河南工业大学 操作系统原理 实验报告

班级：软件 1305 班　　学号：201316920311　　姓名：田劲锋　　指导老师：刘扬　　日期：2015 年 6 月 6 日

实验 4　可变分区的内存分配算法模拟

1. 实验步骤

1. 以下是memory.c的源代码，注释已详细给出：

Listing 1: memory.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <limits.h>
5
6  #define MEMMAN_N 4096
7
8  #define TASKLIST 4096
9
10 typedef struct MEMBLK { /* 内存块 */
11     int addr; /* 起始地址 */
12     int size; /* 大小 */
13     int proc; /* 进程号 */
14 } memblk;
15
16 typedef struct MEMMAN { /* 内存管理 */
17     int length;
18     memblk at[MEMMAN_N];
19 } memman;
20
21 void memblk_init(memblk* blk)
22 {
23     blk->addr = 0;
24     blk->size = 0;
25     blk->proc = -1;
26 }
27
28 void memblk_initd(memblk* blk, int addr, int size)
29 {
30     blk->addr = addr;
31     blk->size = size;
32     blk->proc = -1;
33 }
34
35 /* 打印内存管理情况表 */
36 void memman_print(memman* mm, int id)
37 {
38     printf("%d:", id);
39     int n = mm->length;
40     for (int i = 0; i < n; i++) {
41         if (mm->at[i].proc != -1) { /* 已分配 */
42             printf("  %d[%d](%d)", mm->at[i].addr, mm->at[i].size, mm->at[i].proc);
43         } else { /* 未分配 */
44             printf("  %d[%d]", mm->at[i].addr, mm->at[i].size);
45         }
46     }
47     printf(" \n");
48     return;
49 }
50
51 /* 插入一个块 */
52 int memman_insert(memman* mm, int index, memblk* blk)
```

```
53  {
54      if (index > mm->length) {
55          index = mm->length;
56      }
57      if (index < 0) {
58          index = 0;
59      }
60      int i;
61      for (i = mm->length - 1; i >= index; i--) {
62          mm->at[i + 1] = mm->at[i];
63      }
64      mm->length++;
65      mm->at[index] = *blk;
66      return 0;
67  }
68
69  /* 删除一个块 */
70  int memman_erase(memman* mm, int index)
71  {
72      if (index > mm->length    index < 0) {
73          return -1;
74      }
75      int i;
76      mm->length--;
77      for (i = index; i < mm->length; i++) {
78          mm->at[i] = mm->at[i + 1];
79      }
80      return 0;
81  }
82
83  /* 分配内存 */
84  void memman_alloc(memman* mm, int i, int pid, int size)
85  {
86      /* 有更大的空间 */
87      if (mm->at[i].size > size) {
88          /* 分成两个内存块后，为进程分配内存 */
89          memman_insert(mm, i, &mm->at[i]);
90          mm->at[i].size = size;
91          mm->at[i + 1].addr = mm->at[i].addr + size;
92          mm->at[i + 1].size -= size;
93          mm->at[i].proc = pid;
94      }
95      /* 恰好空间相等 */
96      if (mm->at[i].size == size) {
97          /* 直接分配给进程 */
98          mm->at[i].proc = pid;
99      }
100 }
101
102 /* 释放内存 */
103 void memman_free(memman* mm, int pid)
104 {
105     /* 遍历内存块 */
106     for (int i = 0; i < mm->length; i++) {
107         /* 找到请求释放内存的进程 */
108         if (mm->at[i].proc == pid) {
109             /* 释放当前块 */
110             mm->at[i].proc = -1;
111             /* 合并到前面的块 */
112             if (i > 0 && mm->at[i - 1].proc == -1) {
113                 mm->at[i - 1].size += mm->at[i].size;
114                 memman_erase(mm, i);
115                 i--;
116             }
117             /* 合并后面的块 */
118             if (i < mm->length - 1 && mm->at[i + 1].proc == -1) {
```

