

# 操作系统课程设计 Project 7

## Contiguous Memory Allocation

姓名：郭倩昀

班级：F1903303

学号：519021910095

Email: guoqianyun@sjtu.edu.cn

2021 年 5 月 22 日

### 目录

<b>1</b>	<b>Contiguous Memory Allocation</b>	<b>2</b>
1.1	实验内容与目标 . . . . .	2
1.2	实验过程及步骤 . . . . .	2
1.3	实验代码 . . . . .	3
1.4	实验测试 . . . . .	12
<b>2</b>	<b>Conclusion</b>	<b>13</b>
2.1	问题与解决方案 . . . . .	13
2.2	实验心得 . . . . .	13

# 1 Contiguous Memory Allocation

## 1.1 实验内容与目标

本实验需要利用 C 语言实现连续内存分配，支持功能如下：

- 支持 RQ 指令申请内存空间
- 支持 RL 指令释放内存空间
- 支持 C 指令整理内存空间
- 支持 STAT 指令输出当前内存分配情况

## 1.2 实验过程及步骤

- 设计内存分配结点

使用链接表来表示已经分配内存，结点类 `mem_node` 内有内存开始地址，内存结束地址，被分配的进程名称以及下一被分配内存的结点，并创建头结点初始化为 `NULL`。

- 为进程申请内存空间

设计 `request` 函数给进程申请内存空间，相应的参数有进程名称，需求内存大小，内存分配方案 (`first fit/best fit/worst fit`)。首先，如果内存未被分配 (头结点为 `NULL`)，则从开始地址分配内存，如果内存大小足够则创建相应的内存分配结点加入链表，否则报错退出。然后针对三种内存分配策略分别设计分配方案。由于空闲内存块有三类，一类是第一个被分配内存之前，一类是两个被分配内存之间，还有一类是最后一个被分配内存之后到结束，因此设计分配方案时扫描空闲内存块要分成三类。

- 对于 `first fit`，从第一个空闲内存块开始扫描，扫描到第一个可以容纳大小的空闲内存块就创建内存分配结点分配内存，若扫描结束都没有找到则报错退出。
- 对于 `best fit`，设计两个指针，一个记录当前大于内存需求且空间最小的空闲内存块，一个继续向后扫描空闲内存块以便比较与更新。扫描结束后若没有找到则报错退出，若找到则创建相应的内存分配结点加入链表。
- 对于 `worst fit`，与 `best fit` 类似，设计两个指针，一个记录当前大于内存需求且空间最大的空闲内存块，一个继续向后扫描空闲内存块以便比较与更新。扫描结束后若没有找到则报错退出，若找到则创建相应的内存分配结点加入链表。

- 为进程释放内存空间

设计 `release` 函数给进程释放内存空间，传入参数进程名称。从内存分配链表的头结点开始查找对应进程的内存分配结点，找到之后将结点删除并释放空间，若没有搜索到则报错退出。

- 整理内存空间

设计 `compact` 函数整理内存空间。从头结点开始扫描，按顺序修改每一个内存分配空间的开始地址和结束地址。

- 输出当前内存分配情况

设计 `report` 函数报告当前内存分配情况。从头结点开始扫描，打印每个区间的使用情况，注意未分配的内存地址空间也要报告情况。

- 设计 main() 函数

这里将指令分析步骤放在 main() 函数开始。在标准化处理输入指令后，先检测特殊指令“X”退出，“C”整理，“STAT”报告，调用相应函数。若检测到 RQ 指令，从原命令中获取进程名称，需求内存大小和内存分配策略，调用 request 函数；若检测到 RL 指令，从原命令中获取进程名称，调用 release 函数。过程中需要有应对指令不合法的异常处理，及时报错退出。

