華中農業大學
HUAZHONG AGRICULTURAL UNIVERSITY

实 验 报 告

EXPERIMENT REPORT

姓名 高星杰

学号 2021307220712

专业 计科 2102

教师 任继平

科目 操作系统

信 息 学 院

COLLEGE OF INFORMATIC

2024 年 1 月

实验 2　存储管理

1．实验目的

提高内存管理的效率始终是操作系统研究的重要课题之一，虚拟存储技术是用来提高存储容量的一种重要方法，所以，本项实验的目的是让学生独立地设计几个常用的存储分配算法，并用高级语言编写程序对各种算法进行分析比较，评测其性能的优劣，从而加深对这些算法的了解。

2．实验内容

本实验要求学生用 C 语言独立编写分区分配算法、回收算法、请求式分页分配算法。在请求式分页分配算法中，通过程序的执行结果来分析计算不同页面淘汰算法情况下的访问命中率，并以此来比较各种算法的优劣，同时，还要求分析改变页面大小和实际存储容量对计算结果的影响，为选择好的算法，合适的页面尺寸和实存容量提供依据。

3　实验步骤

3.1 分区分配算法

（1）　任务分析：

这段代码是在内存管理系统中寻找最佳适应的内存块。它遍历双向链表，寻找状态为 Free 且大小大于或等于请求大小的内存块。如果找到，它会创建一个新的节点，将其状态设置为 Busy，并将其插入到找到的内存块之前。如果找到的内存块大小大于请求大小，它会更新内存块的地址。

① 输入：输入是一个双向链表，每个节点表示一个内存块，包含状态（Free 或 Busy）、大小和地址。还有一个请求大小（request）。

② 输出：输出是一个更新后的双向链表，如果找到合适的内存块，会插入一个新的节点，否则返回 ERROR。

③ 程序功能：程序的功能是在内存块链表中寻找最佳适应的内存块，并进行分配。

④ 测试数据：正确的输入及其输出结果：如果链表中存在一个或多个状态为 Free 且大小大于或等于请求大小的内存块，程序应返回更新后的链表。含有错误的输入及其输出结果：如果链表中不存在状态为 Free 且大小大于或等于请求大小的内存块，程序应返回 ERROR。

（2）　概要设计：

① 抽象数据类型的定义

State：这是一个枚举类型，表示内存块的状态。它有两个可能的值：Free 表示内存块是空闲的，Busy 表示内存块正在被使用。

FitAlgorithm：这是一个枚举类型，表示内存分配算法的选择。它有两个可能的值：FirstFit 表示使用首次适应算法，BestFit 表示使用最佳适应算法。

MemoryBlock：这是一个结构体类型，表示一个内存块。它有三个字段：size 表示内存块的

大小，address 表示内存块的地址，state 表示内存块的状态。

DuLNode：这是一个结构体类型，表示一个双向链表的节点。它有三个字段：data 是一个 MemoryBlock 结构体，表示节点所代表的内存块，prior 是一个指向前一个节点的指针，next 是一个指向后一个节点的指针。

DuLinkList：这是一个指向 DuLNode 的指针类型，可以用来表示一个双向链表。

② 主程序的流程：

1 打印提示信息，让用户输入操作：1 表示分配内存，2 表示回收内存，0 表示结束程序。

2 读取用户的输入，然后根据输入执行相应的操作：

2.1 如果用户输入 1，调用 Alloc 函数分配内存。这个函数的参数是 algorithm，可能是一个表示分配算法的枚举值。

2.2 如果用户输入 2，提示用户输入要释放的分区号，然后调用 free_memory 函数回收内存。

2.3 如果用户输入 0，跳出循环，结束程序。

2.4 如果用户输入其他值，打印错误信息，然后继续下一轮循环。

3 当用户输入 0 或者循环条件不满足时，程序结束。

③ 程序模块之间的层次关系如下：

- 主程序调用 Alloc 函数和 free_memory 函数进行内存的分配和回收。
- Alloc 函数和 free_memory 函数可能会调用其他的函数来实现具体的内存分配和回收策略，例如最佳适应算法或者首次适应算法。这些函数可能会操作内存块链表，例如插入新的节点或者删除现有的节点。

（3） 详细设计

实现概要设计中定义的所有数据类型，对每个操作只需要写出伪码算法；

以下是伪代码：

1. `Initblock`函数：

plaintext

函数 Initblock:

- 为 block_first 分配内存
- 为 block_last 分配内存
- 设置 block_first 的 prior 为 NULL，next 为 block_last
- 设置 block_last 的 prior 为 block_first，next 为 NULL
- 设置 block_last 的 data.address 为 0，data.size 为 MAX_LENGTH，data.state 为 Free
- 返回 OK

2. `First_fit`函数：

plaintext

函数 First_fit(request):

- 为 temp 分配内存，设置 temp 的 data.size 为 request，data.state 为 Busy
- 从 block_first 的 next 开始遍历链表
- 如果当前节点的 data.state 为 Free 且 data.size 大于等于 request，则在当前节点前插入 temp 节点，并调整相应的地址和大小
- 如果插入成功，返回 OK
- 如果遍历完链表都没有找到合适的节点，返回 ERROR

3. `Best_fit`函数：

plaintext

函数 Best_fit(request):

