**吉林建筑科技学院**

**上机实验报告**

## 实验题目： 动态分区分配方式的模拟

课程名称： 操作系统原理

专业班级： 计算机2022专升本4班

学生姓名： 崔智辉

学生学号： 224030415

指导教师： 常颖

实验日期： 2023 年 6 月 1 日

第 十四 周 周 四 第 5 6 节

**计算机工程与人工智能学院**

|  |
| --- |
| **一、实验目的**  （1）掌握动态分区分配方式使用的数据结构和分配算法（首次/最佳/最坏适应算法）  （2）进一步加深对动态分区分配管理方式及实现过程的理解。 |
| **二、主要实验仪器、编译环境等**  （1）普通PC机  （2）C语言编译环境——DEV-C++/Visual Studio 2010 |
| **三、实验原理说明**  为了模拟实现首次/最佳/最坏适应分配算法的内存块分配与回收,首先封装了三个算法，分别是首次适应、最佳适应和最坏适应。   1. 首次适应算法（First Fit）：  * 遍历内存链表，查找第一个未分配且大小足够的内存块。 * 如果找到了匹配的内存块，根据进程大小进行分配： * 如果匹配的内存块大小与进程大小相等，直接将该内存块标记为已分配。 * 如果匹配的内存块大小大于进程大小，将该内存块分割成两个部分： * 一个已分配的内存块，大小为进程大小。 * 一个空闲的内存块，大小为原始内存块大小减去进程大小。 * 如果找不到足够大小的内存块，表示无法分配。  1. 最佳适应算法（Best Fit）：  * 遍历内存链表，查找所有未分配且大小足够的内存块。 * 在所有匹配的内存块中，选择大小最接近进程大小的内存块进行分配。 * 分配过程与首次适应算法类似。  1. 最坏适应算法（Worst Fit）：  * 遍历内存链表，查找所有未分配且大小足够的内存块。 * 在所有匹配的内存块中，选择大小最大的内存块进行分配。 * 分配过程与首次适应算法类似。   对于分配和释放内存的操作，代码使用了单链表来表示内存块。每个内存块结构体包含了起始地址、大小、分配状态和指向下一个内存块的指针。初始化内存时，创建一个具有给定大小的初始内存块，并将其设置为未分配状态。分配内存时，根据所选算法在链表中找到适合的内存块进行分配。释放内存时，将分配状态标记为未分配，并根据需要合并相邻的空闲内存块。  在用户界面中，用户可以选择分配内存、释放内存、显示内存状态或退出程序。根据用户选择的算法和输入的进程大小，执行相应的操作并显示相关信息。  当用户运行程序时，首先需要输入内存的大小。这个内存大小将作为一个初始的内存块，并被标记为未分配状态。然后，用户将进入一个循环菜单，可以选择以下操作：   1. 分配内存：用户需要输入要分配的进程大小和分配算法（1.首次适应 2.最佳适应 3.最坏适应）。根据选择的算法，程序将在内存链表中查找可用的内存块进行分配。如果找到合适的内存块，程序将进行分配，并显示分配成功的起始地址。如果找不到足够的内存空间，则显示无足够内存空间可供分配的消息。 2. 释放内存：用户需要输入要释放的进程大小。程序将在内存链表中查找已分配且大小匹配的内存块。如果找到匹配的内存块，程序将将其标记为未分配状态，并尝试合并相邻的空闲内存块。如果找不到要释放的内存块，则显示找不到要释放的内存块的消息。 3. 显示内存状态：程序将遍历内存链表，并显示当前的空闲分区和已分配分区的情况。每个分区都将显示起始地址和大小。 4. 退出：程序将结束循环，退出运行。 |
| 四、题目分析（设计算法描述）   1. 首次适应算法（First Fit）：  * 初始化一个内存链表，包含一个初始的未分配内存块，起始地址为0，大小为整个内存大小。 * 当接收到分配内存的请求时：   + 遍历内存链表，查找第一个未分配且大小足够的内存块。   + 如果找到匹配的内存块：     - 如果匹配的内存块大小与进程大小相等，将该内存块标记为已分配。     - 如果匹配的内存块大小大于进程大小，将该内存块分割成两个部分：     - 一个已分配的内存块，大小为进程大小。     - 一个空闲的内存块，大小为原始内存块大小减去进程大小。   + 如果找不到足够大小的内存块，表示无法分配该进程。 * 当接收到释放内存的请求时：   + 遍历内存链表，查找已分配且大小匹配的内存块。   + 如果找到匹配的内存块：     - 将该内存块标记为未分配状态。     - 合并相邻的空闲内存块，如果有的话。   + 如果找不到要释放的内存块，显示错误消息。 * 当接收到显示内存状态的请求时：   + 遍历内存链表，显示当前的空闲分区和已分配分区的情况。  1. 最佳适应算法（Best Fit）：  * 初始化一个内存链表，包含一个初始的未分配内存块，起始地址为0，大小为整个内存大小。 * 当接收到分配内存的请求时：   + 遍历内存链表，查找所有未分配且大小足够的内存块。   + 在所有匹配的内存块中，选择大小最接近进程大小的内存块。   + 如果找到匹配的内存块：     - 如果匹配的内存块大小与进程大小相等，将该内存块标记为已分配。     - 如果匹配的内存块大小大于进程大小，将该内存块分割成两个部分：       * 一个已分配的内存块，大小为进程大小。       * 一个空闲的内存块，大小为原始内存块大小减去进程大小。     - 如果找不到足够大小的内存块，表示无法分配该进程。 * 当接收到释放内存的请求时：   + 遍历内存链表，查找已分配且大小匹配的内存块。   + 如果找到匹配的内存块：     - 将该内存块标记为未分配状态。     - 合并相邻的空闲内存块，如果有的话。   + 如果找不到要释放的内存块，显示错误消息。 * 当接收到显示内存状态的请求时：   + 遍历内存链表，显示当前的空闲分区和已分配分区的情况。  1. 最坏适应算法（Worst Fit）：  * 初始化一个内存链表，包含一个初始的未分配内存块，起始地址为0，大小为整个内存大小。 * 当接收到分配内存的请求时：   + 遍历内存链表，查找所有未分配且大小足够的内存块。   + 在所有匹配的内存块中，选择大小最大的内存块进行分配。   + 如果找到匹配的内存块：     - 如果匹配的内存块大小与进程大小相等，将该内存块标记为已分配。     - 如果匹配的内存块大小大于进程大小，将该内存块分割成两个部分：       * 一个已分配的内存块，大小为进程大小。       * 一个空闲的内存块，大小为原始内存块大小减去进程大小。   + 如果找不到足够大小的内存块，表示无法分配该进程。 * 当接收到释放内存的请求时：   + 遍历内存链表，查找已分配且大小匹配的内存块。   + 如果找到匹配的内存块：     - 将该内存块标记为未分配状态。     - 合并相邻的空闲内存块，如果有的话。   + 如果找不到要释放的内存块，显示错误消息。 * 当接收到显示内存状态的请求时：   + 遍历内存链表，显示当前的空闲分区和已分配分区的情况。 |
