实验 12：动态可重定位分区内存管理模拟设计与实现

计科 1903 班 19281171 王雨潇

1. 实验目的

探索、理解并掌握动态可重定位分区内存管理的设计原理和实现机制。

2. 实验环境

运行环境：操作系统 Windows 10 家庭版

编译器：MinGW-w64 GCC 11.2.0

IDE：VSCode

实验代码见“附录：源程序的完整代码”，所有流程图均使用 plantuml 绘制

3. 实验内容

实验用到的宏定义如下，主要内容为随机生成场景的配置：

```
/* 地址固定分配定义 */
#define physical_memory_size    (512 * 1024)    // 物理内存空间为 512MB，统一单位到 KB
#define system_memory_size      (128 * 1024)    // 操作系统占用低址的 128MB，统一单位到 KB
#define user_memory_size        (384 * 1024)    // 用户可使用高址的 384MB，统一单位到 KB
#define MByte                   1024            // 1MB = 1024KB
#define process_num             50              // 进程总数
#define process_max_size        (200 * MByte)   // 生成进程的最大内存需求

/* 进程的各种状态定义 */
#define unassigned  -1   // 进程未被分配空间
#define suspended    0   // 内存空间不足，进程挂起
#define assigned     1   // 进程已经被分配空间
#define finished     2   // 进程结束运行

/* 其他定义 */
#define author  19281171    // 署个名
#define bool    int         // 假装有 bool 类型
#define true    1
#define false   0
```

3.1. 关键数据结构定义

双向链表是一种数据结构，它的每个结点存在两个指针域，分别存储该结点的前驱结点引用和后继结点引用，从任意一个结点出发，都能通过前驱引用以及后继引用完成整个链表结点的访问。

选用双向链表结构描述内存分区的情况，再用成员包括一个分区指针的结构体描述进程情况，进程和分区链表结点之间为聚合关系：

```
// 分区情况描述，双向链表结构
struct Subregion {
    // 数据部分
    int beginAddress;       // 分区起始地址
    int size;               // 分区大小
    bool isEmpty;           // 分区是否已经被占用

    struct Subregion *prev; // 前项指针，指向相邻的上个分区
    struct Subregion *next; // 后项指针，指向相邻的下个分区
};

// 进程描述
struct Process {
    int id;                     // 进程名
    struct Subregion *address; // 进程的存储地址
    int size;                   // 进程大小
    int state;                  // 进程内存空间的状态
    int startTime;              // 进程的创建时间
    int remainingTime;          // 进程的剩余运行时间
};
```

3.2. 物理内存空间布局初始化

根据实验要求，设定物理内存空间为 512MB，为操作系统划分出占 128MB 的子空间，标记为已被占用，剩余高地址的 384MB 为尚未使用的空闲子空间，再连接两链表结点；

```
// 初始化内存空间，操作系统占用低址的 128MB 不参与内存分配，而高址的 384MB 为用户区可供系统分配
struct Subregion* memorySpaceInit() {
    // 初始化双向链表头结点和尾结点
    struct Subregion* head = (struct Subregion*)malloc(sizeof(struct Subregion));
    struct Subregion* tail = (struct Subregion*)malloc(sizeof(struct Subregion));

    // 初始化操作系统低址，分配为已占用状态
    head->beginAddress = 0;
    head->size = system_memory_size;
```

```
    head->isEmpty = false;
    head->prev = NULL;
    head->next = tail;

    // 初始化用户空间地址，分配为未占用状态
    tail->beginAddress = system_memory_size;
    tail->size = user_memory_size;
    tail->isEmpty = true;
    tail->next = NULL;
    tail->prev = head;

    return head;
}
```