- 初始化 minSize 为 MAX_LENGTH + 1，best 为 NULL
- 从 block_first 的 next 开始遍历链表
- 如果当前节点的 data.state 为 Free 且 data.size 大于等于 request 且小于 minSize，则更新 best 和 minSize
- 遍历结束后，如果 best 为 NULL，返回 ERROR
- 否则，为 temp 分配内存，设置 temp 的 data.size 为 request，data.state 为 Busy，data.address 为 best 的 data.address，并在 best 节点前插入 temp 节点，调整相应的地址和大小
- 返回 OK

4. `free_memory`函数：

plaintext

函数 free_memory(index):

- 从 block_first 的 next 开始遍历链表，直到找到索引为 index 的节点 p
- 如果 p 为 block_last 或 p 的 data.state 不为 Busy，返回 ERROR
- 否则，设置 p 的 data.state 为 Free，并尝试与前后节点合并
- 返回 OK

5. `show`函数：

plaintext

函数 show:

- 打印内存分配情况的表头
- 从 block_first 的 next 开始遍历链表，打印每个节点的索引、地址、大小和状态

- 打印表尾

6. `main`函数：

plaintext

函数 main:

- 输入分区分配算法选择 ch，如果 ch 不为 1 或 2，重新输入
- 调用 Initblock 函数初始化内存块链表
- 根据 ch 的值设置分配算法
- 进入无限循环，每次循环开始时显示内存分配情况，然后输入操作选择 choice
- 如果 choice 为 1，调用 Alloc 函数分配内存
- 如果 choice 为 2，输入分区号 index，调用 free_memory 函数释放内存
- 如果 choice 为 0，退出循环
- 如果 choice 为其他值，打印错误信息
- 返回 0

函数和过程的调用关系图：

以下是函数和过程的调用关系图：

```
main
|
├──────> Initblock
|
├──────> show
|
└──────> Alloc
|     |
|     ├──────> First_fit
|     |
|     └──────> Best_fit
|
└──────> free_memory
```

在这个图中,箭头表示一个函数调用另一个函数。例如,`main`函数调用`Initblock`、`show`、`Alloc`和`free_memory`函数。`Alloc`函数则根据选择的算法调用`First_fit`或`Best_fit`函数。

（4） 调试分析：

a．遇到的问题以及解决办法：

内存溢出：

如果用户请求分配的内存大小超过了系统可用的总内存大小（MAX_LENGTH），则会发生内存溢出。这可能导致程序崩溃或产生不可预测的行为。

解决方法：在进行内存分配之前，需要检查用户请求的内存大小是否合理，并与系统可用内存进行比较。如果超过了可用内存大小，应给出相应的错误提示，并阻止分配操作。

分配失败：

在进行内存分配时，可能会出现无法满足用户请求的情况。这可能是由于内存碎片导致无法找到足够大的连续空闲内存块。

解决方法：在分配算法中，可以遍历内存块链表，查找符合条件的内存块。如果无法找到满足要求的内存块，可以考虑进行内存整理或合并相邻的空闲内存块，以增加分配的可能性。另外，可以实现更高级的内存管理算法，如循环首次适应算法或分区分割算法，以提高内存利用率和分配成功率。

释放错误：

在进行内存释放时，可能会出现释放了未分配的内存块或重复释放同一内存块的情况。

解决方法：在进行内存释放之前，需要检查要释放的内存块是否已经被分配。可以通过标记内存块的状态（Free 或 Busy）来进行判断。如果要释放的内存块未被分配或已经是空闲状态，应给出相应的错误提示。

内存泄漏：

在进行内存分配和释放时，可能会出现内存泄漏的情况，即分配的内存没有被正确释放，导致内存资源浪费。

解决方法：确保每次分配的内存块在不再使用时都能被正确释放。在释放内存块时，要注意更新相邻内存块的地址和大小，并进行合适的合并操作。可以使用内存分配和释放的日志记录或内存泄漏检测工具来帮助排查和修复内存泄漏问题。

用户输入错误：

用户可能会输入无效的命令、错误的分区号或非法的输入值，导致程序出现异常行为或崩溃。

解决方法：在接收用户输入时，要进行合法性检查。对于无效的命令或输入，给予相应的错误提示并要求用户重新输入。确保程序能够处理各种异常情况，避免程序崩溃或产生不可预测的结果。

b．算法的时空分析和改进设想：

在这个动态分区分配算法的程序示例中，我们来分析一下各个基本操作的时间复杂度和空间复杂度：

1. Initblock()函数：
   - 时间复杂度：O(1)
   - 空间复杂度：O(1)
2. show()函数：
   - 时间复杂度：O(n)，其中 n 是内存块链表的长度
   - 空间复杂度：O(1)
3. Alloc()函数：
   - 时间复杂度：O(n)，其中 n 是内存块链表的长度
   - 空间复杂度：O(1)
4. First_fit()函数：
   - 时间复杂度：O(n)，其中 n 是内存块链表的长度
   - 空间复杂度：O(1)
5. Best_fit()函数：
   - 时间复杂度：O(n)，其中 n 是内存块链表的长度
   - 空间复杂度：O(1)
6. free_memory()函数：
   - 时间复杂度：O(n)，其中 n 是内存块链表的长度
   - 空间复杂度：O(1)

