

# Project 7: Contiguous Memory Allocation

CS307-Operating System (D), CS356-Operating System Course Design, Chentao Wu, Spring 2020.

Name: 方泓杰(Hongjie Fang) Student ID: 518030910150 Email: galaxies@sjtu.edu.cn

## 1 Contiguous Memory Allocation

### 1.1 Requirements

In Section 9.2 in textbook, we presented different algorithms for contiguous memory allocation. This project will involve managing a contiguous region of memory of size  $MAX$  where address may range from  $0 \cdots (MAX - 1)$ . This project requires us to respond to four different requests.

- Request for a contiguous block of memory;
- Release of a contiguous block of memory;
- Compact unused holes of memory into one single block;
- Report the regions of free and allocated memory.

The program will be passed the initial amount of memory at startup, and it will allocate memory using one of the three approaches highlighted in Section 9.2.2, depending on the flag that is passed to RQ commands. The flags are:

- F: first-fit;
- B: best-fit;
- W: worst-fit.

This will require the program keep track of the different holes representing available memory. When a request for memory arrives, it will allocate the memory from one of the available holes based on the allocation strategy. If there is insufficient memory to allocate to a request, it will output an error message and reject the request.

The program will also need to keep track of which region of memory has been allocated to which process. This is necessary to support the STAT command and is also needed when memory is released via the RL command, as the process releasing memory is passed to this command. If a partition being released is adjacent to an existing hole, be sure to combine the two holes into a single hole.

If the user enters C command, the program will compact the set of holes into one larger hole. There are several strategies for implementing compaction, one of which is suggested in Section 9.2.3 in textbook. Be sure to update the beginning address of any processes that have been affected by compaction.

### 1.2 Methods

Here are some specific methods of the contiguous memory allocator program.

- We use the linked list to keep track of every allocated regions of memory, and the rest of the memory is the holes. Since we have the size information about the memory, we can keep track of every free regions of memory according to the data we have. Moreover, the adjacent holes causing by RL command is automatically combined, which makes our programming easier.

- For first-fit algorithm, we examine the holes one by one, and if we found a suitable hole, then we will put the process into it.
- For best-fit algorithm and worst-fit algorithm, we examine all holes, and put the process into one suitable hole according to the type of the algorithm.
- For **STAT** command, we will traverse the linked list and print out all information about the memory allocation.
- For compaction command, we put all the allocated memory in the front contiguous region, therefore the holes are automatically compacted into a big one in the end of the memory.

### 1.3 Implementation

Here is the specific implementation of the contiguous memory allocator (`allocator.c`).

```

1 # include <stdio.h>
2 # include <stdlib.h>
3 # include <string.h>
4
5 # define LINE_MAX 256
6
7 int memory_limit;
8
9 typedef struct memory_node {
10     // belong
11     char *bel;
12     // begin, end
13     int beg, end;
14     // next
15     struct memory_node * nxt;
16 } mem_node;
17
18 mem_node *head = NULL;
19
20
21 // Requested by process [process_name], size of [size], memory allocation type [
    type].
22 // If success, return 0; or return 1 with the error message printed in the screen
    .
23 int request(char *process_name, int size, char type) {
24     if (size <= 0) {
25         fprintf(stderr, "[Err] the process size should be positive!\n");
26         return 1;
27     }
28
29     int name_len = strlen(process_name);
30     if (head == NULL) {
31         if (size <= memory_limit) {
32             head = (mem_node *) malloc (sizeof(mem_node));
33             head -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
34             strcpy(head -> bel, process_name);

```

```

35     head -> beg = 0;
36     head -> end = 0 + size - 1;
37     head -> nxt = NULL;
38     return 0;
39 } else {
40     // No enough spaces
41     fprintf(stderr, "[Err] No enough spaces!\n");
42     return 1;
43 }
44 }
45
46 // First-fit
47 if (type == 'F') {
48     mem_node *p = head;
49     int hole_len;
50
51     hole_len = p -> beg - 0;
52     if (size <= hole_len) {
53         mem_node *tmp = head;
54         head = (mem_node *) malloc (sizeof(mem_node));
55         head -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
56         strcpy(head -> bel, process_name);
57         head -> beg = 0;
58         head -> end = 0 + size - 1;
59         head -> nxt = tmp;
60         return 0;
61     }
62
63     while (p -> nxt != NULL) {
64         hole_len = p -> nxt -> beg - p -> end - 1;
65         if (size <= hole_len) {
66             mem_node *tmp = p -> nxt;
67             p -> nxt = (mem_node *) malloc (sizeof(mem_node));
68             p -> nxt -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
69             strcpy(p -> nxt -> bel, process_name);
70             p -> nxt -> beg = p -> end + 1;
71             p -> nxt -> end = p -> nxt -> beg + size - 1;
72             p -> nxt -> nxt = tmp;
73             return 0;
74         }
75         p = p -> nxt;
76     }
77
78     hole_len = memory_limit - p -> end - 1;
79     if (size <= hole_len) {
80         p -> nxt = (mem_node *) malloc (sizeof(mem_node));
81         p -> nxt -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
82         strcpy(p -> nxt -> bel, process_name);
83         p -> nxt -> beg = p -> end + 1;
84         p -> nxt -> end = p -> nxt -> beg + size - 1;