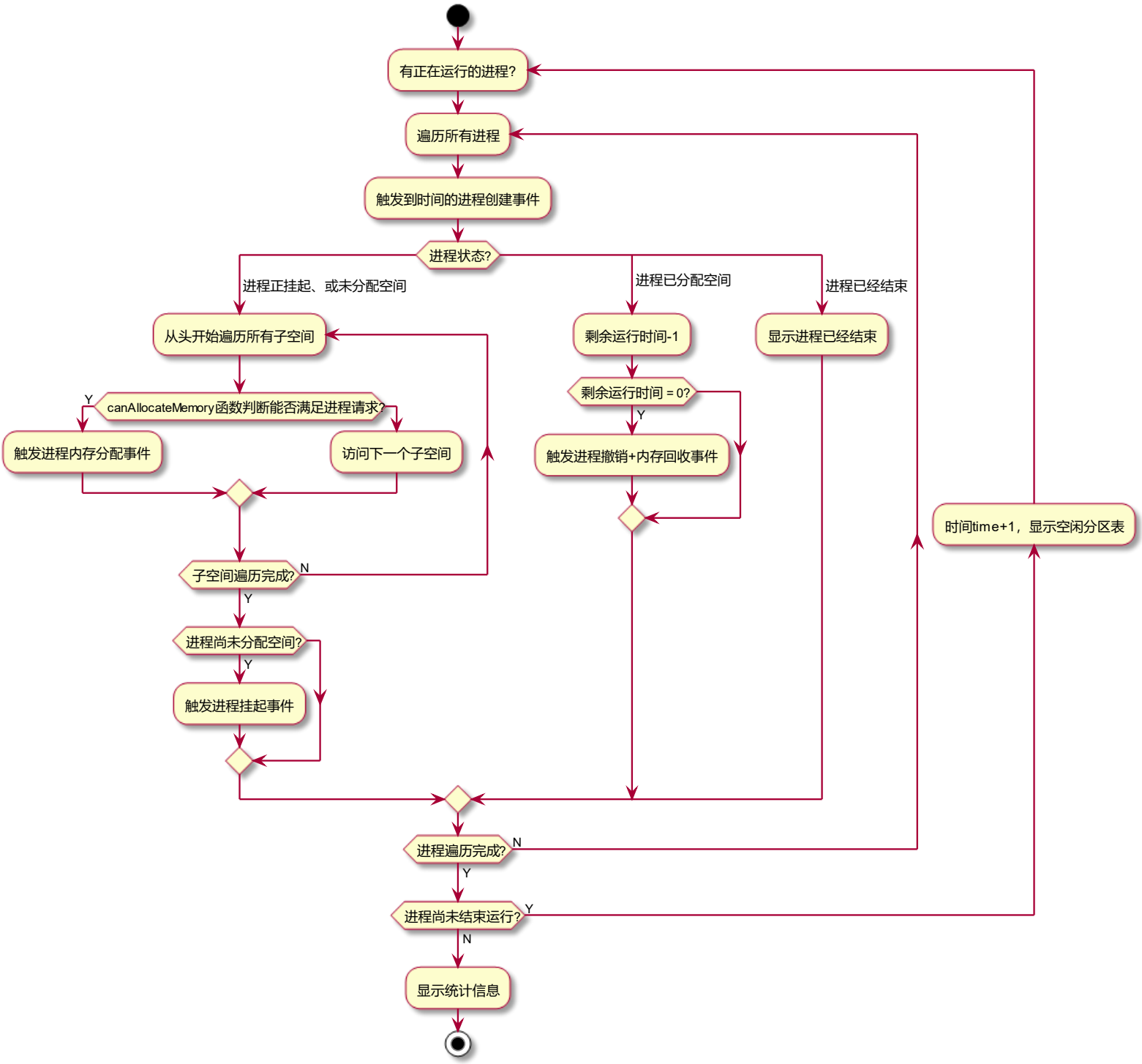
3.3. 构造进程事件随机发生序列

由于 3.1. 关键数据结构定义 用进程名、进程的存储地址、进程大小、进程内存空间的状态、进程的创建时间、进程的剩余运行时间描述一个进程的状态。因此，可以用随机生成一定范围的数字作为进程的开始时间和运行服务时间，再构造多个这样的进程，从而构造进程事件的发生序列；

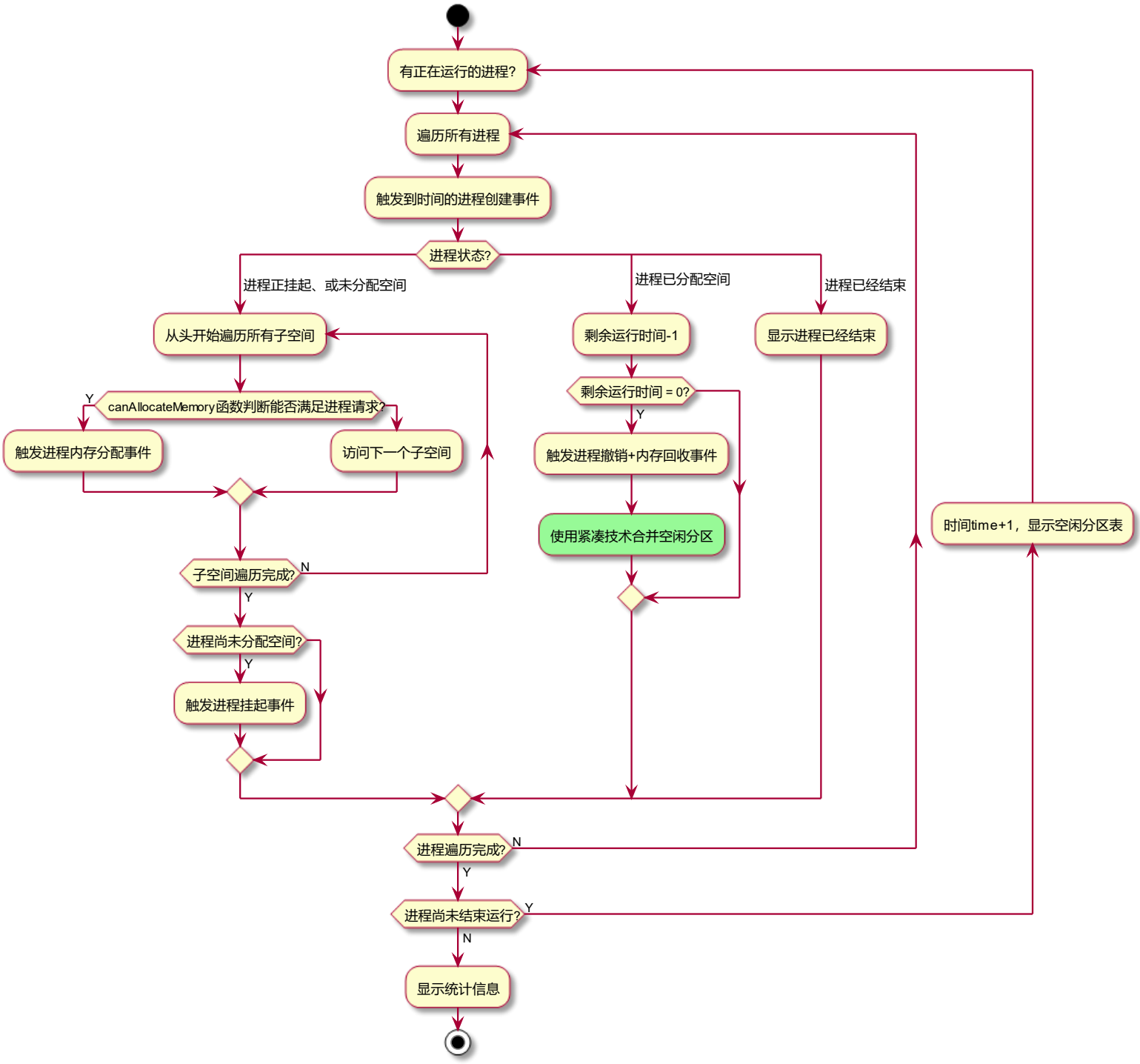
```
    // 随机创建一个进程
    struct Process* p = (struct Process*)malloc(sizeof(struct Process));
    p->id = id;
    p->address = NULL;
    p->size = rand() % (process_max_size + 1);  // 进程所需空间范围 [0,
process_max_size]，单位 KB
    p->startTime = rand() % 25;                 // 进程开始时间范围 [0, 24]
    p->remainingTime = rand() % 21;             // 进程持续时间范围 [0, 20]
    p->state = unassigned;                      // 初始状态为未分配空间
```