### 1.3 实验代码

#### allocator.c

```
1 # include <stdio.h>
2 # include <stdlib.h>
3 # include <string.h>
4
5 # define MAX_LINE 500
6 # define TRUE 1
7
8 int memory;
9 typedef struct mem_node{
10     char *process;
11     int start;
12     int end;
13     struct mem_node *next;
14 }mem_node;
15 struct mem_node *head=NULL;
16 //request for a contiguous block of memory
17 int request(char *process, int size, char strategy);
18 //release of a contiguous block of memory
19 int release(char *process);
20 //compact unused holes of memory into one single block
21 void compact();
22 //report the regions of free and allocated memory
23 void report();
24
25 int main(int argc, char *argv[])
26 {
27     if (argc != 2)
28     {
29         fprintf(stderr, "  ERROR: Arguments Error\n");
30         exit(1);
31     }
32     //get memory size
33     memory=atoi(argv[1]);
34     char arg[MAX_LINE];    //full instruction
35     char op[MAX_LINE];     //operation
36     char process[MAX_LINE]; //process name
37     while(TRUE)
38     {
39         for (int i = 0; i < MAX_LINE; ++ i) //initialize
40         {
41             arg[i] = 0;
42             op[i] = 0;
43             process[i] = 0;
44         }
45         fprintf(stdout, "allocator> ");
```

```

46     fgets(arg, MAX_LINE, stdin);
47
48     //standardize arg
49     char tmp[MAX_LINE];
50     int last_blank = 1;
51     int dex = 0;
52     for (int i = 0; arg[i]; ++ i) {
53         if (arg[i] == ' ' || arg[i] == '\t' || arg[i] == '\n') {
54             if (last_blank == 0) {
55                 last_blank = 1;
56                 tmp[dex ++] = ' ';
57             }
58             else {
59                 tmp[dex++] = arg[i];
60                 last_blank = 0;
61             }
62         }
63         if (dex > 0 && tmp[dex - 1] == ' ') dex --;
64         for (int i = 0; i < dex; ++ i) arg[i] = tmp[i];
65         arg[dex] = 0;
66
67         if (strcmp(arg, "X") == 0) //EXIT
68             break;
69         if (strcmp(arg, "C") == 0) //COMPACT
70         {
71             compact();
72             continue;
73         }
74         if (strcmp(arg, "STAT") == 0)//REPORT
75         {
76             report();
77             continue;
78         }
79
80         for(dex=0;arg[dex];dex++)
81         {
82             if(arg[dex]==' ')
83                 break;
84         }
85
86         //get op
87         for (int i = 0; i < dex; ++ i)
88             { op[i] = arg[i];}
89         op[dex] = 0;
90
91         //op
92         if (strcmp(op, "RQ") == 0) //new process request
93         {
94             if (arg[dex] == 0)
95             {
96                 fprintf(stderr, " ERROR: Invalid input\n");
97                 continue;
98             }
99
100             int i;
101             int invalid= 0;
102             int size = 0;

```

```

103     char strategy;
104     //get process name
105     for (i = dex + 1; arg[i]; i++)
106     {
107         if (arg[i] == ' ') break;
108         process[i-dex-1] = arg[i];
109     }
110     process[i-dex-1]=0;
111     if (arg[i] == 0)
112     {
113         fprintf(stderr, "  ERROR: Invalid input\n");
114         continue;
115     }
116
117     //get size and check
118     dex = i;
119     for (i = dex + 1; arg[i]; ++ i)
120     {
121         if(arg[i] == ' ') break;
122         if(arg[i] < '0' || arg[i] > '9')
123         {
124             invalid = 1;
125             break;
126         }
127         size = size * 10 + arg[i] - '0';
128     }
129     if (invalid || arg[i] == 0)
130     {
131         fprintf(stderr, "  ERROR: Invalid input\n");
132         continue;
133     }
134     if (size<=0)
135     {
136         fprintf(stderr, "  ERROR: Size invalid\n");
137         continue;
138     }
139
140     //get strategy and check
141     dex = i;
142     if(arg[dex+1]==0||arg[dex+2]!=0)
143     {
144         fprintf(stderr, "  ERROR: Invalid input\n");
145         continue;
146     }
147     strategy=arg[dex+1];
148     request(process, size, strategy);
149 }
150 else if (strcmp(op, "RL") == 0)
151 {
152     if (arg[dex] == 0) {
153         fprintf(stderr, "  ERROR: Invalid input\n");
154         continue;
155     }
156     int invalid= 0;
157     //get process name
158     int i;
159     for (i = dex + 1; arg[i]; i++) {

