安徽大学人工智能学院《操作系统》实验报告

学号___WA2214014___ 姓名___杨跃浙___ 年级___大三___

【实验名称】_____连续分配与分区管理_____

【实验内容】

设计并实现一个简单的内存管理算法，支持首次适应算法内存分配策略：

设计合适的内存块结构（内存分区管理），并根据输入的进程请求和内存状态进行分配和回收。

模拟内存中分区的管理，输出每次内存分配的结果，包括内存分配情况和分区使用情况。

在程序中模拟进程的创建和销毁，根据用户输入的请求和释放内存的操作进行管理。

【实验要求】

设计并实现一个简单的内存管理算法，能够模拟连续内存分配。

支持动态内存分配和释放，模拟进程请求内存和释放内存的过程。

使用首次适应算法（First Fit）进行内存分配。

对每次分配后的内存使用情况进行打印输出，展示内存分配后的状态。

允许用户输入内存请求和释放的操作，模拟内存分配的过程。

最后统计每种内存分配算法的分配效率，并比较其性能。

测试样例

测试样例 1：

输入：

空闲内存块大小：[100，500，200，300，600]

请求的进程大小：[212，417，112，426]

输出：

进程	请求大小	分配内存块
1	212	500
2	417	600
3	112	200
4	426	无法分配

测试样例 2：

输入：

空闲内存块大小：[400，600，200，800]

请求的进程大小：[300，500，600，250，700]

输出：

进程	请求大小	分配内存块
1	300	400
2	500	600
3	600	800
4	250	无法分配
5	700	无法分配

【实验原理】

首次适应算法的工作原理及实现

首次适应算法（First Fit）作为动态分区分配的一种策略，其核心思想是尽可能早地使用内存中的空闲空间，以降低碎片的生成。该算法通过维护一个按物理地址排序的空闲分区链表来实现。当系统接收到一个内存分配请求时，算法从链表的头部开始，顺序搜索第一个足够大的空闲区块。具体操作是比较请求内存大小 M 与每个空闲块 S_i 的大小，直到找到 $S_i \geq M$ 的块。如果找到，该空闲块大小更新为 $S_i - M$；如果 S_i 正好等于 M，该块则从链表中删除。这种方法的主要优点是实现简单，并且在内存块较大时可以快速找到位置，但缺点是随着内存分配的进行，可能导致较大的外部碎片和增加搜索时间