| 1. **程序代码清单**  5.1 定义内存结构体代码清单 // 定义内存块结构体  typedef struct MemoryBlock {      int start\_address;  // 起始地址      int size;           // 大小      int is\_allocated;   // 是否已分配      struct MemoryBlock\* next;  // 下一个内存块的指针  } MemoryBlock;  // 全局变量，表示内存链表的头节点  MemoryBlock\* memory = NULL; 5.2 初始化内存代码清单 // 初始化内存  void initializeMemory(int size) {      memory = (MemoryBlock\*)malloc(sizeof(MemoryBlock));      memory->start\_address = 0;      memory->size = size;      memory->is\_allocated = 0;      memory->next = NULL;  } 5.3 首次适应算法分配内存代码清单 // 首次适应算法分配内存  void firstFit(int process\_size) {      MemoryBlock\* current = memory;      while (current != NULL) {          if (!current->is\_allocated && current->size >= process\_size) {              // 找到一个未分配且大小足够的内存块              if (current->size == process\_size) {                  current->is\_allocated = 1;                  printf("分配成功，起始地址：%d\n", current->start\_address);                  return;              } else {                  MemoryBlock\* new\_block = (MemoryBlock\*)malloc(sizeof(MemoryBlock));                  new\_block->start\_address = current->start\_address;                  new\_block->size = process\_size;                  new\_block->is\_allocated = 1;                  new\_block->next = current->next;                  current->size -= process\_size;                  current->start\_address += process\_size;                  current->next = new\_block;                  printf("分配成功，起始地址：%d\n", new\_block->start\_address);                  return;              }          }          current = current->next;      }      printf("无足够的内存空间可供分配\n");  } 5.4 最佳适应算法分配内存代码清单 // 最佳适应算法分配内存  void bestFit(int process\_size) {      MemoryBlock\* current = memory;      MemoryBlock\* best\_block = NULL;      int best\_fit = -1;      while (current != NULL) {          if (!current->is\_allocated && current->size >= process\_size) {              // 找到一个未分配且大小足够的内存块              if (best\_block == NULL || current->size < best\_fit) {                  best\_block = current;                  best\_fit = current->size;              }          }          current = current->next;      }      if (best\_block != NULL) {          if (best\_block->size == process\_size) {              best\_block->is\_allocated = 1;              printf("分配成功，起始地址：%d\n", best\_block->start\_address);          } else {              MemoryBlock\* new\_block = (MemoryBlock\*)malloc(sizeof(MemoryBlock));              new\_block->start\_address = best\_block->start\_address;              new\_block->size = process\_size;              new\_block->is\_allocated = 1;              new\_block->next = best\_block->next;              best\_block->size -= process\_size;              best\_block->start\_address += process\_size;              best\_block->next = new\_block;              printf("分配成功，起始地址：%d\n", new\_block->start\_address);          }      } else {          printf("无足够的内存空间可供分配\n");      }  } 5.4 最坏适应算法分配内存代码清单 // 最坏适应算法分配内存  void worstFit(int process\_size) {      MemoryBlock\* current = memory;      MemoryBlock\* worst\_block = NULL;      int worst\_fit = -1;      while (current != NULL) {          if (!current->is\_allocated && current->size >= process\_size) {              // 找到一个未分配且大小足够的内存块              if (worst\_block == NULL || current->size > worst\_fit) {                  worst\_block = current;                  worst\_fit = current->size;              }          }          current = current->next;      }      if (worst\_block != NULL) {          if (worst\_block->size == process\_size) {              worst\_block->is\_allocated = 1;              printf("分配成功，起始地址：%d\n", worst\_block->start\_address);          } else {              MemoryBlock\* new\_block = (MemoryBlock\*)malloc(sizeof(MemoryBlock));              new\_block->start\_address = worst\_block->start\_address;              new\_block->size = process\_size;              new\_block->is\_allocated = 1;              new\_block->next = worst\_block->next;              worst\_block->size -= process\_size;              worst\_block->start\_address += process\_size;              worst\_block->next = new\_block;              printf("分配成功，起始地址：%d\n", new\_block->start\_address);          }      } else {          printf("无足够的内存空间可供分配\n");      }  } 5.5分配内存代码清单 // 分配内存  void allocateMemory(int process\_size, int algorithm) {      switch (algorithm) {          case 1:              firstFit(process\_size);              break;          case 2:              bestFit(process\_size);              break;          case 3:              worstFit(process\_size);              break;          default:              printf("无效的算法选择\n");      }  } 5.6释放内存代码清单 // 释放内存  void deallocateMemory(int process\_size) {      MemoryBlock\* current = memory;      MemoryBlock\* prev = NULL;      while (current != NULL) {          if (current->is\_allocated && current->size == process\_size) {              // 找到已分配且大小匹配的内存块              current->is\_allocated = 0;              // 合并相邻的空闲内存块              if (current->next != NULL && !current->next->is\_allocated) {                  current->size += current->next->size;                  MemoryBlock\* temp = current->next;                  current->next = temp->next;                  free(temp);              }              if (prev != NULL && !prev->is\_allocated) {                  prev->size += current->size;                  prev->next = current->next;                  free(current);                  current = prev;              }              printf("释放成功，起始地址：%d\n", current->start\_address);              return;          }          prev = current;          current = current->next;      }      printf("找不到要释放的内存块\n");  } 5.7显示内存状态代码清单 // 显示内存状态  void printMemoryStatus() {      MemoryBlock\* current = memory;      