3.4. 两种内存分配算法流程

首次适应算法流程图：



动态重定位分区算法流程图：



3.5. 进程内存分配回收流程

由于 3.1. 关键数据结构定义 用双向链表描述子空间，因此内存的分配、回收、紧凑技术分别用对双向链表结点的分裂、合并实现。

分配空间的过程，由子空间大小和进程申请内存大小的关系决定：

（1）子空间<申请内存，则拒绝本次申请；

（2）子空间=申请内存，则把该子空间的状态改为被该进程占用；

（3）子空间>申请内存，将子空间拆分为两个结点，其中一个结点是被进程占用的部分空间，另一个结点是被分剩下的空闲空间，再重新连接链表。

回收空间的过程，由待回收子空间的相邻结点判空情况决定：

（1）左侧是空闲结点，则把该子空间的大小合并到空闲结点上，重新连接链表；

（2）右侧是空闲结点，则把该子空间的大小合并到空闲结点上，重新连接链表；

（3）两侧结点都不空闲，则把该待回收子空间改为空闲状态；

实现紧凑技术的方法：

（1）遍历双向链表中所有子空间，把非空闲的空间连接起来，舍弃中间的空闲结点；

（2）把碎片空闲结点的尺寸相加，得到一个大空闲内存子空间，接在之前的链表后面；

详细实现见 5. 附录：源程序的完整代码

3.6. 算法性能分析

实验方法：在每组场景下，对两种算法分别测试 10 次（测试用例每次均为随机生成）得到 10 组数据，为减少测试用例不同带来的统计差异，对数据进行整理，采用行排序，再取平均值作为第 11 组数据。

性能分析指标：内存空间平均利用率、平均分配查找分区比较次数。

原始数据如下：

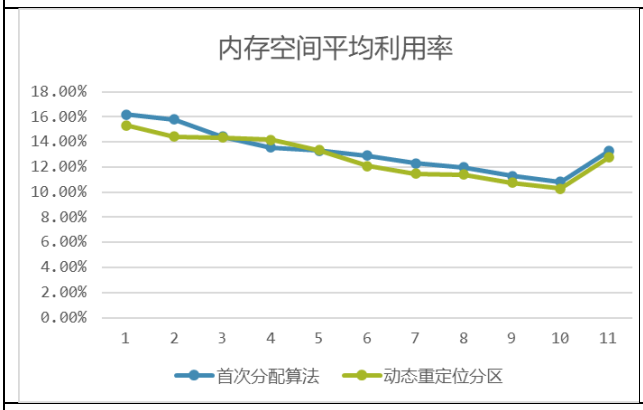
内存空间平均利用率											
首次分配算法	16.20%	15.80%	14.43%	13.55%	13.33%	12.91%	12.32%	11.97%	11.29%	10.82%	13.26%
动态重定位分区	15.35%	14.42%	14.35%	14.20%	13.36%	12.09%	11.47%	11.39%	10.75%	10.28%	12.77%
平均分配查找分区比较次数											
首次分配算法	6	5	5	5	5	5	5	5	5	4	5
动态重定位分区	6	5	5	5	5	5	5	5	4	4	4.9
内存空间平均利用率											
首次分配算法	16.34%	15.84%	14.93%	14.69%	14.24%	12.99%	12.62%	12.56%	12.54%	10.20%	13.69%
动态重定位分区	17.59%	16.14%	15.80%	14.05%	13.86%	13.30%	13.02%	12.68%	11.93%	11.31%	13.97%
平均分配查找分区比较次数											
首次分配算法	6	5	5	5	5	5	5	5	5	4	5
动态重定位分区	6	5	5	5	5	5	5	5	5	4	5
内存空间平均利用率											
首次分配算法	33.27%	32.60%	30.79%	29.39%	26.90%	25.09%	23.31%	21.63%	18.20%	17.56%	25.87%
动态重定位分区	36.74%	32.18%	31.75%	31.44%	31.08%	29.23%	28.08%	27.73%	23.87%	19.97%	29.21%
平均分配查找分区比较次数											
首次分配算法	9	8	8	8	8	8	8	7	6	6	7.6
动态重定位分区	10	10	10	9	9	9	9	8	8	8	9
内存空间平均利用率											
首次分配算法	53.62%	51.91%	51.86%	47.66%	46.60%	45.87%	44.04%	43.21%	42.83%	42.54%	47.02%
动态重定位分区	61.25%	54.01%	52.70%	51.40%	49.86%	49.36%	46.87%	45.42%	39.58%	36.67%	48.71%
平均分配查找分区比较次数											
首次分配算法	23	22	22	20	19	18	16	15	13	12	18
动态重定位分区	20	19	19	19	18	18	17	17	16	16	17.9

