

北京交通大学操作系统实验报告

姓名：程维森

学号：21231264

I. 引言

a) 实验目的

- 理解内存分配策略（如首次适应、最佳适应和最差适应策略）的工作原理。
- 学习如何实现内存分配和释放功能，并理解其中的算法和数据结构。
- 掌握内存碎片的概念，以及如何通过压缩内存减少碎片。
- 增强编程能力和调试技巧，通过实践加深对内存管理机制的认识。

b) 实验背景

在计算机科学领域，内存管理是一个关键的主题，尤其是在操作系统和软件开发中。程序在执行时需要分配和释放内存，而内存管理算法的选择直接影响到计算机系统的性能和资源利用率。在本次实验中，我们将探讨并实现不同的内存分配算法，具体来说，是基于连续内存分配的算法。

计算机的内存是一个连续的存储区域，每个程序在执行时需要一定大小的内存空间来存储数据和指令。内存管理的任务包括将可用内存空间分配给不同的程序，确保每个程序能够得到足够的内存以运行，并在程序运行结束后及时释放内存以便其他程序使用。

本实验中，我们面临的挑战包括如何高效地管理内存，以及如何选择合适的内存分配策略。在现实世界的应用场景中，我们可能会面临多个程序同时运行的情况，这就需要考虑如何合理地分配和释放内存，避免出现内存碎片问题。内存碎片是指内存中的一些小块空闲区域，它们太小以至于不能满足大程序的内存需求，这会导致内存资源的浪费。

在本实验中，我们将实现三种不同的内存分配策略：首次适应算法（First Fit）、最佳适应算法（Best Fit）、最坏适应算法（Worst Fit）。这些算法有各自的优势和限制，我们将通过实验数据和分析来比较它们的性能，以便更好地理解 and 选择合适的内存分配策略。

通过本次实验，我们将学习如何设计和实现内存分配算法，加深对内存管理原理的理解，提高问题解决和编程能力，为今后的系统设计和开发提供有益的经验。

c) 实验任务概述

- **请求内存:** 程序将接收到形如“allocator>RQ P0 40000 W”的请求，表示一个新进程 P0 请求分配 40000 字节的内存，且采用“Worst Fit”策略。你的程序需要根据所选策略，在可用的内存块中为该请求找到一个连续的内存区域。
- **释放内存:** 当一个进程完成任务时，它将释放所占用的内存块。释放内存的请求将以形如“allocator>RL P0”的形式呈现，表示进程 P0 释放其已分配的内存。如果被释放的内存块与相邻的空闲内存块相邻，你的程序需要将它们合并为一个更大的空闲内存块。
- **内存压缩:** 当存在多个不连续的空闲内存块时，可以通过“allocator>C”命令将它们合并为一个更大的空闲内存块。内存压缩命令将合并所有未使用的内存块，以提供更大的连续内存空间。
- **内存状态报告:** 程序将响应“allocator>STAT”命令，输出当前内存的分配状态。报告应该包括已分配内存块的地址范围，以及空闲内存块的地址范围。

II. 设计与实现

a) 设计思路

- **数据结构选择:** 我们可以使用结构体来表示内存块，结构体包含起始地址、大小和分配状态等信息。使用结构体可以方便地管理内存块的状态，并且支持动态地插入和删除内存块。

```
struct MemoryBlock
{
    int start;
    int size;
    bool allocated;

    MemoryBlock(int s, int sz, bool a) : start(s), size(sz), allocated(a) {}
};
```

- **内存分配策略:** 实现首次适应算法 (First Fit)、最佳适应算法 (Best Fit) 和最坏适应算法 (Worst Fit) 的内存分配逻辑。对于不同的策略，我们需要遍历内存块列表，找到符合要求的内存块进行分配。

