这些功能的加入，使得文件系统能够更好地满足现代应用场景对数据管理的多样化需求，为用户提供了更加可靠和便捷的文件管理服务。

设计存在的不足：

尽管团队在本次课程设计中取得了诸多成果，但在设计和实现过程中仍暴露出一些不足之处，需要在未来的研究和实践中加以改进。

1) TLB 预取算法有待完善

虽然 TLB 预取技术在提升系统性能方面发挥了重要作用，但在某些复杂的多核环境下，预取算法可能会出现不必要的内存占用情况。这主要是由于预取的准确性还不够高，需要进一步优化预取算法，提高其对内存访问模式的预测能力，减少不必要的预取操作，从而降低内存开销，提高系统的整体性能。

2) 大页配置灵活性仍需提升

现有的自动化大页配置工具虽然能够根据系统负载情况进行动态调整，但在一些特殊的应用场景下，其配置策略还不够精细。例如，对于某些对内存延迟和带宽要求极高的应用，现有的大页配置可能无法完全满足其性能需求，需要进一步研究更加智能和灵活的大页配置策略，以适应不同应用场景的多样化需求。

3) 文件系统扩展性面临挑战

虽然当前实现的文件系统能够满足基本的文件操作需求，并具备一些创新功能，但在面对大规模文件系统或分布式环境时，其扩展性和性能还存在一定的局限性。例如，在分布式文件系统中，文件的跨节点访问和管理效率还有待提高，需要进一步探索和引入先进的分布式技术，如一致性哈希、副本机制等，以提升文件系统在大规模环境下的可用性、容错能力和性能表现。

展望：

基于本次课程设计的经验和成果，团队对未来的研究和实践方向有了清晰的规

划，将从以下几个关键方面继续优化和完善系统设计：

1) TLB 预取算法优化

深入研究预取算法的优化策略，引入先进的机器学习算法和数据分析技术，通过对历史内存访问模式的学习和分析，更加准确地预测未来的内存需求，从而提高 TLB 预取的准确性和效率。同时，结合系统的实时运行状态，动态调整预取策略，避免不必要的内存占用，进一步提升系统的内存访问性能。

2) 增强大页配置灵活性

开发更加智能的大页配置工具，利用人工智能技术和系统性能监测数据，实现根据不同应用场景的自动优化配置。例如，在高并发场景下，能够自动优先分配大页，减少上下文切换开销，提高系统的并发处理能力；而在低延迟场景下，能够灵活调整大页的使用策略，确保系统的响应速度不受影响。通过这种智能化的配置方式，进一步提升系统的整体性能和适应性。

3) 提升文件系统扩展性

积极探索分布式文件系统的设计与实现，引入先进的分布式技术，如一致性哈希算法用于文件的分布式存储和定位，副本机制确保文件的高可用性和容错能力。同时，优化文件系统的性能，特别是在大规模数据存储和检索方面，通过数据分片、并行处理等技术，提高文件系统的读写速度和吞吐量。此外，加强对分布式文件系统的安全性研究，保障数据在分布式环境下的安全传输和存储。

通过以上持续的优化和改进措施，团队期望能够进一步提升 OpenEuler 操作系统的整体性能、稳定性和功能性，为未来的实际应用提供更加坚实和强大的技术支持，推动操作系统技术的发展和应用创新。

在本次课程设计中，团队成员张清玥、王宇骐和杨心怡紧密协作，充分发挥各自的专业优势，通过明确的分工、高效的沟通与协作机制以及资源共享和技术支持，确保了项目的顺利推进和高质量完成。团队定期召开会议，汇报工作进展、讨论问题并共同寻求解决方案；成员之间积极共享学习资料和技术文档，遇到技术难题时相互提供支持和帮助；在创新与优化方面，各成员不仅在各自负责的领域提出并实现了多项创新点，还积极开展跨领域协作，共同探索系统的整体优化方案，为项目的成功奠定了坚实的基础。