四川大学计算机学院、软件学院

实验报告

学号：2022141460155 姓名：林诺晗 专业：计算金融 班级：计金班 第12周

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 课程名称 | 操作系统实验 | | 实验课时 | 2 |
| 实验项目 | 动态分区分配方式的模拟 | | 实验时间 | 2024.05.16 |
| 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.  2）进一步加深对动态分区分配管理方式及其实现过程的理解. | | | |
| 实验环境 | VMware Workstation 17 Pro  Ubuntu 22.04.3 LTS | | | |
| 实验内容（算法、程序、步骤和方法） | 编写C语言程序，模拟实现首次/最佳/最坏适应算法的内存块分配和回收，要求每次分配和回收后显示出空闲分区和已分配分区的情况。假设初始状态下，可用的内存空间为640KB。  **实验要求:**  1）数据结构设计  已分配分区表、空闲分区表  2）分配算法设计  首次适应、最佳适应、最差适应分配算法  根据分配算法决定空闲分区表的排序  3）回收算法设计  考虑回收区所属的四种情况，有上空分区无下空分区、无上空分区有下空分区、上下分区都为空分区，上下都无空分区，根据情况来决定回收区的处理。  **实验相关原理：**  **Linux 内存管理简介**  物理内存管理器：负责物理内存的分配与回收，以页为单位实施管理，目的是提高性能，减少碎片。  虚拟内存管理器：它在物理内存管理器的基础上，通过页目录、页表和交换机制，为系统中的每个进程模拟了一个大小为4G的虚拟地址空间。  内核内存管理器：负责内核中小内存的分配和回收。  内核虚拟内存管理器：为了满足内核对大内存的需求，利用虚拟内存管理的思想，在内核虚拟地址空间实现内核虚拟内存管理。  用户空间内存管理器：负责进程用户态虚拟内存的动态分配和回收，它管理的内存在进程的堆中。  **内存操作函数**  1. 内存分配函数  alloc()、calloc()、malloc()和relloc()  2. 内存映射函数  mmap()  3. 取消映射函数  munmap()  4. 释放内存函数  free()  5. 取得内存分页大小  getpagesize()  **源代码：**  #include <stdio.h>  #include <stdlib.h>  #define TOTAL\_MEMORY 640 // 初始内存大小为640KB  typedef struct Block {  int start;  int size;  struct Block \*next;  int is\_allocated;  } Block;  Block \*free\_list = NULL; // 空闲分区链表  Block \*alloc\_list = NULL; // 已分配分区链表  void init\_memory() {  free\_list = (Block \*)malloc(sizeof(Block));  free\_list->start = 0;  free\_list->size = TOTAL\_MEMORY;  free\_list->next = NULL;  free\_list->is\_allocated = 0;  }  void print\_memory(Block \*list, const char \*type) {  Block \*current = list;  int index = 0;  printf("Index | Start | End | Size | %s\n", type);  printf("-------------------------------------------------\n");  while (current != NULL) {  printf("%5d | %6d | %6d | %4d\n", index++, current->start, current->start + current->size - 1, current->size);  current = current->next;  }  printf("\n");  }  Block \*find\_fit(Block \*\*free\_list, int size) {  Block \*current = \*free\_list;  while (current != NULL) {  if (current->is\_allocated == 0 && current->size >= size) {  return current;  }  current = current->next;  }  return NULL;  }  void split\_block(Block \*block, int size) {  int new\_size = block->size - size;  Block \*new\_block = (Block \*)malloc(sizeof(Block));  new\_block->start = block->start + size;  new\_block->size = new\_size;  new\_block->is\_allocated = 0;  new\_block->next = block->next;  block->size = size;  block->next = new\_block;  }  void allocate\_memory(Block \*\*free\_list, Block \*\*alloc\_list, int size) {  Block \*block\_to\_allocate = find\_fit(free\_list, size);  if (block\_to\_allocate != NULL) {  block\_to\_allocate->is\_allocated = 1;  // 如果分配后有剩余空间，则分割该块  if (block\_to\_allocate->size > size) {  split\_block(block\_to\_allocate, size);  }  // 将分配的块添加到已分配分区链表  Block \*new\_alloc\_block = (Block \*)malloc(sizeof(Block));  \*new\_alloc\_block = \*block\_to\_allocate; // 复制找到的块的信息  new\_alloc\_block->next = \*alloc\_list;  \*alloc\_list = new\_alloc\_block;  printf("Allocation Success! ADDRESS=%d\n", block\_to\_allocate->start);  } else {  printf("No suitable block found.