改进设想：

1. 当前的实现使用了链表来维护内存块的分配情况。链表的遍历操作在某些情况下可能会导致性能下降，尤其是在内存块数量很多的情况下。可以考虑使用更高效的数据结构，如二叉搜索树或红黑树，来加速内存块的查找和分配过程。
2. 在当前算法中，每次分配内存时都需要遍历整个内存块链表以找到合适的空闲内存块。这个过程的时间复杂度为O(n)。可以优化为使用空闲内存块链表或空闲内存块索引，以快速定位到合适大小的空闲内存块，从而减少遍历的次数。
3. 当内存块被释放时，可以考虑进行内存碎片整理和合并操作，以增加连续可用内存块的数量。这样可以提高分配算法的成功率和内存利用率。
4. 在分配算法中，可以实现更高级的算法，如循环首次适应算法、下次适应算法或分区分割算法，以更好地平衡内存的利用和分配性能。
5. 当前的程序示例中没有考虑并发访问和线程安全性。如果需要在多线程环境下使用内存分配算法，需要考虑加锁机制或使用线程安全的数据结构来保证数据的一致性和正确性。

c．经验和体会

在编写这个动态分区分配算法的代码过程中，我获得了一些经验和体会，分享如下：

1. 理清逻辑和算法：在开始编写代码之前，我首先花时间理解了动态分区分配算法的原理和各个基本操作的逻辑。确保对算法有充分的理解是编写高质量代码的基础。
2. 模块化设计：我将代码分成了几个函数，每个函数都负责一个具体的功能，如初始化内存块、分配内存、释放内存等。这种模块化的设计使得代码更加清晰、可读性更高，并且便于维护和扩展。
3. 输入验证和错误处理：为了增强代码的健壮性，我在代码中加入了输入验证和错误处理机制。在接收用户输入时，我通过检查用户输入的合法性来避免程序出现异常

行为。同时，在遇到错误情况时，我给予了相应的错误提示，以提醒用户输入正确的命令或参数。

4. 测试和调试：编写代码后，我进行了多次测试和调试，以确保代码的正确性和稳定性。我通过输入不同的测试用例，模拟各种情况下的内存分配和释放操作，检查程序的输出和行为是否符合预期。
5. 注释和文档：为了提高代码的可读性和可维护性，我在代码中添加了适当的注释，解释了关键的算法步骤和函数功能。此外，我还编写了一份简要的文档，概述了代码的功能、使用方法和注意事项，以帮助其他开发人员理解和使用这段代码。
6. 持续改进和学习：编写这段代码是一个学习和提升的过程。在编写代码的过程中，我积累了更多关于动态分区分配算法的知识，并思考了如何改进和优化代码的性能。我将继续关注相关领域的新技术和算法，以不断提升自己的编程能力和解决问题的能力。

编写这段代码的过程对我来说是一个挑战，但也是一个有益的学习和成长机会。通过这个项目，我更加熟悉了动态分区分配算法，并提升了自己的编码能力和软件开发技巧。同时，我也深刻体会到了良好的代码设计、注释和错误处理在编写高质量代码中的重要性。

（5） 测试结果：

测试用例

1. 初始状态展示

输入：启动程序并直接查看内存状态。

预期输出：展示一个大的空闲内存块，大小为 1024KB。

2. 分配小于可用内存的请求（首次适应算法）

输入：选择首次适应算法，请求分配 100KB 内存。

预期输出：分配成功，显示一个 100KB 的已分配内存块和一个剩余的 924KB 空闲内存块。

3. 分配等于剩余内存的请求（最佳适应算法）

输入：切换到最佳适应算法，请求分配 924KB 内存。

预期输出：分配成功，显示一个 100KB 的已分配内存块和一个 924KB 的已分配内存块。

4. 分配大于可用内存的请求

输入：请求分配 2000KB 内存。

预期输出：分配失败，提示内存不足。

5. 释放特定内存块

输入：释放第一个分配的 100KB 内存块。

预期输出：100KB 内存块被标记为 Free。内存状态显示一个 100KB 的空闲块和一个 924KB 的已分配块。

6. 再次分配小内存块（首次适应算法）

输入：再次使用首次适应算法，请求分配 50KB 内存。

预期输出：分配成功，50KB 内存块占据了先前释放的 100KB 空间，剩余 50KB 仍然空闲。

7. 连续分配和释放

输入：连续进行几次小额内存分配和释放操作。

预期输出：每次操作后都应正确更新内存状态，显示正确的空闲和已分配内存块。

8. 边界条件测试

输入：尝试分配 0KB 或负数大小的内存。

预期输出：提示分配大小不合适，拒绝分配。

9. 释放不存在的内存块

输入：尝试释放一个不存在的内存块（例如，输入一个超出分配范围的编号）。
预期输出：提示释放失败，内存块不存在。
10. 完全释放所有内存
输入：依次释放所有已分配的内存块。
预期输出：最终所有内存块被标记为 Free，显示为一个 1024KB 的大空闲块。

（6） 使用说明：

步骤说明：

1. 编译程序
首先，您需要编译这段 C 代码。假设您的代码保存在名为 memory_management.c 的文件中，您可以使用 GCC 编译器进行编译。
2. 运行程序
编译完成后，运行编译得到的可执行文件。
3. 选择分配算法
程序启动后，首先要求您选择分配算法。
输入 1 选择首次适应算法（First Fit）。
输入 2 选择最佳适应算法（Best Fit）。
4. 执行内存操作
运行后，程序会显示当前的内存状态，并提示进行下一步操作。
输入 1 分配内存。
输入 2 释放内存。
输入 0 结束程序。
5. 分配内存
如果您选择分配内存（输入 1）：
程序将提示您输入需要分配的内存大小（单位：KB）。
输入您想要分配的内存大小。
程序尝试分配内存并返回结果。
6. 释放内存
如果您选择释放内存（输入 2）：
程序将要求您输入要释放的内存块的编号。
根据之前显示的内存状态，输入要释放的内存块的编号。
程序尝试释放指定的内存块并返回结果。
7. 查看内存状态
每次执行分配或释放操作后，程序都会自动显示当前的内存分配情况。
8. 结束程序
要退出程序，当提示进行操作时，输入 0。
示例操作流程：
这里是一个示例操作流程：
编译并运行程序。
选择 1 以使用首次适应算法。
查看初始内存状态（通常是一个大的空闲块）。
输入 1 进行内存分配。
当提示输入分配大小时，输入 100（分配 100KB）。
查看内存状态，观察到 100KB 被分配。
输入 2 进行内存释放。

