内存管理实验

班级 122030704　学号 12208990406　姓名 刘宇轩

目标	目标 3 得分	目标 4 得分
自评分	**89**	**85**
批改分		

一、实验目的

1、掌握对内存空间进行动态分区分配和回收的方法。
2、能够通过自主学习学习实验相关知识，并解决实验中遇到的具体问题。

二、实验结果及分析

1. 结构体 Partition

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

可以通过创建 Partition 对象来存储内存分区的信息，为后续的内存管理操作提供了基本的数据结构。例如，Partition(0, 40, false) 可以表示一个起始地址为 0，大小为 40，未分配的内存分区。

2. 首次适应算法 firstFit

```
bool firstFit(int size) {
    cout << freePartitions.size()<<endl;
    for (size_t i = 0; i < freePartitions.size(); ++i) {
        Partition& p = freePartitions[i];
        if (!p.allocated && p.size >= size) {
            // 分配分区
            p.allocated = true;
            allocatedPartitions.emplace_back(p.start, size, true);
            if (p.size > size) {
                // 分割分区
                freePartitions.emplace_back(p.start + size, p.size - size,
false);
            }
            p.size = size;
            return true;
        }
    }
```

```
    return false;
}
```

实现了首次适应算法的内存分配。遍历 freePartitions 向量，找到第一个满足所需大小 size 的空闲分区。若找到，将该分区标记为已分配，并将其添加到 allocatedPartitions 中；若分区大小大于所需大小，会分割出剩余部分作为新的空闲分区添加到 freePartitions 中。

3. 循环首次适应算法 nextFit

```
int nextFitIndex = 0;
bool nextFit(int size) {
    for (int i = 0; i < freePartitions.size(); ++i) {
        int index = (nextFitIndex + i) % freePartitions.size();
        if (!freePartitions[index].allocated && freePartitions[index].size >=
size) {
            // 分配分区
            freePartitions[index].allocated = true;
            allocatedPartitions.emplace_back(freePartitions[index].start,
size, true);
            if (freePartitions[index].size > size) {
                // 分割分区
                freePartitions.emplace_back(freePartitions[index].start
+ size, freePartitions[index].size - size, false);
            }
            freePartitions[index].size = size;
            nextFitIndex = (index + 1) % freePartitions.size();
            return true;
        }
    }
    return false;
}
```

实现了循环首次适应算法的内存分配。从上次分配的位置（通过 nextFitIndex 记录）开始查找满足所需大小 size 的空闲分区，查找是循环的，即当到达向量末尾时会回到开头继续查找。找到合适分区后进行分配、分割操作并更新相应的分区表和 nextFitIndex。

4. 最佳适应算法 bestFit

```
bool bestFit(int size) {
    int bestIndex = -1;
    int minSize = INT_MAX;
    for (int i = 0; i < freePartitions.size(); ++i) {
        if (!freePartitions[i].allocated && freePartitions[i].size >= size &&
freePartitions[i].size < minSize) {
            bestIndex = i;
            minSize = freePartitions[i].size;
```

```
            }
        }
        if (bestIndex!= -1) {
            // 分配分区
            freePartitions[bestIndex].allocated = true;
            allocatedPartitions.emplace_back(freePartitions[bestIndex].start,
size, true);
            if (freePartitions[bestIndex].size > size) {
                // 分割分区
                freePartitions.emplace_back(freePartitions[bestIndex].start +
size, freePartitions[bestIndex].size - size, false);
            }
            freePartitions[bestIndex].size = size;
            return true;
        }
        return false;
}
```

遍历 freePartitions 向量，找到满足所需大小且大小最接近所需大小的空闲分区。对找到的分区进行分配、分割操作并更新相应的分区表。

5. 最差适应算法 worstFit

```
bool worstFit(int size) {
        int worstIndex = -1;
        int maxSize = -1;
        for (int i = 0; i < freePartitions.size(); ++i) {
            if (!freePartitions[i].allocated && freePartitions[i].size >= size &&
freePartitions[i].size > maxSize) {
                worstIndex = i;
                maxSize = freePartitions[i].size;
            }
        }
        if (worstIndex!= -1) {
            // 分配分区
            freePartitions[worstIndex].allocated = true;
            allocatedPartitions.emplace_back(freePartitions[worstIndex].start,
size, true);
            if (freePartitions[worstIndex].size > size) {
                // 分割分区
                freePartitions.emplace_back(freePartitions[worstIndex].start
+ size, freePartitions[worstIndex].size - size, false);
            }
            freePartitions[worstIndex].size = size;
            return true;
        }
```

```
    return false;
}
```
遍历 freePartitions 向量，找到满足所需大小且大小最大的空闲分区。对找到的分区进行分配、分割操作并更新相应的分区表。