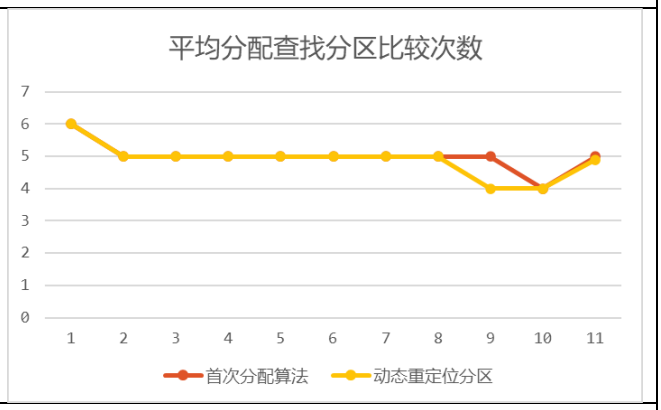
运行结果显示，在进程数量少、服务时间短、请求的内存大小较低时，内存整体利用率低的场景下，首次分配和动态重定位两种算法没有明显的性能差异；但随着进程数量、请求内存、服务时间、总数逐步增加，动态重定位分区算法逐渐表现出了更高的内存空间利用率、和更少的查找分区比较次数。

可以发现，随着用户对内存需求的不断增加，作为应运而生的改进技术，动态重定向分区技术确实能够解决内存空闲空间碎片化的问题。

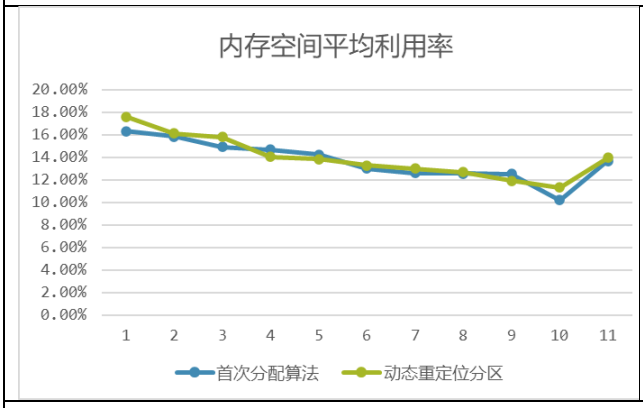
性能分析的折线统计图如下表所示（翻页）：

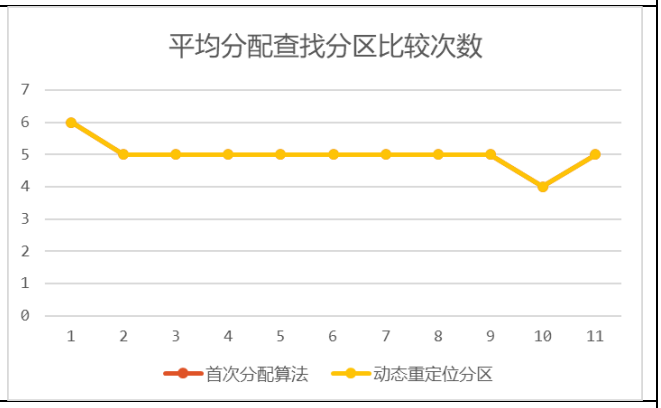
场景 1：随机生成进程的最大内存为 100MB，最大服务时间为 6，进程总数为 25



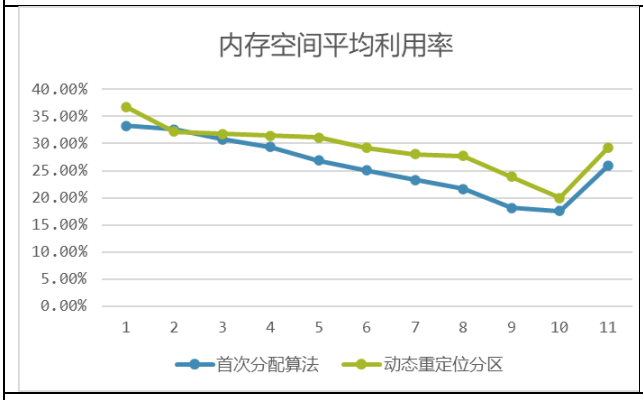


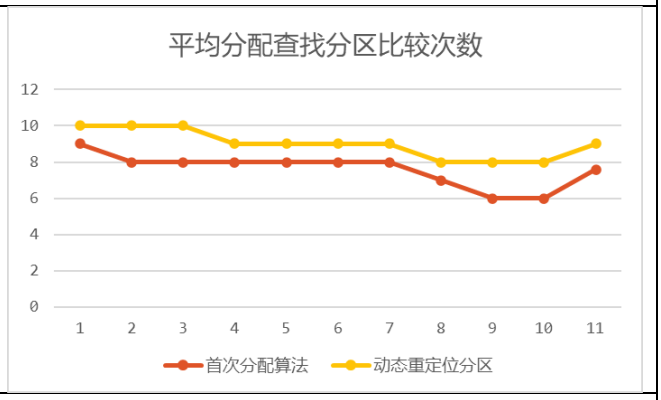
场景 2：随机生成进程的最大内存为 200MB，最大服务时间为 6，进程总数为 25