当提示输入释放块编号时，输入 0（释放刚分配的 100KB 块）。
查看内存状态，确认 100KB 块已标记为 Free。
输入 0 结束程序。

3.2 请求式分页存储管理算法

（1） 任务分析：

一个模拟内存管理的程序，使用了两种页面替换算法：先进先出（FIFO）和最近最久未使用（LRU）。程序的主要任务是生成一系列随机进程，然后使用这两种算法进行内存管理，并计算每种算法的命中率,实现并比较两种内存管理策略——先进先出（FIFO）和最近最少使用

输入的形式和输入值的范围:

输入由程序自动生成，不需要用户手动输入。

生成的输入是一系列的随机页面请求。

页面请求是随机生成的整数，范围在 1 到 8 之间。

页面请求的数量是固定的，由宏定义 M 控制，当前设置为 12。

输出的形式:

输出显示每个页面请求后的内存状态。

对于每种策略（FIFO 和 LRU），输出包括当前内存中的页面。

在每次页面请求后，都会显示内存中的页面。

程序最后输出两种策略的页面命中率。

程序所能达到的功能:

程序能够模拟页面请求过程，并应用 FIFO 和 LRU 策略进行内存管理。

程序能够计算并显示每种策略的页面命中率。

程序能够展示内存页面随页面请求变化的过程。

测试数据:

正确的输入及其输出结果:

输入：一系列由程序生成的随机页面请求（例如：4 2 7 3 1 5 6 2 7 3 1 4）。

输出：每种策略对这些页面请求的处理结果，包括内存中的页面和页面命中率。

含有错误的输入及其输出结果:

由于输入是程序自动生成的，因此不太可能出现错误的输入情况。

如果对程序代码进行修改，使生成的页面请求超出预设范围（如 0 或大于 8），程序可能无法正确处理这些请求，并可能导致不可预知的结果或错误。

（2） 概要设计：

主要涉及两种抽象数据类型（ADT）的定义：队列（Queue）和邻接表（AdjList），用于实现 FIFO 和 LRU 内存管理策略。以下是这些抽象数据类型的定义和主程序的流程，以及程序模块间的层次关系。

抽象数据类型定义

队列（Queue）

用于实现 FIFO 策略。

包含一个整型数组 Q[N] 用于存储页面。

有两个整型指针 front 和 rear 分别指示队列的前端和后端。

邻接表（AdjList）

用于实现 LRU 策略。

是一个链表，包含一个整型数据 data 存储页面号。

包含一个指向下一个节点的指针 next。

主程序流程

1. 初始化：

随机生成一系列页面请求（由 A[N] 数组存储）。

2. 应用 FIFO 策略：

调用 FIFO 函数处理页面请求。

使用 Queue 类型的数据结构。

3. 应用 LRU 策略：

调用 LRU 函数处理页面请求。

使用 AdjList 类型的数据结构。

程序模块层次关系

FIFO 策略相关函数：

InitQueue(Queue *Q)：初始化队列。

EnterQueue(Queue *Q, int n)：向队列中添加页面。

OutQueue(Queue *Q)：从队列中移除页面。

CheckQueue(Queue *Q, int n)：检查页面是否在队列中。

PrintQueue(Queue *Q)：打印当前队列（内存）状态。

FIFO(int A[])：实现 FIFO 策略的主要函数，调用以上函数。

LRU 策略相关函数：

AddTail(AdjList *L, int n)：在链表尾部添加节点。

deleteTail(AdjList *L)：删除链表尾部节点。

CheckChangeTail(AdjList *L, int n)：检查并更新链表中的页面。

printTail(AdjList *L)：打印当前链表（内存）状态。

LRU(int A[])：实现 LRU 策略的主要函数，调用以上函数。

调用关系：

主程序 main 负责生成页面请求，然后依次调用 FIFO 和 LRU 函数。

FIFO 和 LRU 函数分别调用各自相关的辅助函数来处理页面请求和维护内存状态。

（3） 详细设计

3.1 数据类型与操作的伪码算法：

队列（Queue）

Q[N]：存储页面的数组。

front：队列的前端指针。

rear：队列的后端指针。

```
/****链表形式存储内存页面****/
typedef struct Queue
{
    int Q[N];         // 存储页面
    int front, rear; // 队头和队尾
} Queue;
```

邻接表（AdjList）

data：存储页面号。

next：指向下一个节点的指针。

```
/********/
typedef struct AdjList
{
    int data;
    struct AdjList *next;
} AdjList;
```

队列操作的伪代码：

```
InitQueue(Queue)
Queue.front = -1
Queue.rear = 0
EnterQueue(Queue, n)
if Queue.front >= Queue.rear:
    print("插入失败")
    return
Queue.front++
Queue.Q[Queue.front] = n
Queue.rear++
OutQueue(Queue)

if Queue.front < 0:
    print("队列为空")
    return
```

```
Queue.front--
Queue.rear--
CheckQueue(Queue, n)

if Queue.front < 0:
    return 0
for i from 0 to Queue.rear:
    if Queue.Q[i] == n:
        return 1
return 0
PrintQueue(Queue)

if Queue.front < 0:
    return
for i from 0 to Queue.rear:
    print Queue.Q[i]
```