内存分区管理与碎片化的处理

有效的内存分区管理是提高系统整体性能和内存利用率的关键。

在首次适应算法中，处理外部碎片是一个重要的问题。外部碎片是由于小的空闲内存块分散在内存中而无法被有效利用所造成的。

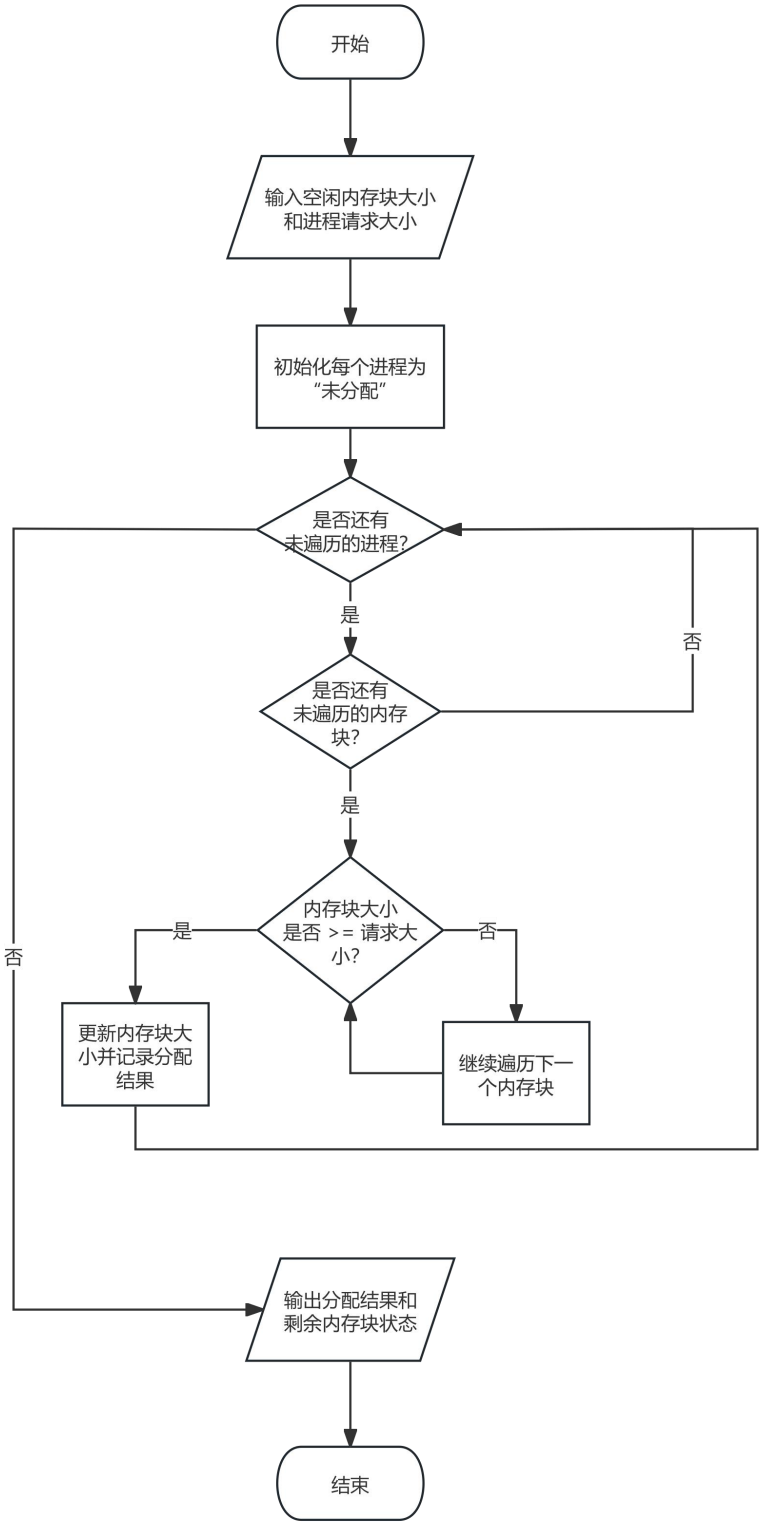
碎片的量可以通过公式 $E = T - \sum_{i=1}^{n} A_i$ 来评估，其中 T 是总内存， A_i 是第 i 个已分配内存块的大小。为了减轻外部碎片的影响，内存管理系统可以定期进行碎片整理，将多个小的空闲块合并成更大的块。此外，当内存被释放时，系统需要检查相邻的空闲区域并进行合并，以此减少碎片的产生和提高内存的使用效率。

动态内存分配的性能评估

评估动态内存分配算法的性能涉及多个方面，包括内存利用率、分配和回收的效率以及系统的响应时间。内存利用率 U 可以通过公式 $U = \frac{\sum_{i=1}^{n} A_i}{T} \times 100\%$ 计算，这反映了分配的内存量占总内存的比例。首次适应算法在实际应用中通常能快速响应小内存请求，但对于大内存请求，可能因为需要遍历较长的空闲链表而效率下降。此外，由于每次都从头开始搜索可能会导致性能形颈，特别是当内存分配和释放频繁时。因此，在实嬐中，通过模拟不同的内存请求情况，可以实际测量和比较首次适应算法与其他算法（如最佳适应和最差适应）的性能差异。

【实验步骤】

程序流程图


开始
输入空闲内存块大小
和进程请求大小
初始化每个进程为
“未分配”
是否还有
未遍历的进程?
是
否
是否还有
未遍历的内存
块?
是
否
内存块大小
是否 >= 请求大
小?
是
否
更新内存块大
小并记录分配
结果
继续遍历下一
个内存块
输出分配结果和
剩余内存块状态
结束