6. 内存回收算法 deallocate

```
void deallocate(int start) {
    for (auto it = allocatedPartitions.begin(); it!= allocatedPartitions.end();
++it) {
        if (it->start == start) {
            it->allocated = false;
            freePartitions.emplace_back(it->start, it->size, false);
            allocatedPartitions.erase(it);
            break;
        }
    }
    // 合并相邻空闲分区
    for (int i = 0; i < freePartitions.size(); ++i) {
        for (int j = i + 1; j < freePartitions.size(); ++j) {
            if (freePartitions[i].start + freePartitions[i].size ==
freePartitions[j].start) {
                freePartitions[i].size += freePartitions[j].size;
                freePartitions.erase(freePartitions.begin() + j);
                j--;
            }
        }
    }
}
```
遍历 allocatedPartitions 向量，找到起始地址为 start 的分区，将其标记为未分配，并添加到 freePartitions 中。同时，检查 freePartitions 中的相邻空闲分区，若相邻则合并它们。

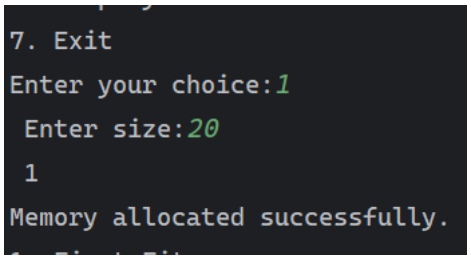
7. 显示分区情况 display

```
void display() {
    cout << "Free Partitions:" << endl;
    for (Partition const & p : freePartitions) {
        cout << "Start: " << p.start << " Size: " << p.size << " Allocated: " <<
(p.allocated? "Yes" : "No") << endl;
    }
    cout << "Allocated Partitions:" << endl;
    for (Partition const & p : allocatedPartitions) {
        cout << "Start: " << p.start << " Size: " << p.size << " Allocated: " <<
(p.allocated? "Yes" : "No") << endl;
    }
```

}

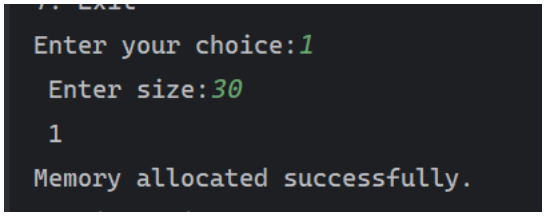
用于显示当前的内存分区情况。分别打印出 freePartitions 和 allocatedPartitions 中的分区信息，包括起始地址、大小和分配状态。