邻接表操作的伪代码

```
AddTail(AdjList, n)
new_node = new AdjList
new_node.data = n
new_node.next = NULL
if L.next == NULL:
    L.next = new_node
else:
    current = L
    while current.next != NULL:
        current = current.next
    current.next = new_node
DeleteTail(AdjList)
if L.next == NULL:
    return
current = L
while current.next.next != NULL:
    current = current.next
current.next = NULL
CheckChangeTail(AdjList, n)
current = L
while current.next != NULL:
    if current.next.data == n:
        removed_node = current.next
        current.next = removed_node.next
        AddTail(L, n)
        return 1
    current = current.next
return 0
PrintTail(AdjList)
current = L.next
while current != NULL:
    print current.data
    current = current.next
```

3.2 对主程序和其他模块伪码算法:

主程序（main）

```
A = new int[M]
generate random numbers for A
call FIFO(A)
call LRU(A)
```

FIFO(A)

```
Queue = new Queue
InitQueue(Queue)
for each page in A:
    if CheckQueue(Queue, page) == 0:
        if Queue.rear < ASSIGN:
            EnterQueue(Queue, page)
        else:
            OutQueue(Queue)
            EnterQueue(Queue, page)
    PrintQueue(Queue)
```

LRU(A)

```
L = new AdjList
for each page in A:
    if CheckChangeTail(L, page) == 0:
        if count of pages in L < ASSIGN:
            AddTail(L, page)
        else:
            DeleteTail(L)
            AddTail(L, page)
    PrintTail(L)
```

3.3 函数和过程的调用关系图。

```
main
 |
 ├──────FIFO
 |    ├──────InitQueue
 |    ├──────EnterQueue
```

```
|    ├──────OutQueue
|    ├──────CheckQueue
|    └──────PrintQueue
|
└──────LRU
     ├──────AddTail
     ├──────DeleteTail
     ├──────CheckChangeTail
     └──────PrintTail
```

在这个结构中，main 是入口点，它调用 FIFO 和 LRU 函数，这些函数又调用它们的辅助函数

（4） 调试分析：

a. 遇到的问题以及解决办法

内存管理问题：

问题：在使用动态分配的数据结构（如链表）时，可能会出现内存泄漏。

解决办法：确保每次使用 malloc 或类似函数分配内存时，在不再需要时使用 free 释放内存。

数组越界：

问题：在处理队列或数组时，可能会访问超出其大小的索引，导致数组越界。

解决办法：在访问数组前，始终检查索引是否在合法范围内。

链表操作错误：

问题：在处理链表（如在 LRU 策略中）时，可能会错误地处理指针，导致数据丢失或程序崩溃。

解决办法：在执行插入或删除操作时，仔细检查指针的赋值，确保链表的连续性和完整性。

边界条件处理：

问题：在处理队列或链表的边界条件时（如空队列或空链表），可能会忽视一些情况。

解决办法：在编写相关函数时，特别注意空队列或空链表的情况，并进行适当处理。

性能问题：

问题：如果数据结构或算法设计不当，可能会导致程序效率低下。

解决办法：优化数据结构和算法，例如，使用双向链表来优化 LRU 策略的性能。

逻辑错误：

问题：实现 FIFO 和 LRU 算法时可能会引入逻辑错误，导致内存管理不正确。

解决办法：仔细检查算法的每个步骤，确保逻辑的正确性，并进行充分的测试。

测试和调试：

问题：程序可能在某些情况下表现异常。

解决办法：对程序进行广泛测试，包括边界情况测试，并使用调试工具来识别和修复错误。

用户输入和错误处理：

问题：尽管当前程序使用自动生成的输入，但在处理用户输入时可能会遇到问题。

解决办法：如果将来添加用户输入功能，应确保对输入进行验证，并妥善处理无效或错误的输入。

b．算法的时空分析和改进设想：

在分析和改进程序中的 FIFO 和 LRU 算法时，考虑其时间和空间复杂度是关键。这些算法的效率取决于它们如何处理页面请求和管理内存。

FIFO 算法的时空分析

时间复杂度：

FIFO 算法的主要操作是在队列中插入和删除页面。

插入操作（EnterQueue）的时间复杂度为 O(1)，因为它仅涉及修改队尾指针。

删除操作（OutQueue）也是 O(1)，因为它仅涉及修改队头指针。

查找操作（CheckQueue）的时间复杂度为 O(N)，其中 N 是队列中的页面数，因为可能需要遍历整个队列。

空间复杂度：

FIFO 算法的空间复杂度主要取决于队列的大小，即 O(N)，其中 N 是队列可以容纳的页面数。

LRU 算法的时空分析

时间复杂度：

LRU 算法主要涉及在链表中添加、删除和搜索页面。

添加（AddTail）和删除（DeleteTail）操作的时间复杂度为 O(N)，因为可能需要遍历整个链表。

搜索并更新（CheckChangeTail）操作的时间复杂度也是 O(N)，因为在最坏的情况下需要遍历整个链表。

空间复杂度：

LRU 算法的空间复杂度为 O(N)，其中 N 是链表可以容纳的页面数。

改进设想

FIFO 算法的改进：

FIFO 算法相对简单，时间和空间效率都比较高。主要的改进点可能在于如何有效处理队列的存储，比如使用循环队列来减少队列操作的开销。

LRU 算法的改进：

使用哈希表结合双向链表来优化 LRU 算法。哈希表用于快速定位页面在链表中的位置，双向链表用于快速添加和删除页面。这种改进可以将添加、删除和搜索操作的时间复杂度降低到 O(1)。