```

```

160         if (arg[i] == ' ')
161             { invalid=1;break;}
162         process[i-dex-1] = arg[i];
163     }
164     process[i-dex-1]=0;
165     if (invalid) {
166         fprintf(stderr, " ERROR: Invalid input\n");
167         continue;
168     }
169     release(process);
170 }
171 else
172 {
173     fprintf(stderr, " ERROR: Invalid input\n");
174     continue;
175 }
176 }
177 return 0;
178 }
179
180 //request for a contiguous block of memory
181 int request(char *process, int size, char strategy)
182 {
183     int name_len=strlen(process);
184     int hole_len;
185     if (head == NULL) {
186         if (size <= memory) {
187             head = (mem_node *) malloc (sizeof(mem_node));
188             head -> process = (char *) malloc (sizeof(char) * (name_len + 1));
189             strcpy(head -> process, process);
190             head -> start = 0;
191             head -> end = 0 + size - 1;
192             head -> next = NULL;
193             return 0;
194         } else {
195             fprintf(stderr, " ERROR: No enough space\n");
196             return 1;
197         }
198     }
199
200 //first fit
201 if(strategy=='F')
202 {
203     mem_node *p = head;
204
205     //first hole
206     hole_len = p -> start - 0;
207     if (size <= hole_len) //fit in
208     {
209         mem_node *tmp = head;
210         head = (mem_node *) malloc (sizeof(mem_node));
211         head -> process = (char *) malloc (sizeof(char) * (name_len + 1));
212         strcpy(head -> process, process);
213         head -> start = 0;
214         head -> end = 0 + size - 1;
215         head -> next = tmp;
216         return 0;

```

```

217     }
218
219     //middle
220     while (p -> next != NULL)    //search
221     {
222         hole_len = p->next->start - p->end - 1;
223         if (size <= hole_len)    //fit in
224         {
225             mem_node *tmp = p->next;
226             p->next = (mem_node *) malloc (sizeof(mem_node));
227             p->next->process = (char *) malloc (sizeof(char) * (name_len + 1));
228             strcpy(p->next->process, process);
229             p->next->start = p->end + 1;
230             p->next->end = p->next->start + size - 1;
231             p->next->next = tmp;
232             return 0;
233         }
234         p = p -> next;
235     }
236
237     //last hole
238     hole_len = memory - p->end - 1;
239     if (size <= hole_len)
240     {
241         p->next = (mem_node *) malloc (sizeof(mem_node));
242         p->next->process = (char *) malloc (sizeof(char) * (name_len + 1));
243         strcpy(p->next->process, process);
244         p->next->start = p->end + 1;
245         p->next->end = p->next->start + size - 1;
246         p->next->next = NULL;
247         return 0;
248     }
249
250     // No enough space
251     fprintf(stderr, " ERROR: No enough space\n");
252     return 1;
253 }
254
255 //best fit
256 if (strategy == 'B')
257 {
258     mem_node *p = head;
259     int min_best = memory;
260     int type = 0;
261     mem_node *best;
262
263     //search hole
264     //first hole
265     hole_len = p -> start - 0;
266     if (size <= hole_len && hole_len < min_best) {
267         min_best = hole_len;
268         type = 1;
269     }
270     //middle
271     while (p -> next != NULL) {
272         hole_len = p -> next -> start - p -> end - 1;
273         if (size <= hole_len && hole_len < min_best) {

```

```

274         min_best = hole_len;
275         type = 2;
276         best = p;
277     }
278     p = p -> next;
279 }
280 //last hole
281 hole_len = memory - p -> end - 1;
282 if (size <= hole_len && hole_len < min_best) {
283     min_best = hole_len;
284     type = 3;
285 }
286 // No enough space
287 if (type == 0) {
288     fprintf(stderr, "[Err] No enough spaces!\n");
289     return 1;
290 }
291
292 //allocate memory
293 //first hole
294 if (type == 1)
295 {
296     mem_node *tmp = head;
297     head = (mem_node *) malloc (sizeof(mem_node));
298     head -> process = (char *) malloc (sizeof(char) * (name_len + 1));
299     strcpy(head -> process, process);
300     head -> start = 0;
301     head -> end = 0 + size - 1;
302     head -> next = tmp;
303     return 0;
304 }
305 //middle
306 if (type == 2)
307 {
308     p = best;
309     mem_node *tmp = p->next;
310     p->next = (mem_node *) malloc (sizeof(mem_node));
311     p->next->process = (char *) malloc (sizeof(char) * (name_len + 1));
312     strcpy(p->next->process, process);
313     p->next->start = p->end + 1;
314     p->next->end = p->next->start + size - 1;
315     p->next->next = tmp;
316     return 0;
317 }
318 //last hole
319 if (type == 3)
320 {
321     p->next = (mem_node *) malloc (sizeof(mem_node));
322     p->next->process = (char *) malloc (sizeof(char) * (name_len + 1));
323     strcpy(p->next->process, process);
324     p->next->start = p->end + 1;
325     p->next->end = p->next->start + size - 1;
326     p->next->next = NULL;
327     return 0;
328 }
329 }
330