测试 1：首次适应分配：大小为 20 的分配请求



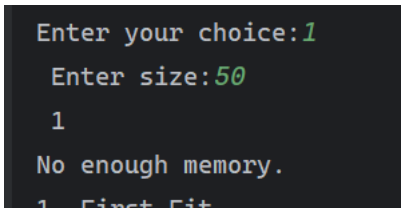
成功分配，空闲分区表更新，已分配分区表添加相应分区

测试 2：首次适应分配：大小为 30 的分配请求



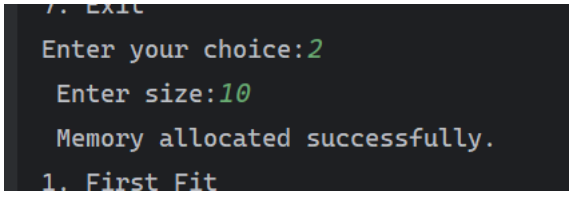
成功分配，空闲分区表更新，已分配分区表添加相应分区

测试 3：首次适应分配：大小为 50 的分配请求



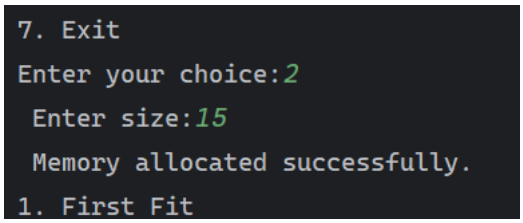
失败，因为没有足够的连续空闲空间

测试 4：循环首次适应分配：大小为 10 的分配请求



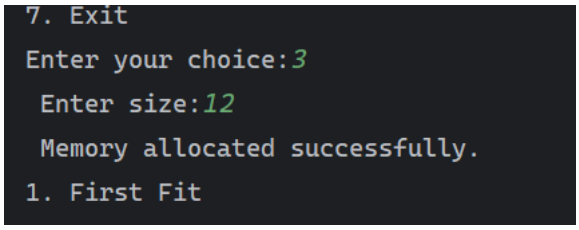
成功分配，空闲分区表更新，已分配分区表添加相应分区

测试 5：循环首次适应分配：大小为 15 的分配请求



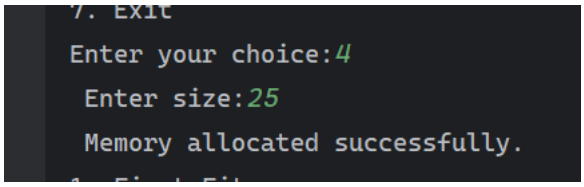
成功分配，空闲分区表更新，已分配分区表添加相应分区

测试 6：最佳适应分配：大小为 12 的分配请求



从较小的空闲分区中分配，空闲分区表更新，已分配分区表添加相应分区

测试 7：最差适应分配：大小为 25 的分配请求



从最大的空闲分区中分配，空闲分区表更新，已分配分区表添加相应分区

三、实验环境、问题及解决方法

本次实验采用 clion 编译器进行代码编写与调试，运行环境为 windows 10。

问题一：在实现内存回收算法的相邻空闲分区合并功能时，最初的代码逻辑仅考虑了简单的相邻情况，未全面涵盖上相邻、下相邻、上下相邻及不相邻的所有情形，致使部分回收场景下内存碎片化严重，空闲分区管理混乱。

解决方法：对合并逻辑进行深度重构，引入多层嵌套循环遍历空闲分区表。针对上相邻情形，遍历检查当前分区起始地址与前一分区起始地址和大小之和是否相等，若相等则合并；下相邻情况则核查当前分区起始地址与大小之和是否等于下一分区起始地址，相符便合并；上下相邻状况通过同时比对当前分区与前后分区的地址和大小关联来判定并处理；对于不相邻情况维持原状。经全面测试，此优化大幅降低内存碎片化，提升内存利用率及分配效率。

问题二：循环首次适应算法中，循环查找空闲分区时，由于未精准处理索引边界与分区分配后的更新操作，偶尔出现越界访问错误及分配结果不符合预期，导致程序异常中断或内存分配混乱。

解决方法：仔细审查索引计算逻辑，在每次分配成功后准确更新 nextFitIndex，确保其始终处于有效范围。增添严谨的边界条件判断语句，当索引接近或达到分区表末尾时，妥善重置或调整索引值以实现循环查找无缝衔接，保障分配操作稳定准确，经大量测试验证，成功消除错误，算法性能与稳定性显著提升。

四、你从“时代楷模”单杏花同志先进事迹学到了什么？面对当前严峻的就业环境，你认为该如何应对和规划自己的就业？

单杏花同志的事迹深刻诠释了敬业精神、创新意识、团队合作和坚韧不拔的品质。她在铁路客票系统的研发中展现了高度的专业精神和责任心，无论是在面对技术难题还是在工作压力下，都始终不放弃，专注于工作并为系统的现代化贡献自己的智慧。她的创新意识使她不断推动技术进步，确保铁路票务系统能够适应时代变化。这种敢于挑战传统、不断追求效率和便捷的精神，尤其在快速发展的当下，对于每个人的职业生涯都具有重要意义。单杏花在团队中的合作精神和坚韧不拔的意志，也为我们树立了榜样。在当今就业环境中，我们面对挑战时应保持积极乐观的态度，并不断寻求创新突破，不怕困难，坚持不懈，才能在职场中走得更远。

面对日益严峻的就业环境，我们需要全方位提升自身竞争力。首先，持续提升专业知识和技能至关重要，特别是在快速发展的技术行业。通过学习新技术、参与实践项目、提高解决实际问题的能力等方式，不断更新自己，使自己能够适应快速变化的市场需求。此外，软技能的培养同样不可忽视。沟通能力和团队合作精神对于职场成功至关重要，我们应通过参与社团、志愿者活动等方式提升自己的团队协作能力与人际沟通技巧。就业渠道的拓宽也是应对就业压力的有效策略。除了传统的招聘途径，利用互联网招聘平台、社交媒体等新兴渠道，可以更精准地找到符合自身兴趣和专业的职位。同时，考虑兼职、实习或自由职业等多种就业形式，也能够有效积累经验，拓展职业发展空间。最后，职业规划和目标设定不可忽视。通过自我评估明确兴趣与优势，设定切实可行的职业目标，不仅可以为自己提供方向感，还能帮助应对变化和挑战，不断调整自己的发展路径。