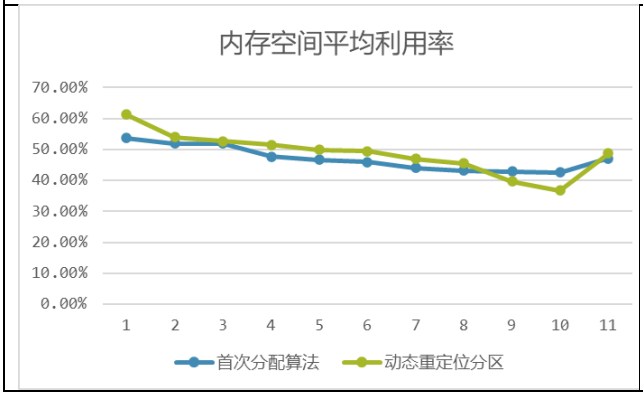


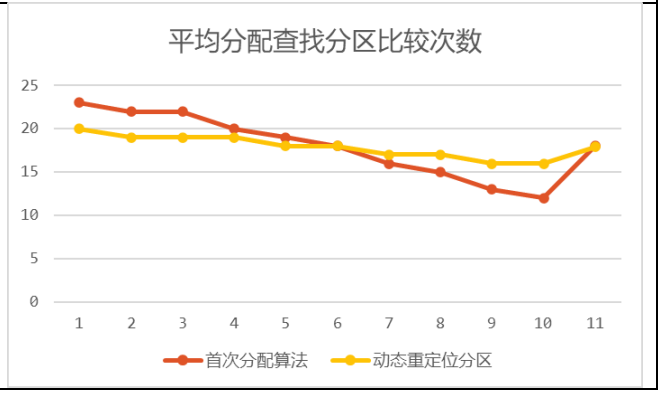
场景 3：随机生成进程的最大内存为 200MB，最大服务时间为 20，进程总数为 25



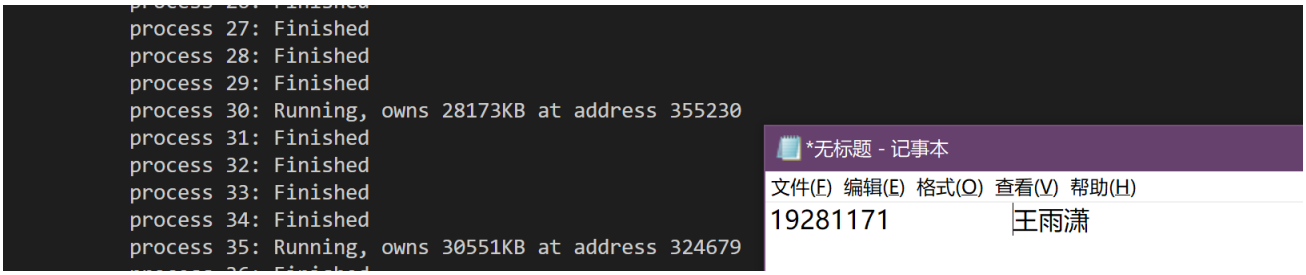


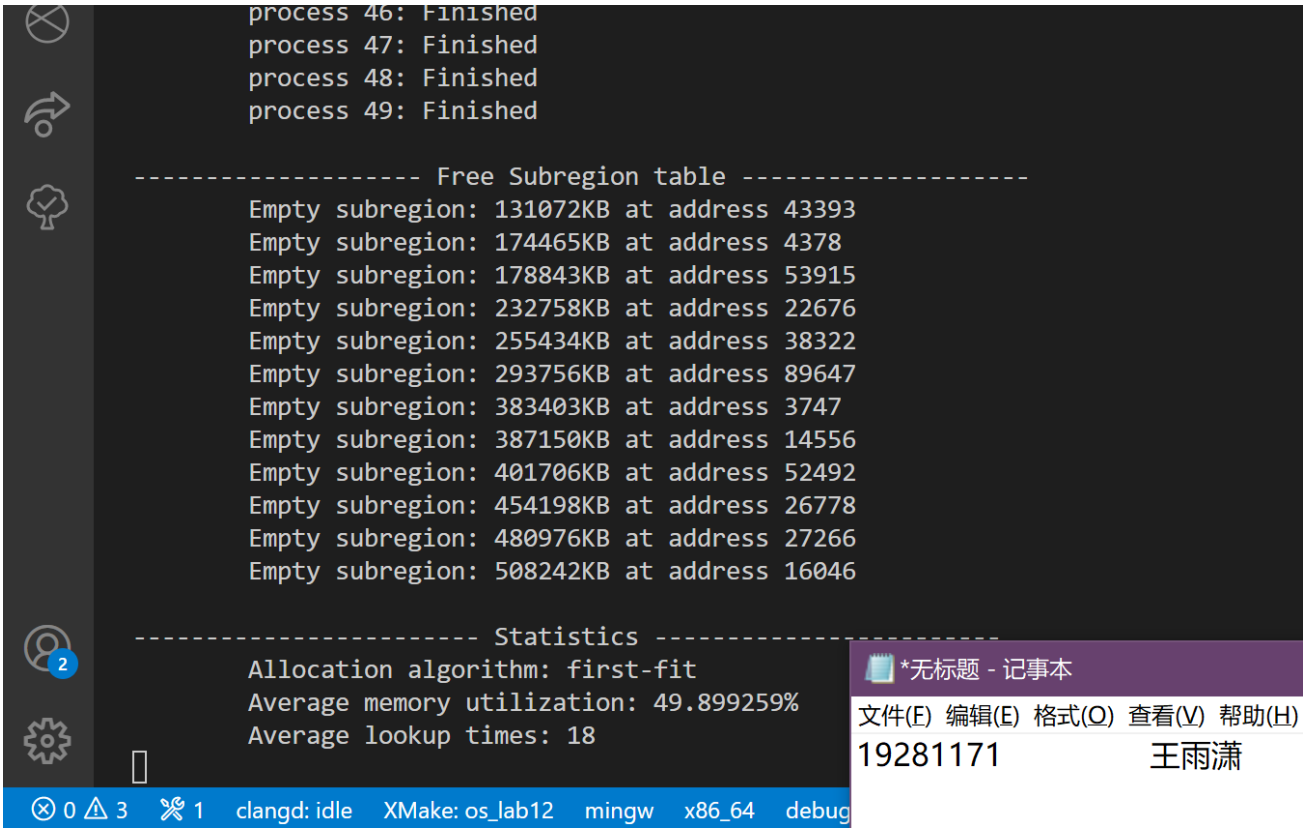
场景 4：随机生成进程的最大内存为 200MB，最大服务时间为 20，进程总数增加到 50