```
119                 mm->at[i].size += mm->at[i + 1].size;
120                 memman_erase(mm, i + 1);
121             }
122             break;
123         }
124     }
125     return;
126 }
127
128 void first_fit(memman* mm, int id, int pid, int op, int size)
129 {
130     if (op == 1) { /* 分配 */
131         /* 遍历查找第一个能够放得下的内存块 */
132         int i;
133         for (i = 0; i < mm->length; i++) {
134             if (mm->at[i].proc == -1 && mm->at[i].size >= size) {
135                 memman_alloc(mm, i, pid, size);
136                 break;
137             }
138         }
139     } else { /* 释放 */
140         memman_free(mm, pid);
141     }
142 }
143
144 void next_fit(memman* mm, int id, int pid, int op, int size)
145 {
146     static int next = -1; /* 使用静态变量记录上次位置 */
147     if (op == 1) { /* 分配 */
148         /* 从上次位置开始循环查找第一个能够放得下的内存块 */
149         int i;
150         for (i = next + 1; i != next; i = (i + 1) % mm->length) {
151             if (mm->at[i].proc == -1 && mm->at[i].size >= size) {
152                 next = i; /* 记录这次位置 */
153                 memman_alloc(mm, i, pid, size);
154                 break;
155             }
156         }
157     } else { /* 释放 */
158         memman_free(mm, pid);
159     }
160 }
161
162 void best_fit(memman* mm, int id, int pid, int op, int size)
163 {
164     if (op == 1) { /* 分配 */
165         int min = INT_MAX, index = -1;
166         /* 遍历查找能够被分配的最小内存块 */
167         int i;
168         for (i = 0; i < mm->length; i++) {
169             if (mm->at[i].proc == -1 && mm->at[i].size >= size && mm->at[i].size < min) {
170                 min = mm->at[i].size;
171                 index = i;
172             }
173         }
174         /* 如果能找到则分配之 */
175         if (index != -1) {
176             memman_alloc(mm, index, pid, size);
177         }
178     } else { /* 释放 */
179         memman_free(mm, pid);
180     }
181 }
182
183 void worst_fit(memman* mm, int id, int pid, int op, int size)
184 {
```

```
185     if (op == 1) { /* 分配 */
186         int max = INT_MIN, index = -1;
187         /* 遍历查找能够被分配的最大内存块 */
188         int i;
189         for (i = 0; i < mm->length; i++) {
190             if (mm->at[i].proc == -1 && mm->at[i].size >= size && mm->at[i].size > max) {
191                 max = mm->at[i].size;
192                 index = i;
193             }
194         }
195         /* 如果能找到则分配之 */
196         if (index != -1) {
197             memman_alloc(mm, index, pid, size);
198         }
199     } else { /* 释放 */
200         memman_free(mm, pid);
201     }
202 }
203
204 int memman_init(memman* mm, int n, int addr[], int size[])
205 {
206     int i;
207     mm->length = 0;
208     for (i = 0; i < n; i++) {
209         memblk_initd(&mm->at[mm->length++], addr[i], size[i]);
210     }
211     if (mm->length != n) {
212         return -1;
213     }
214     return 0;
215 }
216
217 int main(int argc, char* argv[])
218 {
219     memman* mm = (memman*)malloc(sizeof(memman));
220     if (mm == NULL) {
221         exit(-1);
222     }
223
224     /* 初始化内存空间 */
225     int i, n, addr[MEMMAN_N], size[MEMMAN_N];
226     scanf("%d", &n);
227     for (i = 0; i < n; i++) {
228         scanf("%d%d", addr + i, size + i);
229     }
230
231     /* 执行内存分配 */
232     int pid[TASKLIST], op[TASKLIST], siz[TASKLIST];
233     n = 0;
234     while (scanf("%d%d%d", pid + n, op + n, siz + n) == 3) {
235         n++; /* 格式为：进程号 分配还是释放 大小 */
236     }
237
238     /* 执行内存分配 */
239     printf("首次适应分配算法:\n");
240     memman_init(mm, n, addr, size);
241     for (i = 0; i < n; i++) {
242         memman_print(mm, i);
243         first_fit(mm, i, pid[i], op[i], siz[i]);
244     }
245     memman_print(mm, i);
246
247     printf("循环适应分配算法:\n");
248     memman_init(mm, n, addr, size);
249     for (i = 0; i < n; i++) {
250         memman_print(mm, i);
```

```
251             next_fit(mm, i, pid[i], op[i], siz[i]);
252         }
253         memman_print(mm, i);
254 
255         printf("最佳适应分配算法:\n");
256         memman_init(mm, n, addr, size);
257         for (i = 0; i < n; i++) {
258             memman_print(mm, i);
259             best_fit(mm, i, pid[i], op[i], siz[i]);
260         }
261         memman_print(mm, i);
262 
263         printf("最坏适应分配算法:\n");
264         memman_init(mm, n, addr, size);
265         for (i = 0; i < n; i++) {
266             memman_print(mm, i);
267             worst_fit(mm, i, pid[i], op[i], siz[i]);
268         }
269         memman_print(mm, i);
270 
271         return 0;
272 }
```

该程序的实现使用的是数组，以实现随机访问。当然也可以使用链表，对于此题来说，使用链表会得到更好的时间和空间效率。这里作为实验，就暂时用数组实现以减小难度。由于动态分配内存其实降低了效率，在实际的操作系统中，也有使用数组的实现。

内存块类型memblk是一个结构，存储该内存块的起始地址、内存块大小和进程编号。当然，当该内存块为空时进程编号指定为特殊值。内存管理使用了一个memman结构，包含了一个内存块的数组和其长度。memman_alloc用来分配内存，当空间较大时分成两个块，空间正好时直接分配。memman_free用来释放内存块，释放后会将前后的空闲块进行合并。程序中没有对超过申请大小的操作作错误检查。

我们为该程序准备了一个输入文件：