First Fit: 遍历内存块列表，找到第一个大小满足要求的空闲块进行分配。

Best Fit: 遍历内存块列表，找到最小但仍然满足要求的空闲块进行分配。

Worst Fit: 遍历内存块列表，找到最大的空闲块进行分配。

```

switch (strategy)
{
case 'F': // First Fit
    for (int i = 0; i < memoryBlocks.size(); ++i)
    {
        if (!memoryBlocks[i].allocated && memoryBlocks[i].size >= size)
        {
            newBlock.start = memoryBlocks[i].start;
            indexToInsert = i;
            break;
        }
    }
    break;
case 'B': // Best Fit
    {
        int bestSize = std::numeric_limits<int>::max();
        for (int i = 0; i < memoryBlocks.size(); ++i)
        {
            if (!memoryBlocks[i].allocated && memoryBlocks[i].size >= size && memoryBlocks[i].size < bestSize)
            {
                bestSize = memoryBlocks[i].size;
                newBlock.start = memoryBlocks[i].start;
                indexToInsert = i;
            }
        }
    }
    break;
case 'W': // Worst Fit
    {
        int worstSize = -1;
        for (int i = 0; i < memoryBlocks.size(); ++i)
        {
            if (!memoryBlocks[i].allocated && memoryBlocks[i].size > worstSize)
            {
                worstSize = memoryBlocks[i].size;
                newBlock.start = memoryBlocks[i].start;
                indexToInsert = i;
            }
        }
    }
    break;
default:
    std::cout << "未知策略。请使用 'F', 'B', 或 'W'。\\n";
    return;
}

```

- **内存释放和合并：** 当释放内存时，需要将对应的内存块标记为未分配，并且考虑将相邻的未分配内存块合并成更大的内存块。

合并相邻内存块： 在释放内存后，检查前后相邻的内存块，如果它们都未分配，可以将它们合并成一个更大的内存块。

```

// 内存释放方法
void releaseMemory(int start)
{
    for (int i = 0; i < memoryBlocks.size(); ++i)
    {
        if (memoryBlocks[i].start == start && memoryBlocks[i].allocated)
        {
            // Mark the block as unallocated
            memoryBlocks[i].allocated = false;

            // Try to merge with the previous block if it's unallocated
            if (i > 0 && !memoryBlocks[i - 1].allocated)
            {
                memoryBlocks[i - 1].size += memoryBlocks[i].size;
                memoryBlocks.erase(memoryBlocks.begin() + i);
                i--; // Adjust the index after erasing
            }

            // Try to merge with the next block if it's unallocated
            if (i < memoryBlocks.size() - 1 && !memoryBlocks[i + 1].allocated)
            {
                memoryBlocks[i].size += memoryBlocks[i + 1].size;
                memoryBlocks.erase(memoryBlocks.begin() + i + 1);
            }
        }

        return; // Memory released and merged if necessary, exit the function
    }

    std::cout << "未找到起始地址为 " << start << " 的已分配内存块。\\n";
}

```

- **内存压缩：** 实现内存压缩功能，将多个不连续的空闲内存块合并为一个大的内存块。内存压缩可以提高内存的利用率。

合并所有未分配内存块： 当用户输入压缩命令时，遍历所有未分配的内存块，将它们合并成一个大的内存块。

```

void compactMemory()
{
    if (memoryBlocks.empty())
        return; // If no memory blocks, nothing to compact.

    // First, we sort the blocks by starting address to ensure adjacent blocks are next to each other.
    std::sort(memoryBlocks.begin(), memoryBlocks.end(), [](const MemoryBlock &a, const MemoryBlock &b)
        { return a.start < b.start; });

    for (int i = 0; i < memoryBlocks.size() - 1; ++i)
    {
        // If the current block is free and the next block is also free, merge them.
        if (!memoryBlocks[i].allocated && !memoryBlocks[i + 1].allocated)
        {
            memoryBlocks[i].size += memoryBlocks[i + 1].size; // Increase the size of the current block.
            memoryBlocks.erase(memoryBlocks.begin() + i + 1); // Remove the next block.
            --i; // Decrement i since we removed an element from the vector.
        }
    }
}