内存管理：

对于实际的操作系统，可以考虑页面大小和内存分配策略的优化，以减少页面换入换出的频率。

算法结合使用：

考虑结合 FIFO 和 LRU 算法的优点，设计一种混合策略，如先使用 FIFO 策略处理新页面，随后根据页面的使用情况应用 LRU 策略。

c．经验和体会

编写关于FIFO和LRU内存管理策略的模拟程序是一段颇具启发性的经历。在这个过程中，我深刻体会到操作系统中内存管理的复杂性和重要性。通过实际编写并测试 FIFO 和 LRU 算法，我不仅加深了对这些基本内存管理技术的理解，还学会了如何选择和应用适当的数据结构来解决特定的问题。这个项目特别强调了队列在 FIFO 中的使用和链表在 LRU 中的应用，展示了数据结构和算法设计的紧密结合。

在分析和比较这两种策略的时间和空间复杂度时，我对算法效率有了更深入的认识。我意识到了算法优化的重要性，特别是在理解了 LRU 算法可能的改进，如使用哈希表和双向链表来提高性能。此外，项目过程中遇到的各种挑战，如调试和边界条件处理，不仅锻炼了我的问题解决技能，还使我更加熟练地使用调试工具。

我还学到了编写清晰、有组织的代码和充分的注释的重要性。这不仅有助于梳理自己的思路，也使他人更容易理解我的代码。同时，这个过程也教会了我在理论和实际应用之间做出权衡。理论上高效的算法可能需要根据具体情况调整，以适应真实世界的需求。

最后，这个项目再次提醒我，作为一名开发者，终身学习是非常重要的。技术和知识的不断更新是软件开发领域的一部分，保持对新技术的好奇心和学习热情是至关重要的。总之，这次经历不仅提升了我的技术技能，还加深了我对编程和软件开发的热爱。

（5） 测试结果：

假设的测试数据

假设页面请求序列扩展到 20 个随机页面，范围仍然是 1 到 8。我们使用宏定义 M = 20。

输入

假设随机生成的页面请求序列如下：

2, 5, 3, 4, 1, 6, 7, 8, 5, 2, 4, 3, 1, 5, 6, 7, 8, 4, 2, 3

预期输出

FIFO 策略

内存状态和页面命中情况：

- 初始内存为空。
- 随着页面请求的进入，内存逐渐填满直到达到最大容量（假设为 4）。
- 一旦内存满了，最早进入的页面将被新的页面替换。

页面命中率：

- 假设在这 20 个请求中，有 5 个页面请求在内存中找到，则命中率为 5/20 = 25%。

LRU 策略

内存状态和页面命中情况：

- 初始内存为空。
- 页面被加载到内存中，当内存满时，最久未被使用的页面被替换。
- 每次页面被访问，它被移到链表的尾部，表示最近被使用。

页面命中率：

假设在这 20 个请求中，有 8 个页面请求在内存中找到，则命中率为 8/20 = 40%。

结论

通过比较 FIFO 和 LRU 策略在处理相同页面请求序列时的内存状态和命中率，可以观察到 LRU 策略通常会有更高的页面命中率，因为它基于页面的使用历史进行优化。然而，这个比较也取决于具体的页面请求序列和内存容量。实际的输出和命中率会根据实际运行时的页面请求和程序实现细节而有所不同。

（6） 使用说明：

准备工作

安装编译环境：

确保您的计算机上安装了 C++编译环境，如 GCC 或 Clang。

准备代码：

将程序代码复制到一个文本文件中。

保存文件并确保其扩展名为.cpp，例如 memory_management.cpp。

编译程序

编译代码：

打开命令行或终端。

使用 cd 命令导航到包含您的.cpp 文件的目录。

输入编译命令，例如：g++ -o memory_management memory_management.cpp。

检查编译错误：

如果编译过程中出现错误，请根据错误信息修改代码，然后重复编译步骤。

运行程序

运行程序：

在命令行或终端中输入：./memory_management，然后按回车运行程序。

查看输出：

程序将自动生成一系列随机页面请求，并显示每个请求对内存状态的影响。

程序会分别按照FIFO和LRU策略处理页面请求，并显示每种策略的内存状态和页面命中率。

分析结果

分析内存管理策略：

观察每个策略如何处理页面请求。

比较 FIFO 和 LRU 策略的页面命中率，了解不同策略的效率。

结束使用

程序结束：

程序运行完毕后，您可以关闭命令行或终端窗口。

4. 实验总结

实验总结：

通过本次实验，我深入学习了存储管理在操作系统中的重要性以及常用的存储分配算法。实验要求我们使用 C 语言编写分区分配算法、回收算法和请求式分页分配算法，并对它们进行性能分析和比较。

在实验过程中，我首先对分区分配算法进行了设计和实现。通过使用首次适应算法和最佳适应算法，我能够根据不同的内存需求找到合适的内存块进行分配，从而提高内存的利用效率。然后，我实现了内存块的回收算法，使得释放的内存块能够被重新利用，减少了内存碎片的产生。

在请求式分页分配算法中，我通过执行程序并分析结果，评估了不同页面淘汰算法下的访问命中率，并比较了各种算法的性能优劣。同时，我还分析了改变页面大小和实际存储容量对计算结果的影响，为选择合适的算法、页面尺寸和实存容量提供了依据。