核心代码（含必要的注释）

```
#include <stdio.h>
#define MAX_BLOCKS 100
#define MAX_PROCESSES 100

// 定义内存块结构，用于存储每个内存块的大小
typedef struct {
int size; // 内存块的大小
} MemoryBlock;

// 定义进程结构，用于存储每个进程的请求大小和已分配的内存块索引
typedef struct {
int size; // 进程请求的内存大小
int allocated_block; // 已分配的内存块索引，-1 表示未分配
} Process;

// 实现首次适应算法（First Fit）进行内存分配
void first_fit(MemoryBlock blocks[], int num_blocks, Process
processes[], int num_processes) {
for (int i = 0; i < num_processes; i++) {
processes[i].allocated_block = -1; // 初始化为未分配状态
for (int j = 0; j < num_blocks; j++) {
// 如果内存块的大小满足进程请求，则分配给当前进程
if (blocks[j].size >= processes[i].size) {
processes[i].allocated_block = j; // 记录分配的内存块索引
blocks[j].size -= processes[i].size; // 更新内存块的剩余大小
break; // 分配完成后退出循环
}
}
}
}

// 打印内存分配结果及剩余内存块信息
void print_allocation(MemoryBlock blocks[], int num_blocks,
Process processes[], int num_processes) {
printf("\n 进程\t 请求大小\t 分配内存块\n");
```

```
for (int i = 0; i < num_processes; i++) {
if (processes[i].allocated_block != -1) {
// 如果已分配，输出分配的内存块编号（从 1 开始计数）
printf("%d\t%d\t\t%d\n", i + 1, processes[i].size,
processes[i].allocated_block + 1);
} else {
// 如果未分配，输出“无法分配”
printf("%d\t%d\t\t 无法分配\n", i + 1, processes[i].size);
}
}

// 打印剩余内存块的大小
printf("\n 剩余内存块情况:\n");
for (int i = 0; i < num_blocks; i++) {
printf("内存块 %d: 剩余大小 = %d\n", i + 1, blocks[i].size);
}
}

int main() {
int num_blocks, num_processes;
MemoryBlock blocks[MAX_BLOCKS]; // 定义空闲内存块数组
Process processes[MAX_PROCESSES]; // 定义进程数组

// 输入空闲内存块的数量
printf("输入空闲内存块的数量: ");
scanf("%d", &num_blocks);
printf("输入每个内存块的大小:\n");
for (int i = 0; i < num_blocks; i++) {
printf("内存块 %d: ", i + 1);
scanf("%d", &blocks[i].size); // 输入每个内存块的大小
}

// 输入进程请求的内存大小
printf("输入进程的数量: ");
scanf("%d", &num_processes);
printf("输入每个进程的请求大小:\n");
for (int i = 0; i < num_processes; i++) {
printf("进程 %d: ", i + 1);
scanf("%d", &processes[i].size); // 输入每个进程请求的内存大小
```

```
}

// 执行首次适应算法进行内存分配
first_fit(blocks, num_blocks, processes, num_processes);

// 输出分配结果和剩余内存块情况
print_allocation(blocks, num_blocks, processes,
num_processes);

return 0;
}
```