```

- 内存状态报告：实现内存状态报告功能，以便用户了解当前内存的分配情况。
打印内存状态：遍历内存块列表，输出已分配和未分配内存块的地址范围和状态。

```

void printMemoryStatus()
{
    for (const MemoryBlock &block : memoryBlocks)
    {
        std::cout << "[" << block.start << " : " << block.start + block.size - 1 << "]" << " "
            << (block.allocated ? "Allocated" : "Free") << std::endl;
    }
}

```

通过以上设计思路，我们能够构建一个能够接受不同内存分配请求，并根据用户选择的策略进行分配、释放、压缩和报告的内存分配管理器。这样的设计使得我们的程序能够灵活地应对不同的内存管理需求，提高内存的利用率和系统性能。在具体的代码实现中，我们将以上设计思路转化为相应的函数和逻辑，以实现一个完整可用的内存分配管理系统。

b) 代码实现

请参考代码解释的截图与仓库

III. 实验结果与分析

a) 实验环境

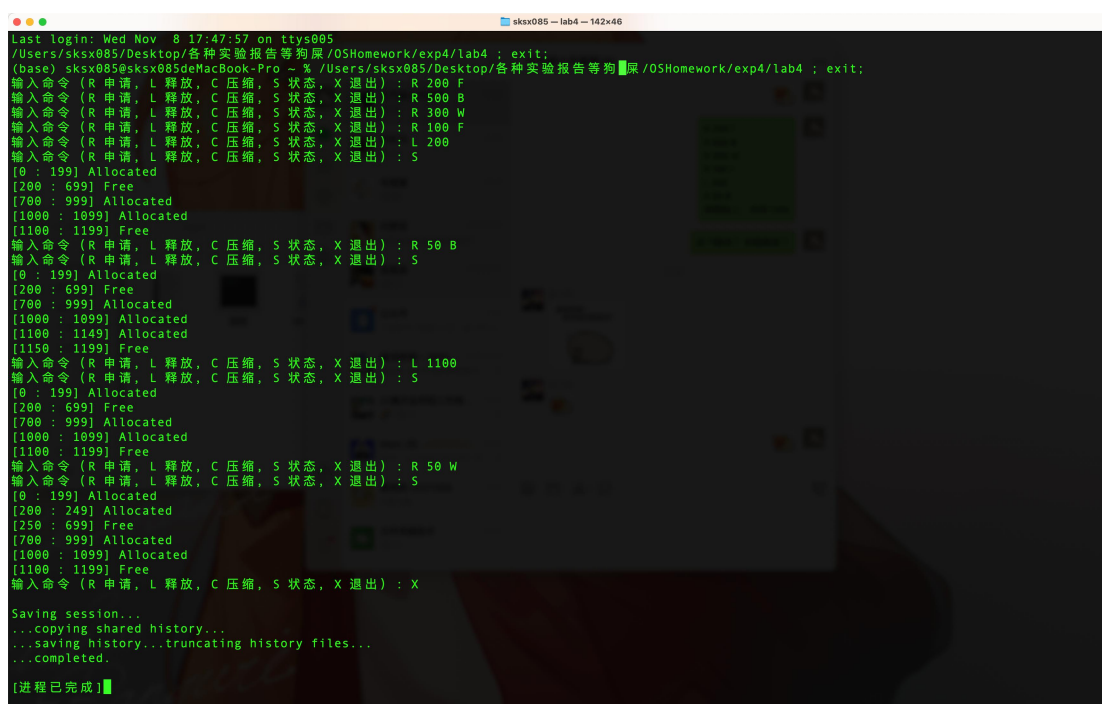
在进行本次实验时，我们需要一个稳定的编程环境来开发、编译和运行 C++ 代码。以下是适合本实验的基本环境要求：