```



```

331 //worst fit
332 if (strategy== 'W')
333 {
334     mem_node *p = head;
335     int max_worst = 0;
336     int type = 0;
337     mem_node *worst;
338
339     //search hole
340     //first hole
341     hole_len = p -> start - 0;
342     if (size <= hole_len && hole_len > max_worst)
343     {
344         max_worst = hole_len;
345         type = 1;
346     }
347     //middle
348     while (p -> next != NULL)
349     {
350         hole_len = p -> next -> start - p -> end - 1;
351         if (size <= hole_len && hole_len > max_worst) {
352             max_worst = hole_len;
353             type = 2;
354             worst = p;
355         }
356         p = p -> next;
357     }
358     //last hole
359     hole_len = memory - p -> end - 1;
360     if (size <= hole_len && hole_len > max_worst)
361     {
362         max_worst = hole_len;
363         type = 3;
364     }
365     // No enough space
366     if (type == 0) {
367         fprintf(stderr, "[Err] No enough spaces!\n");
368         return 1;
369     }
370
371     //allocate memory
372     //first hole
373     if (type == 1)
374     {
375         mem_node *tmp = head;
376         head = (mem_node *) malloc (sizeof(mem_node));
377         head -> process = (char *) malloc (sizeof(char) * (name_len + 1));
378         strcpy(head -> process, process);
379         head -> start = 0;
380         head -> end = 0 + size - 1;
381         head -> next = tmp;
382         return 0;
383     }
384     //middle
385     if (type == 2)
386     {
387         p = worst;

```

```

388     mem_node *tmp = p->next;
389     p->next = (mem_node *) malloc (sizeof(mem_node));
390     p->next->process = (char *) malloc (sizeof(char) * (name_len + 1));
391     strcpy(p->next->process, process);
392     p->next->start = p->end + 1;
393     p->next->end = p->next->start + size - 1;
394     p->next->next = tmp;
395     return 0;
396 }
397 //last hole
398 if (type == 3)
399 {
400     p->next = (mem_node *) malloc (sizeof(mem_node));
401     p->next->process = (char *) malloc (sizeof(char) * (name_len + 1));
402     strcpy(p->next->process, process);
403     p->next->start = p->end + 1;
404     p->next->end = p->next->start + size - 1;
405     p->next->next = NULL;
406     return 0;
407 }
408 }
409 //error argument
410 fprintf(stderr, " ERROR: Arguments Error\n");
411 return 1;
412
413 }
414
415 //release of a contiguous block of memory
416 int release(char *process)
417 {
418     mem_node *p = head;
419     if (head == NULL) {
420         fprintf(stderr, " ERROR: No such process\n");
421         return 1;
422     }
423     //first
424     if (strcmp(head->process, process) == 0) {
425         mem_node *tmp = head;
426         head = head->next;
427         free(tmp->process);
428         free(tmp);
429         return 0;
430     }
431     //search
432     while (p->next != NULL)
433     {
434         if (strcmp(p->next->process, process) == 0)
435         {
436             mem_node *tmp = p->next;
437             p->next = p->next->next;
438             free(tmp->process);
439             free(tmp);
440             return 0;
441         }
442         p = p->next;
443     }
444     //not found

```

```

445     fprintf(stderr, "  ERROR: No such process\n");
446     return 1;
447 }
448
449 //compact unused holes of memory into one single block
450 void compact()
451 {
452     int pos = 0;
453     mem_node *p = head;
454     while (p != NULL) {
455         int size = p->end - p->start + 1;
456         p->start = pos;
457         p->end = pos + size - 1;
458         pos += size;
459         p = p->next;
460     }
461     return;
462 }
463 //report the regions of free and allocated memory
464 void report()
465 {
466     mem_node *p = head;
467     //first
468     if (head == NULL)
469     {
470         fprintf(stdout, "  Address [0 : %d] Unused\n", memory - 1);
471         return ;
472     }
473     else if (head -> start != 0)
474     {
475         fprintf(stdout, "  Address [0 : %d] Unused\n", head -> start - 1);
476     }
477     //middle
478     while (p -> next != NULL) {
479         fprintf(stdout, "  Address [%d : %d] Process %s\n", p->start, p->end, p->process);
480         if (p->next->start - p->end - 1 > 0)
481         {
482             fprintf(stdout, "  Address [%d : %d] Unused\n", p->end + 1, p->next->start - 1);
483         }
484         p = p->next;
485     }
486
487     fprintf(stdout, "  Address [%d : %d] Process %s\n", p->start, p->end, p->process);
488     if (memory - p->end - 1 > 0)
489         fprintf(stdout, "  Address [%d : %d] Unused\n", p->end + 1, memory - 1);
490
491 }