运行结果

```
输入空闲内存块的数量： 5
输入每个内存块的大小：
内存块  1: 100
内存块  2: 500
内存块  3: 200
内存块  4: 300
内存块  5: 600
输入进程的数量： 4
输入每个进程的请求大小：
进程  1: 212
进程  2: 417
进程  3: 112
进程  4: 426

进程        请求大小            分配内存块
1           212                 2
2           417                 5
3           112                 2
4           426                 无法分配

剩余内存块情况：
内存块  1: 剩余大小 = 100
内存块  2: 剩余大小 = 176
内存块  3: 剩余大小 = 200
内存块  4: 剩余大小 = 300
内存块  5: 剩余大小 = 183
```

```
输入空闲内存块的数量： 4
输入每个内存块的大小：
内存块  1: 400
内存块  2: 600
内存块  3: 200
内存块  4: 800
输入进程的数量： 5
输入每个进程的请求大小：
进程  1: 300
进程  2: 500
进程  3: 600
进程  4: 250
进程  5: 700

进程        请求大小          分配内存块
1           300             1
2           500             2
3           600             4
4           250             无法分配
5           700             无法分配

剩余内存块情况：
内存块  1: 剩余大小 = 100
内存块  2: 剩余大小 = 100
内存块  3: 剩余大小 = 200
内存块  4: 剩余大小 = 200
```

结果分析

测试样例 1：

输入数据：

- **空闲内存块大小**: [100, 500, 200, 300, 600]

- **请求的进程大小**: [212, 417, 112, 426]

计算过程：

1. **进程 1 (请求大小: 212)**:

- 查找空闲内存块，从头开始：
- 第一个内存块大小为 100，不足。
- 第二个内存块大小为 500，足够分配。
- 分配内存块 2，剩余大小 = 500 – 212 = 288。
- 当前内存块状态: [100, 288, 200, 300, 600]

2. **进程 2 (请求大小: 417)**:

- 查找空闲内存块，从头开始：
- 第一个内存块大小为 100，不足。
- 第二个内存块大小为 288，不足。
- 第三个内存块大小为 200，不足。
- 第四个内存块大小为 300，不足。
- 第五个内存块大小为 600，足够分配。
- 分配内存块 5，剩余大小 = 600 – 417 = 183。

- 当前内存块状态: [100, 288, 200, 300, 183]

3. **进程 3 (请求大小: 112)**:

- 查找空闲内存块，从头开始：
- 第一个内存块大小为 100，不足。
- 第二个内存块大小为 288，足够分配。
- 分配内存块 2，剩余大小 = 288 – 112 = 176。
- 当前内存块状态: [100, 176, 200, 300, 183]

4. **进程 4 (请求大小: 426)**:

- 查找空闲内存块，从头开始：
- 第一个内存块大小为 100，不足。
- 第二个内存块大小为 176，不足。
- 第三个内存块大小为 200，不足。
- 第四个内存块大小为 300，不足。
- 第五个内存块大小为 183，不足。
- 无法分配。
- 当前内存块状态: [100, 176, 200, 300, 183]

输出结果：

进程	请求大小	分配内存块
1	212	2
2	417	5
3	112	2
4	426	无法分配

测试样例 2 ：

输入数据：

- **空闲内存块大小**: [400, 600, 200, 800]
- **请求的进程大小**: [300, 500, 600, 250, 700]

计算过程：

1. **进程 1 (请求大小: 300)**:
   - 查找空闲内存块，从头开始：
   - 第一个内存块大小为 400，足够分配。

- 分配内存块 1，剩余大小 = 400 – 300 = 100。
- 当前内存块状态: [100, 600, 200, 800]

2. **进程 2 (请求大小: 500)**:

- 查找空闲内存块，从头开始：
- 第一个内存块大小为 100，不足。
- 第二个内存块大小为 600，足够分配。
- 分配内存块 2，剩余大小 = 600 – 500 = 100。
- 当前内存块状态: [100, 100, 200, 800]

3. **进程 3 (请求大小: 600)**:

- 查找空闲内存块，从头开始：
- 第一个内存块大小为 100，不足。
- 第二个内存块大小为 100，不足。
- 第三个内存块大小为 200，不足。
- 第四个内存块大小为 800，足够分配。
- 分配内存块 4，剩余大小 = 800 – 600 = 200。
- 当前内存块状态: [100, 100, 200, 200]

4. **进程 4 (请求大小: 250)**:

- 查找空闲内存块，从头开始：
- 第一个内存块大小为 100，不足。
- 第二个内存块大小为 100，不足。
- 第三个内存块大小为 200，不足。
- 第四个内存块大小为 200，不足。
- 无法分配。
- 当前内存块状态: [100, 100, 200, 200]