3.7. 运行结果截图展示





4. 实验结论和心得体会

在本次实验中，我耗费时间较长的步骤是用编程语言具体化内存管理的过程，通过本次实验，我了解了动态可重定位分区内存管理机制，通过在实践中通过设计数据结构和算法流程，对不同内存分配算法在不同场景下的性能表现有了更深入的理解。现有程序已知的问题：有时（发生频率大概在 10%）程序会在时间为 0 时直接退出算法，但不影响多次反复执行的后续结果，暂未确认造成这个错误的原因；

5. 附录：源程序的完整代码

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

/* 地址固定分配定义 */
#define physical_memory_size    (512 * 1024)    // 物理内存空间为 512MB，统一单位到 KB
#define system_memory_size      (128 * 1024)    // 操作系统占用低址的 128MB，统一单位到 KB
#define user_memory_size        (384 * 1024)    // 用户可使用高址的 384MB，统一单位到 KB
#define MByte                   1024            // 1MB = 1024KB
#define process_num             50              // 进程总数
#define process_max_size        (200 * MByte)   // 生成进程的最大内存需求

/* 进程的各种状态定义 */
#define unassigned  -1   // 进程未被分配空间
#define suspended    0   // 内存空间不足，进程挂起
#define assigned     1   // 进程已经被分配空间
#define finished     2   // 进程结束运行

/* 其他定义 */
#define author  19281171    // 署个名
#define bool    int         // 假装有 bool 类型
#define true    1
#define false   0

// 分区情况描述，双向链表结构
struct Subregion {
    // 数据部分
    int beginAddress;       // 分区起始地址
    int size;               // 分区大小
    bool isEmpty;           // 分区是否已经被占用

    struct Subregion *prev; // 前项指针，指向相邻的上个分区
    struct Subregion *next; // 后项指针，指向相邻的下个分区
};

// 进程描述
struct Process {
    int id;                     // 进程名
    struct Subregion *address;  // 进程的存储地址
    int size;                   // 进程大小
    int state;                  // 进程内存空间的状态
    int startTime;              // 进程的创建时间
    int remainingTime;          // 进程的剩余运行时间
};

// 初始化内存空间，操作系统占用低址的 128MB 不参与内存分配，而高址的 384MB 为用户区可供系统分配
struct Subregion* memorySpaceInit() {
    // 初始化双向链表头结点和尾结点
    struct Subregion* head = (struct Subregion*)malloc(sizeof(struct Subregion));
    struct Subregion* tail = (struct Subregion*)malloc(sizeof(struct Subregion));

    // 初始化操作系统低址，分配为已占用状态
    head->beginAddress = 0;
    head->size = system_memory_size;
    head->isEmpty = false;
    head->prev = NULL;
    head->next = tail;

    // 初始化用户空间地址，分配为未占用状态
    tail->beginAddress = system_memory_size;
    tail->size = user_memory_size;
    tail->isEmpty = true;
    tail->next = NULL;
    tail->prev = head;
```

```
    return head;
}

// 随机创建一个进程
struct Process* generateProcess(int id) {
    // 初始化进程
    struct Process* p = (struct Process*)malloc(sizeof(struct Process));
    p->id = id;
    p->address = NULL;
    p->size = rand() % (process_max_size + 1);  // 进程所需空间范围 [0, process_max_size]，单位 KB
    p->startTime = rand() % 25;                 // 进程开始时间范围 [0, 24]
    p->remainingTime = rand() % 21;             // 进程持续时间范围 [0, 20]
    p->state = unassigned;                      // 初始状态为未分配空间

    return p;
}

// 随机创建一个进程事件序列
struct Process** generateProcessQueue() {
    struct Process **array = malloc(process_num * sizeof(struct Process*));
    for(int i=0; i<process_num; i++) {
        array[i] = generateProcess(i);
    }

    return array;
}

// 判断能否在子空间 node 分配大小 process_size 的内存
bool canAllocateMemory(struct Subregion* node, int process_size) {
    if (node->isEmpty && node->size >= process_size) return true;
    else return false;
}