1. 操作系统：推荐使用 Linux 操作系统，例如 Ubuntu 18.04/16.04。Linux 系统对于 C++ 开发提供了良好的支持，并且具有稳定性和安全性。
2. 编程语言：使用 C++ 编程语言进行实验。C++ 是一种功能强大、高效的编程语言，特别适合系统编程和算法实现。实验中的代码示例已经使用了 C++ 语言。
3. 开发工具：推荐使用 GCC (GNU Compiler Collection) 作为 C++ 的编译器。GCC 是一个开源的编译器集合，提供了丰富的编译器支持，可以在 Linux 系统上进行编译。
4. 集成开发环境 (IDE)：在 Linux 系统上，可以选择使用文本编辑器（如 Vim、Emacs）

进行代码编辑，也可以选择使用集成开发环境（如 Code::Blocks、Eclipse CDT 等）来提高开发效率。

5. 版本控制工具： 推荐使用 Git 进行版本控制。Git 是一个分布式版本控制系统，可以帮助团队协作开发，跟踪代码变化，确保代码的版本管理。
6. 文档编辑工具： 使用文本编辑器或者 Markdown 编辑器，撰写实验报告和代码注释。Markdown 是一种轻量级标记语言，适合用于编写文档。

b) 实验结果



```
Last login: Wed Nov  8 17:47:57 on ttys005
/Users/sksx085/Desktop/各种实验报告等狗屎/05Homework/exp4/lab4 ; exit;
(base) sksx085@sksx085deMacBook-Pro ~ % /Users/sksx085/Desktop/各种实验报告等狗屎/05Homework/exp4/lab4 ; exit;
输入命令 (R 申请, L 释放, C 压缩, S 状态, X 退出) : R 200 F
输入命令 (R 申请, L 释放, C 压缩, S 状态, X 退出) : R 500 B
输入命令 (R 申请, L 释放, C 压缩, S 状态, X 退出) : R 300 W
输入命令 (R 申请, L 释放, C 压缩, S 状态, X 退出) : R 100 F
输入命令 (R 申请, L 释放, C 压缩, S 状态, X 退出) : L 200
输入命令 (R 申请, L 释放, C 压缩, S 状态, X 退出) : S
[0 : 199] Allocated
[200 : 699] Free
[700 : 999] Allocated
[1000 : 1099] Allocated
[1100 : 1199] Free
输入命令 (R 申请, L 释放, C 压缩, S 状态, X 退出) : R 50 B
输入命令 (R 申请, L 释放, C 压缩, S 状态, X 退出) : S
[0 : 199] Allocated
[200 : 699] Free
[700 : 999] Allocated
[1000 : 1099] Allocated
[1100 : 1149] Allocated
[1150 : 1199] Free
输入命令 (R 申请, L 释放, C 压缩, S 状态, X 退出) : L 1100
输入命令 (R 申请, L 释放, C 压缩, S 状态, X 退出) : S
[0 : 199] Allocated
[200 : 699] Free
[700 : 999] Allocated
[1000 : 1099] Allocated
[1100 : 1199] Free
输入命令 (R 申请, L 释放, C 压缩, S 状态, X 退出) : R 50 W
输入命令 (R 申请, L 释放, C 压缩, S 状态, X 退出) : S
[0 : 199] Allocated
[200 : 249] Allocated
[250 : 699] Free
[700 : 999] Allocated
[1000 : 1099] Allocated
[1100 : 1199] Free
输入命令 (R 申请, L 释放, C 压缩, S 状态, X 退出) : X

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[进程已完成]
```