5. **进程 5 (请求大小: 700)**:

- 查找空闲内存块，从头开始：
- 第一个内存块大小为 100，不足。
- 第二个内存块大小为 100，不足。
- 第三个内存块大小为 200，不足。
- 第四个内存块大小为 200，不足。
- 无法分配。
- 当前内存块状态: [100, 100, 200, 200]

输出结果：

进程	请求大小	分配内存块
1	300	1
2	500	2
3	600	4
4	250	无法分配
5	700	无法分配

【实验总结】

1．对首次适应算法的理解

在本次实验中，我通过实现首次适应算法深入理解了其工作机制和实际表现。首次适应算法的核心机制是在内存块链表中顺序搜索，寻找第一个足够大的空闲块来满足进程请求。通过实验测试样例 1 和样例 2，我观察到了算法的具体表现和一些实际问题。

在测试样例 1 中，当进程请求内存块 212, 417, 112，和 426 时，前三个请求成功找到合适的内存块进行分配，但最后一个请求由于没有足够大的空闲内存块，结果显示“无法分配”。这一结果凸显了首次适应算法在处理连续大内存请求后可能导致的碎片化问题，使得较大的请

求难以满足。尤其是当内存块在多次分配后被割裂成多个小块时，这种碎片化显著增加了大块内存请求的失败率。

对于测试样例 2，尽管内存块初始大小较大，但由于连续的请求，内存块被迅速消耗，导致最后两个进程请求（250 和 700 单位）同样因找不到足够的空间而未被满足。这进一步证实了首次适应算法在管理连续大量请求时的局限性，尤其是在面对需求高峰时，碎片化的影响更加明显。

2．实验过程中出现的错误及修改过程

在本次实验中，我最初编写的代码在处理首次适应算法（First Fit）时出现了一个明显的问题：如果找到一个空闲块的大小超过当前进程请求的大小，就会直接将整块标记为 "已分配"，而没有把剩余部分继续保留为空闲。这样做导致了一个结果：当一个大内存块被一个小请求分配后，其余的空间没有得到再利用，后续进程即使需要的内存比剩余空间要小，也会被错误地视为 "无法分配"。这显然与理论知识不符，也与我对首次适应算法的理解背道而驰。

- 问题发现：我在测试时发现，有些大块在被一个小请求分配后就无法再分配给其他需要更少空间的请求，这在样例输出中表现为不合理的无法分配或者错误的分配，而理论上这部分空闲空间本应该还可以满足后续请求。
- 原因分析：在首次适应算法中，如果空闲块大小 BlockSize 严

格大于进程请求大小 RequestSize，程序应该只分配 RequestSize 大小的部分，然后将剩余的 BlockSize – RequestSize 继续保留为新的空闲块。我的初始实现中没有考虑部分分配，导致没有正确地管理剩余空间。

- 实验样例：在进行测试样例时，我最初的记录显示“进程 3 (112) 分配给内存块 3”结果错误地显示在第三个块上，而根据算法步骤和剩余内存的状态，这一分配实际上应当发生在第二块上才符合首次适应算法的原理。究其原因，是我在手动推演或打印输出时，未能准确追踪前一次分配对空闲块大小及顺序造成的影响。修正这一错误后，测试样例结果更加符合理论推断：首次适应算法应当顺序检查各块大小，当检测到第二块仍有足够的剩余空间时，就会在该块上进行分配，而不是跳过它去选择其他块。
- 代码修改：在修改代码的过程中，我意识到，过于依赖一个全局的“已分配”标记，容易导致分配和释放过程出现逻辑混乱。首次适应算法最关键的一点在于“部分分配”：只要当前块的大小足以满足进程请求，就应该将所需部分划分给该进程，而剩余部分仍作为空闲块继续保留。为此，我去除了简单的“是否被分配”标记，转而使用对块大小和位置的动态更新来判断空闲程度。当一个进程请求成功后，我直接调整当前块的大小，并将剩余空间拆分或更新到空闲块列表里。这样，分配过程变得更加灵活，避免了因为一个标记就将整个块视为“已占用”而无法