```

```

85     p -> nxt -> nxt = NULL;
86     return 0;
87 }
88
89 // No enough spaces
90 fprintf(stderr, "[Err] No enough spaces!\n");
91 return 1;
92 }
93
94 // Best-fit
95 if (type == 'B') {
96     mem_node *p = head;
97     int hole_len;
98
99     // position
100    int cur_min = 2e9;
101    int pos_flag = 0;
102    mem_node *pos;
103
104    hole_len = p -> beg - 0;
105    if (size <= hole_len && hole_len < cur_min) {
106        cur_min = hole_len;
107        pos_flag = 1;
108    }
109
110    while (p -> nxt != NULL) {
111        hole_len = p -> nxt -> beg - p -> end - 1;
112        if (size <= hole_len && hole_len < cur_min) {
113            cur_min = hole_len;
114            pos_flag = 2;
115            pos = p;
116        }
117        p = p -> nxt;
118    }
119
120    hole_len = memory_limit - p -> end - 1;
121    if (size <= hole_len && hole_len < cur_min) {
122        cur_min = hole_len;
123        pos_flag = 3;
124    }
125
126    // No enough spaces
127    if (pos_flag == 0) {
128        fprintf(stderr, "[Err] No enough spaces!\n");
129        return 1;
130    }
131
132    if (pos_flag == 1) {
133        mem_node *tmp = head;
134        head = (mem_node *) malloc (sizeof(mem_node));

```

```

135     head -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
136     strcpy(head -> bel, process_name);
137     head -> beg = 0;
138     head -> end = 0 + size - 1;
139     head -> nxt = tmp;
140     return 0;
141 }
142
143 if (pos_flag == 2) {
144     p = pos;
145     mem_node *tmp = p -> nxt;
146     p -> nxt = (mem_node *) malloc (sizeof(mem_node));
147     p -> nxt -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
148     strcpy(p -> nxt -> bel, process_name);
149     p -> nxt -> beg = p -> end + 1;
150     p -> nxt -> end = p -> nxt -> beg + size - 1;
151     p -> nxt -> nxt = tmp;
152     return 0;
153 }
154
155 if (pos_flag == 3) {
156     p -> nxt = (mem_node *) malloc (sizeof(mem_node));
157     p -> nxt -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
158     strcpy(p -> nxt -> bel, process_name);
159     p -> nxt -> beg = p -> end + 1;
160     p -> nxt -> end = p -> nxt -> beg + size - 1;
161     p -> nxt -> nxt = NULL;
162     return 0;
163 }
164 }
165
166 // Worst-fit
167 if (type == 'W') {
168     mem_node *p = head;
169     int hole_len;
170
171     // position
172     int cur_max = -2e9;
173     int pos_flag = 0;
174     mem_node *pos;
175
176     hole_len = p -> beg - 0;
177     if (size <= hole_len && hole_len > cur_max) {
178         cur_max = hole_len;
179         pos_flag = 1;
180     }
181
182     while (p -> nxt != NULL) {
183         hole_len = p -> nxt -> beg - p -> end - 1;
184         if (size <= hole_len && hole_len > cur_max) {

```

```

185     cur_max = hole_len;
186     pos_flag = 2;
187     pos = p;
188 }
189 p = p -> nxt;
190 }
191
192 hole_len = memory_limit - p -> end - 1;
193 if (size <= hole_len && hole_len > cur_max) {
194     cur_max = hole_len;
195     pos_flag = 3;
196 }
197
198 // No enough spaces
199 if (pos_flag == 0) {
200     fprintf(stderr, "[Err] No enough spaces!\n");
201     return 1;
202 }
203
204 if (pos_flag == 1) {
205     mem_node *tmp = head;
206     head = (mem_node *) malloc (sizeof(mem_node));
207     head -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
208     strcpy(head -> bel, process_name);
209     head -> beg = 0;
210     head -> end = 0 + size - 1;
211     head -> nxt = tmp;
212     return 0;
213 }
214
215 if (pos_flag == 2) {
216     p = pos;
217     mem_node *tmp = p -> nxt;
218     p -> nxt = (mem_node *) malloc (sizeof(mem_node));
219     p -> nxt -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
220     strcpy(p -> nxt -> bel, process_name);
221     p -> nxt -> beg = p -> end + 1;
222     p -> nxt -> end = p -> nxt -> beg + size - 1;
223     p -> nxt -> nxt = tmp;
224     return 0;
225 }
226
227 if (pos_flag == 3) {
228     p -> nxt = (mem_node *) malloc (sizeof(mem_node));
229     p -> nxt -> bel = (char *) malloc (sizeof(char) * (name_len + 1));
230     strcpy(p -> nxt -> bel, process_name);
231     p -> nxt -> beg = p -> end + 1;
232     p -> nxt -> end = p -> nxt -> beg + size - 1;
233     p -> nxt -> nxt = NULL;
234     return 0;