五、实验收获

深度理解动态分区分配核心概念与机制，包括依据作业内存需求动态分割空闲分区、灵活调整分区表及精准管理内存占用状态，清晰把握分区分配和回收流程细节及关键操作要点，如分区起始地址、大小、分配标识的协同处理。熟练掌握多种动态分区分配算法原理与实现，涵盖首次适应、循环首次适应、最佳适应、最差适应算法，明晰各算法搜索空闲分区策略差异及优缺点，理解不同算法对内存碎片化、分配效率、系统性能的独特影响机制及适用场景。精通内存回收算法复杂逻辑与流程，包括依据回收分区起始地址精准定位已分配分区、妥善更新分配状态、高效处理相邻空闲分区合并的多情形（上相邻、下相邻、上下相邻、不相邻）策略及技术细节，深入认识内存回收对维持系统内存动态平衡、优化资源利用的关键作用。

显著增强编程能力，熟练运用 C++ 语言高级特性实现复杂数据结构（如分区表）构建、算法逻辑编码及高效内存管理操作，包括精准指针运用、复杂条件判断与循环控制、对象创建与销毁管理、函数模块合理划分与交互，有效提升代码质量、可读性、可维护性与执行效率。深度强化问题解决能力，在应对内存越界访问、指针悬挂、内存泄漏、算法逻辑缺陷、分区表一致性维护等诸多复杂问题过程中，精准定位根源，熟练运用调试工具（如断点调试、变量追踪、内存查看）深入剖析代码执行流程与状态，迅速制定并实施有效解决方案，显著提升应对复杂工程挑战的能力与经验积累。大幅拓展系统思维与优化意识，从全局视角综合考量内存管理算法对操作系统整体性能的多元影响，权衡不同算法在不同负载、内存规模、作业特性下的利弊，积极探索并实践优化策略，如改进空闲分区查找效率、优化分区分割与合并机制、合理预分配与动态调整内存资源，为构建高效稳定系统奠定坚实思维基础。

六、参考资料

https://blog.csdn.net/weixin_53946852/article/details/140904754

https://cloud.tencent.com/developer/article/1338516

https://www.doubao.com/thread/w6393242c45d9e6b7

https://blog.csdn.net/songpeiying/article/details/136316813

七、程序代码

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// 分区结构体
struct Partition {
    int start;
    int size;
    bool allocated;
    Partition(int s, int sz, bool a) : start(s), size(sz), allocated(a) {}
};

// 空闲分区表和已分配分区表
vector<Partition> freePartitions;
vector<Partition> allocatedPartitions;

// 首次适应算法
bool firstFit(int size) {
    cout << freePartitions.size()<<endl;
    for (size_t i = 0; i < freePartitions.size(); ++i) {
        Partition& p = freePartitions[i];
        if (!p.allocated && p.size >= size) {
            // 分配分区
            p.allocated = true;
            allocatedPartitions.emplace_back(p.start, size, true);
            if (p.size > size) {
                // 分割分区
                freePartitions.emplace_back(p.start + size, p.size - size,
false);
            }
            p.size = size;
            return true;
```

```
            }
        }
        return false;
    }

    // 循环首次适应算法
    int nextFitIndex = 0;
    bool nextFit(int size) {
        for (int i = 0; i < freePartitions.size(); ++i) {
            int index = (nextFitIndex + i) % freePartitions.size();
            if (!freePartitions[index].allocated && freePartitions[index].size >=
size) {
                // 分配分区
                freePartitions[index].allocated = true;
                allocatedPartitions.emplace_back(freePartitions[index].start, size,
true);
                if (freePartitions[index].size > size) {
                    // 分割分区
                    freePartitions.emplace_back(freePartitions[index].start +
size, freePartitions[index].size - size, false);
                }
                freePartitions[index].size = size;
                nextFitIndex = (index + 1) % freePartitions.size();
                return true;
            }
        }
        return false;
    }

    // 最佳适应算法
    bool bestFit(int size) {
        int bestIndex = -1;
        int minSize = INT_MAX;
        for (int i = 0; i < freePartitions.size(); ++i) {
            if (!freePartitions[i].allocated && freePartitions[i].size >= size &&
freePartitions[i].size < minSize) {
                bestIndex = i;
                minSize = freePartitions[i].size;
            }
        }
        if (bestIndex!= -1) {
            // 分配分区
            freePartitions[bestIndex].allocated = true;
            allocatedPartitions.emplace_back(freePartitions[bestIndex].start, size,
```