再次被利用的情况。

错误的结果：

```
输入空闲内存块的数量： 5
输入每个内存块的大小：
内存块  1: 100
内存块  2: 500
内存块  3: 200
内存块  4: 300
内存块  5: 600
输入进程的数量： 4
输入每个进程的请求大小：
进程  1: 212
进程  2: 417
进程  3: 112
进程  4: 426

进程      请求大小          分配内存块
1         212               2(288)
2         417               5(183)
3         112               3(88)
4         426               无法分配

剩余内存块情况：
内存块  1: 剩余大小 = 100, 是否已分配 = 否
内存块  2: 剩余大小 = 288, 是否已分配 = 是
内存块  3: 剩余大小 = 88, 是否已分配 = 是
内存块  4: 剩余大小 = 300, 是否已分配 = 否
内存块  5: 剩余大小 = 183, 是否已分配 = 是
```

经过这两方面的修正，我重新运行了测试样例，结果显示当某块能够部分满足进程需求时，就会正确分配出对应的空间，并把剩余空闲部分继续保留以供后续分配；而对于“3 (112) 分配给第二块”的情形，也能得出理论与实践相符合的结果。通过这次改进，我更加深刻地体会到在实现连续内存分配算法时，准确追踪每块内存的大小和位置胜

过依赖简单的分配标记，并且任何一个不起眼的手动操作失误，都可能在实验结果中引发与理论不符的“错误现象”。

3. 性能优化与算法比较

在本次实验中，除了首次适应算法，我还探索了最佳适应（Best Fit）和下一适应（Next Fit）算法，以比较它们在不同内存请求情景下的性能表现。这项比较不仅加深了我对内存分配策略的理解，还让我更清楚地认识到每种策略的优势和局限，以及它们适用的具体场景。

首次适应算法以其简单直接的搜索方式快速响应内存请求，尤其是在内存区块较大或未经多次分配时效率较高。然而，此算法的一个明显缺点是，随着内存分配次数的增加，未能有效利用的碎片逐渐累积，导致对大块内存请求的响应能力下降。此外，由于首次适应算法总是从内存块的起始位置开始搜索，重复的分配和释放操作会使得内存块前部的小碎片增多，进一步影响整体的内存分配效率。

相比之下，最佳适应算法通过寻找与请求大小最接近的内存块来分配，理论上能够最大程度减少碎片的产生。这种方法对于避免小碎片非常有效，因此在处理多个小型内存请求时表现更佳。但其缺点也非常明显，即寻找最合适内存块的过程可能非常耗时，特别是在内存块列表较长时，效率明显下降。

下一适应算法试图解决首次适应算法中重复搜索起始区域的问题，它从上次分配的位置开始搜索下一个合适的内存块，旨在分散内存使用，

避免内存前端的过度碎片化。这种策略在一定程度上提高了内存分配的均匀性和效率，尤其是在内存请求较为分散时。然而，这也可能导致内存中部出现较大未使用区域，特别是在频繁释放和请求较大内存块的情况下。

通过实验中的模拟请求序列，我能够具体评估每种算法的性能表现。例如，在连续请求多个小型内存块时，最佳适应算法表现优于首次适应和下一适应算法，因为它能更有效地利用剩余的小块内存，减少了碎片。然而，在请求大型内存块的场景中，首次适应和下一适应算法因为较少的搜索过程，响应速度可能更快。这些观察结果不仅帮助我理解了每种算法的工作机制，还指导我在未来面对不同内存管理需求时，如何选择最合适的分配策略。

这次性能比较的经验让我意识到，没有一种内存分配算法能够在所有情况下都是最优的。选择哪一种算法取决于具体应用的内存请求模式和系统对响应时间或内存利用率的需求。