printf("\n空闲分区：\n");      while (current != NULL) {          if (!current->is\_allocated) {              printf("起始地址：%d\t大小：%d\n", current->start\_address, current->size);          }          current = current->next;      }      current = memory;      printf("\n已分配分区：\n");      while (current != NULL) {          if (current->is\_allocated) {              printf("起始地址：%d\t大小：%d\n", current->start\_address, current->size);          }          current = current->next;      }  } 5.8主函数代码清单 int main() {      int memory\_size, choice, process\_size;      int exit\_flag = 0;      while (1) {          printf("请输入内存大小：");          if (scanf("%d", &memory\_size) == 1 && memory\_size > 0) {              break;          }          printf("无效的内存大小，请重新输入\n");          while (getchar() != '\n') continue; // 清空输入缓冲区      }      initializeMemory(memory\_size);      while (!exit\_flag) {          printf("\n1. 分配内存\n2. 释放内存\n3. 显示内存状态\n4. 退出\n");          printf("请选择操作：");          scanf("%d", &choice);          switch (choice) {              case 1:                  printf("请输入进程大小：");                  scanf("%d", &process\_size);                  printf("请选择分配算法（1.首次适应 2.最佳适应 3.最坏适应）：");                  scanf("%d", &choice);                  allocateMemory(process\_size, choice);                  break;              case 2:                  printf("请输入要释放的进程大小：");                  scanf("%d", &process\_size);                  deallocateMemory(process\_size);                  break;              case 3:                  printMemoryStatus();                  break;              case 4:                  exit\_flag = 1;                  break;              default:                  printf("无效的选择\n");          }      }      return 0;  } |
| **六、运行结果截图**  输入内存大小  **图6.1 初始化内存**  选项  **图6.2用户选项**  选择算法  **图6.3首次适应算法**  最佳适应  **图6.4最佳适应算法**  最坏适应  **图6.5最坏适应算法**  释放内存  **图6.6释放内存** |
| **六、实验结论及思考**  在实验中，我模拟实现了首次适应、最佳适应和最坏适应三种内存分配算法。通过观察实验结果，可以得出以下结论和思考：  1.首次适应算法：  优点：快速找到第一个符合大小要求的空闲内存块，适用于快速分配内存。  缺点：可能会造成内存碎片化，导致后续无法满足较大的内存分配请求。  2.最佳适应算法：  优点：选择最接近进程大小的空闲内存块，可以最大限度地减少内存碎片。  缺点：需要遍历所有的空闲内存块，效率较低。  3.最坏适应算法：  优点：选择最大的空闲内存块，可以避免大内存块的浪费。  缺点：可能会导致较多的内存碎片，无法充分利用内存空间。  思考：  内存分配算法的选择取决于具体应用场景和需求。在内存碎片较少且大内存块的需求较多时，最坏适应算法可能更为适用。而在需要快速分配内存且内存碎片较少的情况下，首次适应算法可能更合适。  可以考虑结合多种内存分配算法的优点，设计出更优化的算法。例如，可以在分配内存时根据内存大小选择首次适应或最佳适应算法，以平衡分配效率和内存碎片的问题。  内存管理的算法设计不仅仅包括内存分配，还需要考虑内存释放时的合并策略，以便更好地利用内存空间。  在实际应用中，可能会结合其他优化策略，如内存压缩、虚拟内存等，以提高内存利用率和性能。 |

**实验成绩评定（教师填写）**

|  |  |  |  |
| --- | --- | --- | --- |
| 考核项目 | 考核内容及评分标准 | 分值 | 得分 |
| 实验预习 | 明确实验目的，掌握实验原理 | 5 |  |
| 熟悉编译环境，掌握实验基本知识 | 5 |  |
| 实验纪律 | 按规定进入实验室， 无大声喧哗、打闹现象 | 10 |  |
| 保持实验环境卫生，实验结束后将仪器设备整理整齐 | 10 |  |
| 实验操作 | 正确使用仪器设备 | 10 |  |
| 实验步骤正确，独立完成实验操作 | 10 |  |
| 数据记录  与处理 | 图表规范整洁，结论真实准确 | 10 |  |
| 按实验要求处理数据，数据处理过程完整正确 | 10 |  |
| 结果与分析 | 有明确的实验结果或结论 | 10 |  |
| 对实验结果进行了分析，分析简洁、明确、合理 | 10 |  |
| 能正确回答实验思考题，实验讨论有一定的见解 | 10 |  |
| 上机实验成绩：  其他意见或建议：  指导教师（签名）： 年 月 日 | | | |