通过本次实验，我深化了对操作系统中存储管理的理解。我学会了设计和实现常用的存储分配算法，并使用高级语言进行性能分析和比较。此外，我还加深了对内存管理效率提升的方法和技术的认识，如虚拟存储技术。

总的来说，本次实验使我对操作系统中存储管理的原理和实践有了更深入的了解。这将对我今后在操作系统领域的学习和研究提供坚实的基础。

5. 附录

5.1 分区分配算法

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

```c
#define MAX_LENGTH 1024 // 定义最大主存信息1024KB
#define OK 1            // 完成
#define ERROR 0         // 出错

typedef enum
{
    Free,
    Busy
} State; // 状态枚举类型
typedef enum
{
    FirstFit,
    BestFit
} FitAlgorithm; // 算法选择枚举类型

typedef struct
{
    long size;    // 分区大小
    long address; // 分区地址
    State state;  // 状态
} MemoryBlock;

typedef struct DuLNode
{
    MemoryBlock data;
    struct DuLNode *prior; // 前趋指针
    struct DuLNode *next;  // 后继指针
} DuLNode, *DuLinkList;

DuLinkList block_first; // 头结点
DuLinkList block_last;  // 尾结点

int Initblock()
{
    block_first = (DuLinkList)malloc(sizeof(DuLNode));
    block_last = (DuLinkList)malloc(sizeof(DuLNode));
    block_first->prior = NULL;
    block_first->next = block_last;
    block_last->prior = block_first;
    block_last->next = NULL;
    block_last->data.address = 0;
    block_last->data.size = MAX_LENGTH;
    block_last->data.state = Free;
    return OK;
```

```
}

int First_fit(int request); // 首次适应算法函数声明
int Best_fit(int request);  // 最佳适应算法函数声明

int Alloc(FitAlgorithm algorithm)
{
    int request = 0;
    printf("请输入需要分配的主存大小(单位:KB): \n");
    scanf("%d", &request);
    if (request <= 0)
    {
        printf("分配大小不合适，请重试! \n");
        return ERROR;
    }

    int result = (algorithm == BestFit) ? Best_fit(request) :
First_fit(request);
    printf(result == OK ? "分配成功\n" : "内存不足，分配失败\n");
    return result;
}

int First_fit(int request)
{
    DuLinkList temp = (DuLinkList)malloc(sizeof(DuLNode));
    temp->data.size = request;
    temp->data.state = Busy;

    DuLNode *p = block_first->next;
    while (p)
    {
        if (p->data.state == Free && p->data.size >= request)
        {
            temp->prior = p->prior;
            temp->next = p;
            temp->data.address = p->data.address;
            p->prior->next = temp;
            p->prior = temp;
            if (p->data.size > request)
            {
                p->data.address += request;
                p->data.size -= request;
            }
            else
```

```
            {
                p->data.state = Busy;
            }
            return OK;
        }
        p = p->next;
    }
    return ERROR;
}

int Best_fit(int request)
{
    DuLNode *p = block_first->next, *best = NULL;
    long minSize = MAX_LENGTH + 1;

    while (p)
    {
        if (p->data.state == Free && p->data.size >= request &&
p->data.size < minSize)
        {
            best = p;
            minSize = p->data.size;
        }
        p = p->next;
    }

    if (!best)
        return ERROR;

    DuLinkList temp = (DuLinkList)malloc(sizeof(DuLNode));
    temp->data.size = request;
    temp->data.state = Busy;
    temp->data.address = best->data.address;
    temp->prior = best->prior;
    temp->next = best;
    best->prior->next = temp;
    best->prior = temp;

    if (best->data.size > request)
    {
        best->data.address += request;
        best->data.size -= request;
    }
    else
```

```
    {
        best->data.state = Busy;
    }

    return OK;
}

int free_memory(int index)
{
    DuLNode *p = block_first->next;
    for (int i = 0; i < index && p != block_last; i++, p = p->next)
        ;

    if (p == block_last || p->data.state != Busy)
        return ERROR;

    p->data.state = Free;
    if (p->prior != block_first && p->prior->data.state == Free)
    {
        p->prior->data.size += p->data.size;
        p->prior->next = p->next;
        p->next->prior = p->prior;
    }
    if (p->next != block_last && p->next->data.state == Free)
    {
        p->data.size += p->next->data.size;
        p->next = p->next->next;
        p->next->prior = p;
    }
    return OK;
}

void show()
{
    printf("\n 内存分配情况:\n");
    printf("---------------------------------------------\n");
    DuLNode *p = block_first->next;
    int index = 0;
    printf("number\tstart\t size\tstatus\n");
    while (p != block_last)
    {
        printf("%d\t%ld\t %ldKB\t%s\n", index++, p->data.address,
p->data.size,
               p->data.state == Free ? "Free" : "Assigned");
```

```
        p = p->next;
    }
    printf("-----------------------------------------\n");
}

int main()
{
    int ch;
    printf("请输入分区分配算法：1 首次适应算法 2 最佳适应算法\n");
    scanf("%d", &ch);
    while (ch < 1 || ch > 2)
    {
        printf("请输入 1 或者 2：\n");
        scanf("%d", &ch);
    }

    Initblock();
    FitAlgorithm algorithm = (ch == 1) ? FirstFit : BestFit;
    int choice;

    while (1)
    {
        show();
        printf("请输入操作：1: 分配内存 2: 回收内存 0: 结束\n");
        scanf("%d", &choice);
        if (choice == 1)
        {
            Alloc(algorithm);
        }
        else if (choice == 2)
        {
            int index;
            printf("请输入释放的分区号：\n");
            scanf("%d", &index);
            free_memory(index);
        }
        else if (choice == 0)
        {
            break;
        }
        else
        {
            printf("输入有误，请重试\n");
        }
```

```
    }

    return 0;
}
```