// 双向链表插入节点，为进程 process 分配子空间 node 空间和空闲空间
void allocateSubregion(struct Subregion* node, struct Process* process) {
    // 之前判定为能放下，才会进入这个函数
    if (node->size == process->size) {
        // 子空间大小刚好等于进程所需的空间，直接改
        node->isEmpty = false;
        process->address = node;
    } else {
        // 子空间 node 需要被拆分为分配给内存的子空间 processRegion，和空闲的子空间 restRegion 两个结
点
        struct Subregion* leftNode = node->prev;
        struct Subregion* rightNode = node->next;
        struct Subregion* processRegion = (struct Subregion* )malloc(sizeof(struct Subregion));
        struct Subregion* restRegion = (struct Subregion* )malloc(sizeof(struct Subregion));

        processRegion->beginAddress = node->beginAddress;
        processRegion->isEmpty = false;
        processRegion->size = process->size;

        restRegion->beginAddress = node->beginAddress + process->size;
        restRegion->isEmpty = true;
        restRegion->size = (node->size - process->size);

        // 双向链表重新连接
        // before:  leftNode <-> node <-> rightNode
        // after:   leftNode <-> processRegion <-> restRegion <-> rightNode
        if (leftNode != NULL) {
            leftNode->next = processRegion;
        }
        processRegion->prev = leftNode;
        processRegion->next = restRegion;
        restRegion->prev = processRegion;
        restRegion->next = rightNode;
        if (rightNode != NULL) {
            rightNode->prev = restRegion;
```

```
        }

        // 回收结点
        free(node);

        // 进程记录被分配的空间地址
        process->address = processRegion;
    }
}

// 双向链表删除节点，回收子空间 node 空间
void recoverySubregion(struct Subregion* node) {
    // 如果左右是空闲结点，就把 node 的 size 加给它们
    struct Subregion* leftNode = node->prev;
    struct Subregion* rightNode = node->next;
    if (leftNode != NULL && leftNode->isEmpty == true) {
        // node 空间合并到左结点
        leftNode->size += node->size;
        leftNode->next = rightNode;
        if (rightNode != NULL) {
            rightNode->prev = leftNode;
        }
        free(node);
    } else if (rightNode != NULL && rightNode->isEmpty == true) {
        // node 空间合并到右结点
        rightNode->beginAddress = node->beginAddress;
        rightNode->size += node->size;
        if (leftNode != NULL) {
            leftNode->next = rightNode;
        }
        rightNode->prev = leftNode;
        free(node);
    } else {
        // 如果左右结点都被占有或不存在，则把 node 的状态改成空
        node->isEmpty = true;
    }
}

// 计算已分配内存数值，顺便输出空闲分区表（包括各空闲分区起始地址和大小）
int totalAllocatedMemory(struct Subregion* node) {
    int res = 0;
    printf("\n-------------------- Free Subregion table --------------------\n");
    while (node != NULL) {
        // 遍历子空间
        if (node->isEmpty == false) {
            res += node->size;
        } else {
            printf("\tEmpty subregion: %dKB at address %d\n", node->beginAddress, node->size);
        }
        // 访问下一块子空间
        node = node->next;
    }
    return res;
}

// 内存紧凑拼接处理
void compactFreeSubregion(struct Subregion* head) {
    // 空闲的子空间大小之和
    int emptySize = 0;
    // leftNode 即上一个非空子空间指针
    struct Subregion *leftNode = head, *cur = head->next;
    while (cur != NULL) {
        if (cur->isEmpty == true) {
            // 该子空间为空
            emptySize += cur->size;
        } else {
            // 该子空间不为空，和上一个非空空间连接
            cur->prev = leftNode;
            leftNode->next = cur;
```

```
            leftNode = cur;
        }
        // 访问下一个子空间
        cur = cur->next;
    }
    // 把小空闲空间合并为一个最大的空闲空间
    struct Subregion *emptyRegion = (struct Subregion *)malloc(sizeof(struct Subregion));
    emptyRegion->beginAddress = leftNode->beginAddress + leftNode->size;
    emptyRegion->isEmpty = true;
    emptyRegion->size = emptySize;
    emptyRegion->next = NULL;
    emptyRegion->prev = leftNode;
    leftNode->next = emptyRegion;
}

// 分配策略 1: 首次适应算法
void firstFitAlgorithm(struct Subregion* head, struct Process** processes) {
    // 假定内存占用变化可划分为 N 个时间段
    // 而每个时间段的长度及已分配内存占总可分配内存的比率分别为 Ti 和 Ri
    // 内存空间平均利用率为 ∑(Ti × Ri) ÷ ∑(Ti)
    // 此处为 (rSum / T)
    int T = 0;
    double rSum = 0;
    bool hasFinished = false;
    int searchTimes = 0, allocTimes = 0;

    while (!hasFinished) {
        // 如果进程都结束了则结束外循环
        hasFinished = true;
        printf("\n------------ Time: %d --- Process States Diagram ------------\n", T);
        // 遍历进程
        for (int i=0; i<process_num; i++) {
            // 进程创建事件
            if (T == processes[i]->startTime) {
                hasFinished = false;
                printf("\tprocess %d: Created\n", processes[i]->id);
            }

            if ((processes[i]->state == unassigned && T >= processes[i]->startTime)
              || processes[i]->state == suspended) {
                hasFinished = false;
                // 如果到了/过了该进程创建的时间，则判断所有未分配空间、或挂起的进程，在该时间点能否被分
配空间
                struct Subregion* region = head;
                allocTimes++;
                while (region != NULL) {
                    searchTimes++;
                    // 遍历子空间判断能否给该进程分配空间
                    if (canAllocateMemory(region, processes[i]->size)) {
                        // 当该子空间可以存放该进程，链表分裂出新节点
                        allocateSubregion(region, processes[i]);
                        // 进程内存分配事件
                        printf("\tprocess %d: Allocated %dKB at address %d\n", processes[i]->id,
processes[i]->size, processes[i]->address->beginAddress);
                        processes[i]->state = assigned;
                        break;
                    }
                    // 访问下一个子空间
                    region = region->next;
                }

                if (region == NULL) {
                    // 遍历后发现现在没有空间放置该进程，状态改为挂起
                    processes[i]->state = suspended;
                    // 进程挂起事件
                    printf("\tprocess %d: Suspended\n", processes[i]->id);
                }
            } else if (processes[i]->state == assigned) {
                hasFinished = false;
```