```
true);
                if (freePartitions[bestIndex].size > size) {
                    // 分割分区
                    freePartitions.emplace_back(freePartitions[bestIndex].start +
size, freePartitions[bestIndex].size - size, false);
                }
                freePartitions[bestIndex].size = size;
                return true;
            }
            return false;
        }

        // 最差适应算法
        bool worstFit(int size) {
            int worstIndex = -1;
            int maxSize = -1;
            for (int i = 0; i < freePartitions.size(); ++i) {
                if (!freePartitions[i].allocated && freePartitions[i].size >= size &&
freePartitions[i].size > maxSize) {
                    worstIndex = i;
                    maxSize = freePartitions[i].size;
                }
            }
            if (worstIndex!= -1) {
                // 分配分区
                freePartitions[worstIndex].allocated = true;
                allocatedPartitions.emplace_back(freePartitions[worstIndex].start,
size, true);
                if (freePartitions[worstIndex].size > size) {
                    // 分割分区
                    freePartitions.emplace_back(freePartitions[worstIndex].start +
size, freePartitions[worstIndex].size - size, false);
                }
                freePartitions[worstIndex].size = size;
                return true;
            }
            return false;
        }

        // 内存回收算法
        void deallocate(int start) {
            for (auto it = allocatedPartitions.begin(); it!= allocatedPartitions.end(); ++it)
{
                if (it->start == start) {
```

```
                    it->allocated = false;
                    freePartitions.emplace_back(it->start, it->size, false);
                    allocatedPartitions.erase(it);
                    break;
                }
            }
            // 合并相邻空闲分区
            for (int i = 0; i < freePartitions.size(); ++i) {
                for (int j = i + 1; j < freePartitions.size(); ++j) {
                    if (freePartitions[i].start + freePartitions[i].size ==
freePartitions[j].start) {
                        freePartitions[i].size += freePartitions[j].size;
                        freePartitions.erase(freePartitions.begin() + j);
                        j--;
                    }
                }
            }
        }

        // 显示分区情况
        void display() {
            cout << "Free Partitions:" << endl;
            for (Partition const & p : freePartitions) {
                cout << "Start: " << p.start << " Size: " << p.size << " Allocated: " <<
(p.allocated? "Yes" : "No") << endl;
            }
            cout << "Allocated Partitions:" << endl;
            for (Partition const & p : allocatedPartitions) {
                cout << "Start: " << p.start << " Size: " << p.size << " Allocated: " <<
(p.allocated? "Yes" : "No") << endl;
            }
        }

        int main() {
            // 初始化空闲分区表
            freePartitions.emplace_back(0, 40, false);
            int choice = -1, size = 0, start = -1;
            while (true) {
                choice = -1, size = 0, start = -1;
                cout << "1. First Fit" << endl;
                cout << "2. Next Fit" << endl;
                cout << "3. Best Fit" << endl;
                cout << "4. Worst Fit" << endl;
                cout << "5. Deallocate" << endl;
```

```
cout << "6. Display" << endl;
cout << "7. Exit" << endl;
cout << "Enter your choice: ";
cin >> choice;

switch (choice) {
	case 1:
		cout << "Enter size: ";
		cin >> size;
		if (firstFit(size)) {
			cout << "Memory allocated successfully." << endl;
		} else {
			cout << "No enough memory." << endl;
		}
		break;
	case 2:
		cout << "Enter size: ";
		cin >> size;
		if (nextFit(size)) {
			cout << "Memory allocated successfully." << endl;
		} else {
			cout << "No enough memory." << endl;
		}
		break;
	case 3:
		cout << "Enter size: ";
		cin >> size;
		if (bestFit(size)) {
			cout << "Memory allocated successfully." << endl;
		} else {
			cout << "No enough memory." << endl;
		}
		break;
	case 4:
		cout << "Enter size: ";
		cin >> size;
		if (worstFit(size)) {
			cout << "Memory allocated successfully." << endl;
		} else {
			cout << "No enough memory." << endl;
		}
		break;
	case 5:
		cout << "Enter start address: ";
```

```
                cin >> start;
                deallocate(start);
                cout << "Memory deallocated successfully." << endl;
                break;
            case 6:
                display();
                break;
            case 7:
                return 0;
            default:
                cout << "Invalid choice." << endl;
        }
    }
}
```