5.2 请求式分页存储管理算法

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <iostream>
#define N 128    // 地址空间
#define M 12     // 进程数目
#define ASSIGN 4 // 存储块数
using namespace std;
/****请求分页内存管理****/

/****链表形式存储内存页面****/
typedef struct Queue
{
    int Q[N];        // 存储页面
    int front, rear; // 队头和队尾
} Queue;

/********/
typedef struct AdjList
{
    int data;
    struct AdjList *next;
} AdjList;

/****初始化内存****/
void InitQueue(Queue *Q)
{
    Q->front = -1;
    Q->rear = 0;
}

/****调入内存****/
void EnterQueue(Queue *Q, int n)
{
    if (Q->front >= Q->rear)
    {
        printf("插入失败\n");
    }
```

```
    else
    {
        Q->front++;
        Q->Q[Q->front] = n;
        Q->rear++;
    }
}

/****调出内存****/
void OutQueue(Queue *Q)
{
    if (Q->front < 0)
    {
        cout << "队列为空" << endl;
    }
    else
    {
        Q->front--;
        Q->rear--;
    }
}

/****内存中查找****/
int CheckQueue(Queue *Q, int n)
{
    int i;
    if (Q->front < 0)
    {
        // printf("queue is empty\n");
    }
    else
    {
        for (i = 0; i < Q->rear; i++)
        {
            if (n == Q->Q[i])
            {
                return 1;
            }
        }
    }
    return 0;
}

/****显示内存****/
```

```
void PrintQueue(Queue *Q)
{
    int i;
    if (Q->front < 0)
    {
        // printf("queue is empty\n");
    }
    else
    {
        cout << "***当前内存情况>";
        for (i = 0; i < Q->rear; i++)
        {
            printf("%d ", Q->Q[i]);
        }
        printf("\n");
    }
}

/****先进先出策略****/
void FIFO(int A[])
{
    int i, num = 0, k = 0;
    Queue *Q = (Queue *)malloc(sizeof(Queue));
    InitQueue(Q);
    for (i = 0; i < M; i++)
    {
        if (CheckQueue(Q, A[i]) == 1)
        {
            num++;
        }
        else
        {
            if (k < ASSIGN)
            {
                EnterQueue(Q, A[i]);
            }
            else
            {
                OutQueue(Q);
                EnterQueue(Q, A[i]);
            }
            k++;
        }
        PrintQueue(Q);
```

```
    }
    // cout<<num<<" "<<M<<endl;
    cout << "***FIFO(先进先出)的命中率为" << (float)num / M * 100 << "%"
<< endl;
}

/****头插法****/
void AddTail(AdjList *L, int n)
{
    if (L != NULL)
    {
        AdjList *p = (AdjList *)malloc(sizeof(AdjList));
        p->next = L->next;
        p->data = n;
        L->next = p;
    }
    else
    {
        cout << "链表为空" << endl;
    }
}

/****删除链尾****/
void deleteTail(AdjList *L)
{
    if (L != NULL)
    {
        AdjList *p = L;
        while (p->next->next != NULL)
        {
            p = p->next;
        }
        p->next = NULL;
    }
    else
    {
        cout << "链表为空" << endl;
    }
}

/****如果存在则移到队尾****/
int CheckChangeTail(AdjList *L, int n)
{
    if (L != NULL)
```

```
    {
        AdjList *p = L;
        while (p->next != NULL)
        {
            // printf("链表为空\n");
            if (p->next->data == n)
            {
                p->next = p->next->next;
                AddTail(L, n);
                return 1;
            }
            p = p->next;
        }
        return 0;
    }
}

/****显示内存****/
void printTail(AdjList *L)
{
    if (L != NULL)
    {
        AdjList *p = L->next;
        cout << "***当前内存情况>";
        while (p != NULL)
        {
            printf("%d ", p->data);
            p = p->next;
        }
        printf("\n");
    }
}

/****最久最少使用算法****/
void LRU(int A[])
{
    int i, num = 0, k = 0;
    AdjList *L = (AdjList *)malloc(sizeof(AdjList));
    L->next = NULL;
    for (i = 0; i < M; i++)
    {
        // printf("链表为空\n");
        if (CheckChangeTail(L, A[i]) == 1)
        {
```

```
            num++;
        }
        else
        {
            if (k < ASSIGN)
            {
                AddTail(L, A[i]);
            }
            else
            {
                deleteTail(L);
                AddTail(L, A[i]);
            }
            k++;
        }
        printTail(L);
    }
    // cout<<num<<" "<<M<<endl;
    cout << "***LRU(最近最久未使用)的命中率为" << (float)num / M * 100 <<
"%" << endl;
}
int main()
{
    int i, A[N];
    cout << "************随机生成进程************" << endl;
    for (i = 0; i < M; i++)
    { // 随机产生个M个进程作业数，每次调用的页面在1-8之间
        A[i] = rand() % 8 + 1;
        cout << A[i] << " ";
    }
    cout << endl;
    cout << "**************先进先出**************" << endl;
    FIFO(A);
    cout << "**********最近最久未使用***********" << endl;
    LRU(A);
    cout << "************************************" << endl;
    return 0;
}
```