```
                // 对于已经被分配空间的进程，剩余运行时间-1，判断它是否结束运行
                if (processes[i]->remainingTime-- == 0) {
                    // 进程撤销内存回收事件
                    processes[i]->state = finished;
                    recoverySubregion(processes[i]->address);
                    printf("\tprocess %d: Finished\n", processes[i]->id);
                } else {
                    // 进程正在运行
                    printf("\tprocess %d: Running, owns %dKB at address %d\n", processes[i]->id,
processes[i]->size, processes[i]->address->beginAddress);
                }
            } else if (processes[i]->state == finished) {
                // 进程已经结束运行
                printf("\tprocess %d: Finished\n", processes[i]->id);
            }
        }
        T++;    // 时间+1
        // 计算已分配内存占总可分配内存的比率
        rSum += (double)(totalAllocatedMemory(head) - system_memory_size) /
(double)(user_memory_size);
    }
    // 显示内存空间平均利用率和平均分配查找分区比较次数
    printf("\n----------------------- Statistics -----------------------\n");
    printf("\tAllocation algorithm: first-fit\n");
    printf("\tAverage memory utilization: %.6f%%\n", (rSum / T) * 100);
    printf("\tAverage lookup times: %d\n", (allocTimes != 0 ? (searchTimes / allocTimes) : 0));
}

// 分配策略 2：动态可重定位分区分配
void relocatePartition(struct Subregion* head, struct Process** processes) {
    int T = 0;
    double rSum = 0;
    bool hasFinished = false;
    int searchTimes = 0, allocTimes = 0;

    while (!hasFinished) {
        // 如果进程都结束了则结束外循环
        hasFinished = true;
        printf("\n------------ Time: %d --- Process States Diagram ------------\n", T);
        // 遍历进程
        for (int i=0; i<process_num; i++) {
            // 进程创建事件
            if (T == processes[i]->startTime) {
                hasFinished = false;
                printf("\tprocess %d: Created\n", processes[i]->id);
            }

            // 如果到了/过了该进程创建的时间，则判断所有未分配空间、或挂起的进程，在该时间点能否被分配空
间
            if ((processes[i]->state == unassigned && T >= processes[i]->startTime)
              || processes[i]->state == suspended) {
                hasFinished = false;
                struct Subregion* region = head;
                allocTimes++;
                while (region != NULL) {
                    searchTimes++;
                    // 遍历子空间判断能否给该进程分配空间
                    if (canAllocateMemory(region, processes[i]->size)) {
                        // 当该子空间可以存放该进程，链表分裂出新节点
                        allocateSubregion(region, processes[i]);
                        // 进程内存分配事件
                        printf("\tprocess %d: Allocated %dKB at address %d\n", processes[i]->id,
processes[i]->size, processes[i]->address->beginAddress);
                        processes[i]->state = assigned;
                        break;
                    }
                    // 访问下一个子空间
                    region = region->next;
                }
```

```
                if (region == NULL) {
                    // 遍历后发现现在没有空间放置该进程，状态改为挂起
                    processes[i]->state = suspended;
                    // 进程挂起事件
                    printf("\tprocess %d: Suspended\n", processes[i]->id);
                }
            } else if (processes[i]->state == assigned) {
                hasFinished = false;
                // 对于已经被分配空间的进程，剩余运行时间-1，判断它是否结束运行
                if (processes[i]->remainingTime-- == 0) {
                    // 进程撤销内存回收事件
                    processes[i]->state = finished;
                    recoverySubregion(processes[i]->address);
                    // 在发生回收事件时，使用紧凑技术
                    compactFreeSubregion(head);
                    printf("\tprocess %d: Finished\n", processes[i]->id);
                } else {
                    // 进程正在运行
                    printf("\tprocess %d: Running, owns %dKB at address %d\n", processes[i]->id,
processes[i]->size, processes[i]->address->beginAddress);
                }
            } else if (processes[i]->state == finished) {
                // 进程已经结束运行
                printf("\tprocess %d: Finished\n", processes[i]->id);
            }
        }
        T++;    // 时间+1
        // 计算已分配内存占总可分配内存的比率
        rSum += (double)(totalAllocatedMemory(head) - system_memory_size) /
(double)(user_memory_size);
    }
    // 显示内存空间平均利用率和平均分配查找分区比较次数
    printf("\n----------------------- Statistics -----------------------\n");
    printf("\tAllocation algorithm: dynamic relocation\n");
    printf("\tAverage memory utilization: %.6f%%\n", (rSum / T) * 100);
    printf("\tAverage lookup times: %d\n", (allocTimes != 0 ? (searchTimes / allocTimes) : 0));
}

int main(int argc, char** argv) {
    srand((unsigned)time(NULL));    // 初始化随机数种子

    for (int i=0; i<15; i++) {
        struct Subregion* head = memorySpaceInit();
        struct Process** process_array = generateProcessQueue();

        // 实验两种算法
        firstFitAlgorithm(head, process_array);
        // relocatePartition(head, process_array);

        getchar();      // 为了截图方便等一次按键
        free(head);     // 释放空间
        for(int i=0; i<process_num; i++) {
            free(process_array[i]);
        }
        Sleep(2000);    // 等待防止出错
    }

    return 0;
}
```