```

## 1.4 实验测试

- allocator 测试

测试指令如下

```
1 make
2 ./allocator 1048576
3 RQ P1 10000 B
4 RQ P2 20000 F
5 RQ P3 30000 W
6 RQ P4 40000 B
7 STAT
8 RL P3
9 RQ P5 30001 F
10 STAT
11 RQ P6 10000 B
12 RQ P7 10000 F
13 RQ P8 10000 W
14 STAT
15 RL P6
16 RL P5
17 STAT
18 C
19 STAT
20 X
```

```
gqy@gqy-VirtualBox:~/os_proj7$ make
gcc -Wall -c allocator.c
gcc -Wall -o allocator allocator.o
gqy@gqy-VirtualBox:~/os_proj7$ ./allocator 1048576
allocator> RQ P1 10000 B
allocator> RQ P2 20000 F
allocator> RQ P3 30000 W
allocator> RQ P4 40000 B
allocator> STAT
  Address [0 : 9999] Process P1
  Address [10000 : 29999] Process P2
  Address [30000 : 59999] Process P3
  Address [60000 : 99999] Process P4
  Address [100000 : 1048575] Unused
allocator> RL P3
allocator> RQ P5 30001 F
allocator> STAT
  Address [0 : 9999] Process P1
  Address [10000 : 29999] Process P2
  Address [30000 : 59999] Unused
  Address [60000 : 99999] Process P4
  Address [100000 : 130000] Process P5
  Address [130001 : 1048575] Unused
allocator> RQ P6 10000 B
allocator> RQ P7 10000 F
allocator> RQ P8 10000 W
allocator> STAT
  Address [0 : 9999] Process P1
  Address [10000 : 29999] Process P2
  Address [30000 : 39999] Process P6
  Address [40000 : 49999] Process P7
  Address [50000 : 59999] Unused
  Address [60000 : 99999] Process P4
  Address [100000 : 130000] Process P5
  Address [130001 : 140000] Process P8
  Address [140001 : 1048575] Unused
```

图 1: allocator 测试 1

```
allocator> RL P6
allocator> RL P5
allocator> STAT
Address [0 : 9999] Process P1
Address [10000 : 29999] Process P2
Address [30000 : 39999] Unused
Address [40000 : 49999] Process P7
Address [50000 : 59999] Unused
Address [60000 : 99999] Process P4
Address [100000 : 130000] Unused
Address [130001 : 140000] Process P8
Address [140001 : 1048575] Unused
allocator> C
allocator> STAT
Address [0 : 9999] Process P1
Address [10000 : 29999] Process P2
Address [30000 : 39999] Process P7
Address [40000 : 79999] Process P4
Address [80000 : 89999] Process P8
Address [90000 : 1048575] Unused
allocator> X
gqy@gqy-VirtualBox:~/os_proj7$
```

图 2: allocator 测试 2

首先用 Makefile 文件编译，生成可执行文件 allocator，指定内存大小 1048576. 然后分别分配给进程 P1, P2, P3, P4 内存并打印状态；释放 P3，分配进程 P5 后打印状态；以三种分配方式分配 P6, P7, P8 后打印状态；释放 P6, P5 后打印状态；compact 指令整理后再次打印状态；最后退出。测试结果如图 1 和图 2。

## 2 Conclusion

### 2.1 问题与解决方案

本次 project7 主要实现了连续内存分配的管理，连续内存空间分配的三种分配策略理解上比较简单，程序的实现整体上难度也不大。实验中比较重要的地方在于内存分配链表的维护，但只要编程仔细保持清醒的头脑就可以顺利完成。

### 2.2 实验心得

本次 project7 顺利实现了连续内存分配管理与维护，虽然难度不大但是再设计内存分配链表以及实现各种分配算法的时候还是要考虑全面，谨慎操作，否则很容易出错。总的来说本次 project 让我熟悉了数据结构的使用，锻炼了程序设计能力也进一步加深了我对内存管理的理解。