\n");  }  print\_memory(\*free\_list, "Free");  print\_memory(\*alloc\_list, "Allocated");  }  void deallocate\_memory(Block \*\*free\_list, Block \*\*alloc\_list, int start) {  Block \*current\_alloc = \*alloc\_list;  Block \*prev\_alloc = NULL;  // 在已分配链表中找到对应的块  while (current\_alloc != NULL && current\_alloc->start != start) {  prev\_alloc = current\_alloc;  current\_alloc = current\_alloc->next;  }  if (current\_alloc != NULL) {  current\_alloc->is\_allocated = 0;  // 将回收的块添加回空闲分区链表  Block \*current\_free = \*free\_list;  while (current\_free->next != NULL && current\_free->next->start < start) {  current\_free = current\_free->next;  }  if (current\_free->start == start) {  // 与前一个空闲块合并  current\_free->size += current\_alloc->size;  if (current\_alloc->next != NULL) {  current\_free->next = current\_alloc->next;  } else {  current\_free->next = NULL;  }  free(current\_alloc);  } else {  current\_alloc->next = current\_free->next;  current\_free->next = current\_alloc;  }  // 从已分配链表中移除  if (prev\_alloc == NULL) {  \*alloc\_list = current\_alloc->next;  } else {  prev\_alloc->next = current\_alloc->next;  }  free(current\_alloc);  printf("Deallocation Success! ADDRESS=%d\n", start);  } else {  printf("Block not found.\n");  }  print\_memory(\*free\_list, "Free");  print\_memory(\*alloc\_list, "Allocated");  }  int main() {  init\_memory();  alloc\_list = (Block \*)malloc(sizeof(Block)); // 初始化已分配分区链表  alloc\_list->next = NULL;  // 打印初始内存状态  print\_memory(free\_list, "Free");  char choice;  int size, address;  do {  printf("Enter the allocate or reclaim (a/r), or press other key to exit: ");  scanf(" %c", &choice);  if (choice == 'a') {  printf("Input size to allocate: ");  scanf("%d", &size);  allocate\_memory(&free\_list, &alloc\_list, size);  } else if (choice == 'r') {  printf("Input address to reclaim: ");  scanf("%d", &address);  deallocate\_memory(&free\_list, &alloc\_list, address);  } else {  break;  }  } while (1);  return 0;  }          在分配内存时，如果找到合适的空闲块，并且该块的大小大于请求的大小，我们将该块分割成两部分：一部分用于满足当前的分配请求，另一部分仍然是空闲的。  已分配的块将从空闲链表中移除并添加到已分配链表中。在释放内存时，我们在已分配链表中找到对应的块，将其标记为未分配，并且如果它与前面的空闲块相邻，则将它们合并。然后，该块被移除已分配链表并放回空闲链表中。  ****编译程序****：使用编译器（如gcc）将.c文件编译成可执行文件。  ****运行程序****：编译成功后，会生成一个可执行文件，**按照实验相关要求**，程序运行如下：   1. **作业1 申请130 KB**     **（2）作业2 申请60 KB**    **（3）作业3 申请100 KB**    **（4）作业2 释放60 KB**     1. **作业3 释放100 KB**     **（6）作业1 释放130 KB** | | | |
| 结论  （结果） | 实验成功，按照相关要求完成程序编写及实现，并熟练掌握了相关命令的使用。  编写C语言程序，模拟实现了首次/最佳/最坏适应算法的内存块分配和回收，并在每次分配和回收后显示出空闲分区和已分配分区的情况。 | | | |
| 小结 | 1）掌握动态分区分配方式使用的数据结构和分配算法  2）进一步加深对动态分区分配管理方式及其实现过程的理解 | | | |
| 指导老师  评议 |  | | | |
| 成绩评定： | 指导教师签名： | | |