c) 实验分析

输入：

R 200 F

R 500 B

R 300 W

R 100 F

L 200

R 50 B

样例如上，内存 1200

输出：

输入命令：R 200 F

输出：[0 : 199] Allocated

首次适应策略在内存开始处分配 200 KB

输入命令：R 500 B

输出：[200 : 699] Allocated

紧接着，最佳适应策略分配 500 KB，因为目前只有一个大块空闲内存，所以它会紧跟在已分配的内存块后面。

输入命令：R 300 W

输出: [700 : 999] Allocated

最差适应策略此时会在剩余的最大块（这时也只有一个大块）分配 300 KB。

输入命令: R 100 F

输出: [1000 : 1099] Allocated

首次适应策略再次分配 100 KB, 它会找到第一个足够大的空闲块, 这时是从 1000 KB 到 1023 KB 的部分

输入命令: L 200

输出: [200 : 699] Free

释放从 200 KB 开始的内存块, 这个块是之前分配了 500 KB 的内存块

输入命令: R 50 B

输出: [200 : 249] Allocated

最佳适应策略将会在最小的足够大的空闲块分配 50 KB, 由于 200-699 KB 的内存块刚刚被释放, 它是当前最适合 50 KB 需求的块

IV. 内存管理方法总结

在本次实验中, 我们实现了连续内存分配管理器, 并探讨了三种主要的内存分配策略: 首次适应算法 (First Fit)、最佳适应算法 (Best Fit) 和最坏适应算法 (Worst Fit)。以下是对这些内存管理方法的总结:

1. 首次适应算法 (First Fit) :

- 优点: 简单直接, 能够快速找到第一个满足需求的内存块, 减少了搜索时间。
- 缺点: 可能会导致内存碎片问题, 因为它倾向于在内存的前部分分配内存, 后续较大的内存块可能无法被利用。

2. 最佳适应算法 (Best Fit) :

- 优点: 选择最小的合适内存块, 可以最大程度减小内存碎片问题, 提高内存利用率。
- 缺点: 搜索整个内存块列表以找到最小的合适内存块可能会导致较高的时间复杂度。

3. 最坏适应算法 (Worst Fit) :

- 优点: 分配最大的合适内存块, 减少了内存碎片问题, 提高了大内存块的利用率。
- 缺点: 与 Best Fit 算法相反, 可能会导致内存的碎片化。

对比与选择:

- 内存碎片问题: 首次适应算法和最坏适应算法容易导致内存碎片, 而最佳适应算法在某些情况下能更好地利用内存。
- 时间复杂度: 首次适应算法的搜索速度较快, 而最佳适应算法的搜索速度相对较慢。
- 实际应用: 不同的策略适用于不同的场景。如果应用程序具有明确的内存需求, 首次适应算法可能足够。如果需要最大限度地减小内存碎片, 可以考虑使用最佳适应算法。

在实际应用中, 选择合适的内存分配策略需要综合考虑应用程序的特性、内存需求和性能要求。不同的策略适用于不同的场景, 开发者需要根据具体需求做出权衡和选择, 以实现更高效、稳定的内存管理。

V. 结论

通过本次实验，我们深入学习了连续内存分配算法的原理和实现。通过设计和实现内存分配管理器，我们掌握了不同内存分配策略的实际应用，包括首次适应算法（First Fit）、最佳适应算法（Best Fit）和最坏适应算法（Worst Fit）。

在实验中，我们成功实现了内存分配、释放、压缩和状态报告等功能。我们了解到每种内存分配策略在不同情境下的表现，以及它们的优势和限制。首次适应算法简单快速，但可能导致内存碎片；最佳适应算法能最小化内存碎片，但搜索速度较慢；最坏适应算法能够减少大内存块的碎片，但同样可能导致碎片化。

在实际应用中，选择合适的内存分配策略至关重要。不同的应用场景需要根据内存需求、性能要求和内存碎片化情况选择合适的算法。综合考虑内存利用率和搜索效率，选择最合适的策略，可以优化系统性能，提高内存资源的利用效率。

通过本次实验，我们不仅加深了对内存管理原理的理解，也提高了问题解决和编程能力。我们掌握了如何设计、实现和分析不同内存分配策略，为今后的系统设计和开发提供了宝贵的经验和知识。

在实验的过程中，我们遇到了一些挑战，但通过分析问题、探索解决方案，我们成功地克服了这些困难，得以完成了实验任务。这次实验为我们提供了一个深入了解内存管理的机会，让我们更加熟悉了操作系统底层的内存分配机制。

综上所述，本次实验不仅提供了宝贵的学习经验，也为我们今后的编程和系统设计提供了扎实的基础。

VI. 代码与仓库

参考: <https://github.com/sksx085/OSHomework>