```

```

235     }
236 }
237
238 fprintf(stderr, "[Err] Argument Error!\n");
239 return 1;
240 }
241
242
243 // Release the resources allocated to process [process_name].
244 // If success, return 0; or return 1 with the error message printed in the screen
245 .
246 int release(char *process_name) {
247     if (head == NULL) {
248         fprintf(stderr, "[Err] No such process!\n");
249         return 1;
250     }
251     if (strcmp(head -> bel, process_name) == 0) {
252         mem_node *tmp = head;
253         head = head -> nxt;
254         free(tmp -> bel);
255         free(tmp);
256         return 0;
257     }
258     mem_node *p = head;
259     while (p -> nxt != NULL) {
260         if (strcmp(p -> nxt -> bel, process_name) == 0) {
261             mem_node *tmp = p -> nxt;
262             p -> nxt = p -> nxt -> nxt;
263             free(tmp -> bel);
264             free(tmp);
265             return 0;
266         }
267         p = p -> nxt;
268     }
269     fprintf(stderr, "[Err] No such process!\n");
270     return 1;
271 }
272
273
274
275 // Compact the holes.
276 void compact() {
277     int cur = 0;
278     mem_node *p = head;
279     while (p != NULL) {
280         int size = p -> end - p -> beg + 1;
281         p -> beg = cur;
282         p -> end = cur + size - 1;
283         cur += size;

```

```

284     p = p -> nxt;
285 }
286 return ;
287 }
288
289
290 // Display the statistics.
291 void display() {
292     if (head == NULL) {
293         fprintf(stdout, "Address [0 : %d] Unused\n", memory_limit - 1);
294         return ;
295     } else if (head -> beg != 0) {
296         fprintf(stdout, "Address [0 : %d] Unused\n", head -> beg - 1);
297     }
298
299     mem_node *p = head;
300     int hole_len;
301
302     while (p -> nxt != NULL) {
303         fprintf(stdout, "Address [%d : %d] Process %s\n", p -> beg, p -> end, p -> bel
            );
304         hole_len = p -> nxt -> beg - p -> end - 1;
305         if (hole_len > 0)
306             fprintf(stdout, "Address [%d : %d] Unused\n", p -> end + 1, p -> nxt -> beg
                - 1);
307         p = p -> nxt;
308     }
309
310     fprintf(stdout, "Address [%d : %d] Process %s\n", p -> beg, p -> end, p -> bel);
311     hole_len = memory_limit - p -> end - 1;
312     if (hole_len > 0)
313         fprintf(stdout, "Address [%d : %d] Unused\n", p -> end + 1, memory_limit - 1);
314 }
315
316
317 // Standardize input.
318 void standardize_input(char *op) {
319     char tmp[LINE_MAX];
320     int no_need = 1, tmpn = 0;
321
322     for (int i = 0; op[i]; ++ i) {
323         if (op[i] == ' ' || op[i] == '\t' || op[i] == '\n') {
324             if (no_need == 0) {
325                 no_need = 1;
326                 tmp[tmpn ++] = ' ';
327             }
328         } else {
329             tmp[tmpn ++] = op[i];
330             no_need = 0;
331         }

```



```

332     }
333
334     if (tmpn > 0 && tmp[tmpn - 1] == ' ') tmpn--;
335
336     for (int i = 0; i < tmpn; ++ i) op[i] = tmp[i];
337     op[tmpn] = 0;
338 }
339
340
341 int main(int argc, char *argv[]) {
342     if (argc != 2) {
343         fprintf(stderr, "[Err] Arguments Error!\n");
344         exit(1);
345     }
346
347     memory_limit = atoi(argv[1]);
348
349     char op[LINE_MAX], oper[LINE_MAX];
350     char process_name[LINE_MAX];
351     while(1) {
352         for (int i = 0; i < LINE_MAX; ++ i) {
353             op[i] = 0; oper[i] = 0;
354             process_name[i] = 0;
355         }
356
357         fprintf(stdout, "allocator> ");
358         fgets(op, LINE_MAX, stdin);
359
360         standardlize_input(op);
361
362         if (strcmp(op, "EXIT") == 0) break;
363
364         if (strcmp(op, "C") == 0) {
365             compact();
366             continue;
367         }
368
369         if (strcmp(op, "STAT") == 0) {
370             display();
371             continue;
372         }
373
374         // parse the inputs.
375         int pos = 0;
376         for (; op[pos]; ++ pos)
377             if(op[pos] == ' ') break;
378
379         for (int i = 0; i < pos; ++ i)
380             oper[i] = op[i];
381         oper[pos] = 0;