```
6
0 10
10 8
18 10
28 6
34 10
44 20

1 1 6
2 1 8
3 1 3
1 2 0
2 2 0
3 2 0
```

编译并执行该程序：

```
$ cc -Wall memory.c -o memory
$ ./memory < mtable.in > 1
```

得到输出结果如下，每行为一个操作后的内存分布图，其中每个内存块之间用竖线和空格隔开，每个内存块的表示方法为 <起始地址>[<内存块大小>](<进程编号>)：

```
首次适应分配算法：
0: |0[10] |10[8] |18[10] |28[6] |34[10] |44[20]|
1: |0[6](1) |6[4] |10[8] |18[10] |28[6] |34[10] |44[20]|
2: |0[6](1) |6[4] |10[8](2) |18[10] |28[6] |34[10] |44[20]|
3: |0[6](1) |6[3](3) |9[1] |10[8](2) |18[10] |28[6] |34[10] |44[20]|
4: |0[6] |6[3](3) |9[1] |10[8](2) |18[10] |28[6] |34[10] |44[20]|
5: |0[6] |6[3](3) |9[19] |28[6] |34[10] |44[20]|
6: |0[28] |28[6] |34[10] |44[20]|
循环适应分配算法：
0: |0[10] |10[8] |18[10] |28[6] |34[10] |44[20]|
1: |0[6](1) |6[4] |10[8] |18[10] |28[6] |34[10] |44[20]|
2: |0[6](1) |6[4] |10[8](2) |18[10] |28[6] |34[10] |44[20]|
3: |0[6](1) |6[4] |10[8](2) |18[3](3) |21[7] |28[6] |34[10] |44[20]|
4: |0[10] |10[8](2) |18[3](3) |21[7] |28[6] |34[10] |44[20]|
5: |0[18] |18[3](3) |21[7] |28[6] |34[10] |44[20]|
6: |0[28] |28[6] |34[10] |44[20]|
最佳适应分配算法：
0: |0[10] |10[8] |18[10] |28[6] |34[10] |44[20]|
1: |0[10] |10[8] |18[10] |28[6](1) |34[10] |44[20]|
2: |0[10] |10[8](2) |18[10] |28[6](1) |34[10] |44[20]|
3: |0[3](3) |3[7] |10[8](2) |18[10] |28[6](1) |34[10] |44[20]|
4: |0[3](3) |3[7] |10[8](2) |18[26] |44[20]|
5: |0[3](3) |3[41] |44[20]|
6: |0[44] |44[20]|
最坏适应分配算法：
0: |0[10] |10[8] |18[10] |28[6] |34[10] |44[20]|
1: |0[10] |10[8] |18[10] |28[6] |34[10] |44[6](1) |50[14]|
2: |0[10] |10[8] |18[10] |28[6] |34[10] |44[6](1) |50[8](2) |58[6]|
3: |0[3](3) |3[7] |10[8] |18[10] |28[6] |34[10] |44[6](1) |50[8](2) |58[6]|
4: |0[3](3) |3[7] |10[8] |18[10] |28[6] |34[16] |50[8](2) |58[6]|
5: |0[3](3) |3[7] |10[8] |18[10] |28[6] |34[30]|
6: |0[10] |10[8] |18[10] |28[6] |34[30]|
```

程序将四种算法分别在相同的数据集上执行了一遍，出现了不同的结果。其中，首次适应分配算法和循环适应分配算法的差别并不是很大，而最佳适应分配算法和最坏适应分配算法则表现出了截然不同的行为。

2. 下面来分析四个分配算法：
   (1) **首次适应分配算法**（first_fit），也叫**最先匹配法**，是最简单的一种算法。算法从链表的首结点开始，将每一个空闲结点的大小与待分配大小相比较，看看是不是大于或等于它，只要找到第一个符合大小要求的节点，就将内存分配到这里。这种算法查找的结点很少，速度很快。
   (2) **循环适应分配算法**（next_fit），也叫**下次匹配法**，与首次适应匹配算法是类似的，只不过每一次找到合适的内存块后，就记录下来当前位置。等到下一次查找的时候，从上次记录的位置开始查找，如果走到链表尾部，则循环到链表头直到找一遍找到。这种算法查找的结点也不多，速度也很快。缺点是，较大的空闲分区不容易保留。
   (3) **最佳适应分配算法**（best_fit），基本思路是搜索整个链表，将能够装得下该进程的最小空闲区域分配出去。这种算法是很慢的，因为它每次都要遍历整个链表。而且出人意料的是，这种算法在性能上比前两种还要差，因为每次选择的都是与进程大小最接近的空闲内存块，这导致了分割以后剩余的空间将会很小以至于无法使用，造成大量小的碎片，反而浪费了不少空间。
   (4) **最坏适应分配算法**（worst_fit）是为了克服最佳适应分配算法导致大量碎片产生的。与之相反，该算法每次找到空闲最大的空间来分配内存。可惜的是，这种算法虽然不会留下太多的小碎片，但却使得一个大的进程到来找不到合适的空闲分区。这种算法的性能也不是很理想。