```

```

382
383 if (strcmp(oper, "RQ") == 0) {
384     if (op[pos] == 0) {
385         fprintf(stderr, "[Err] Invalid input!\n");
386         continue;
387     }
388
389     int i;
390     for (i = pos + 1; op[i]; ++ i) {
391         if (op[i] == ' ') break;
392         process_name[i - pos - 1] = op[i];
393     }
394
395     if (op[i] == 0) {
396         fprintf(stderr, "[Err] Invalid input!\n");
397         continue;
398     }
399
400     int invalid_input = 0;
401     int size = 0;
402     pos = i;
403     for (i = pos + 1; op[i]; ++ i) {
404         if (op[i] == ' ') break;
405         if (op[i] < '0' || op[i] > '9') {
406             invalid_input = 1;
407             break;
408         }
409         size = size * 10 + op[i] - '0';
410     }
411
412     if (invalid_input || op[i] == 0) {
413         fprintf(stderr, "[Err] Invalid input!\n");
414         continue;
415     }
416
417     if ((i - pos - 1) >= 10) {
418         fprintf(stderr, "[Err] Size too large!\n");
419         continue;
420     }
421
422     pos = i;
423     for (i = pos + 1; op[i]; ++ i) {
424         if (op[i] == ' ') {
425             invalid_input = 1;
426             break;
427         }
428     }
429
430     if (invalid_input || (i - pos - 1) != 1) {
431         fprintf(stderr, "[Err] Invalid input!\n");

```

```

432         continue;
433     }
434
435     request(process_name, size, op[pos + 1]);
436 } else if (strcmp(oper, "RL") == 0) {
437     if (op[pos] == 0) {
438         fprintf(stderr, "[Err] Invalid input!\n");
439         continue;
440     }
441
442     int invalid_input = 0;
443     for (int i = pos + 1; op[i]; ++ i) {
444         if (op[i] == ' ') {
445             invalid_input = 1;
446             break;
447         }
448         process_name[i - pos - 1] = op[i];
449     }
450
451     if (invalid_input) fprintf(stderr, "[Err] Invalid input!\n");
452     else release(process_name);
453 } else
454     fprintf(stderr, "[Err] Invalid input!\n");
455 }
456
457 return 0;
458 }

```

## 1.4 Testing

I write a Makefile file to help testing the program. We only need to enter the following instructions in the terminal and we can begin testing, and the argument of the `allocator` program stands for the memory limit.

```

1 make
2 ./allocator 1048576
3 RQ P1 10000 B
4 RQ P2 20000 B
5 RQ P3 30000 B
6 RQ P4 40000 B
7 STAT
8 RL P2
9 RQ P5 20001 F
10 STAT
11 RQ P6 10000 B
12 RQ P7 10000 W
13 RQ P8 10000 F
14 STAT
15 RL P6
16 RL P3

```

```
17 RL P5
18 STAT
19 C
20 STAT
21 EXIT
```

Here is the execution result of the memory allocation program (Fig. 1).

```
galaxies@ubuntu:~/CS307-Projects/Project7$ ./allocator 1048576
allocator> RQ P1 10000 B
allocator> RQ P2 20000 B
allocator> RQ P3 30000 B
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 P2
allocator> RQ P5 20001 F
allocator> STAT
Address [0 : 9999] Process P1
Address [10000 : 29999] Unused
Address [30000 : 59999] Process P3
Address [60000 : 99999] Process P4
Address [100000 : 120000] Process P5
Address [120001 : 1048575] Unused
allocator> RQ P6 10000 B
allocator> RQ P7 10000 W
allocator> RQ P8 10000 F
allocator> STAT
Address [0 : 9999] Process P1
Address [10000 : 19999] Process P6
Address [20000 : 29999] Process P8
Address [30000 : 59999] Process P3
Address [60000 : 99999] Process P4
Address [100000 : 120000] Process P5
Address [120001 : 130000] Process P7
Address [130001 : 1048575] Unused
allocator> RL P6
allocator> RL P3
allocator> RL P5
allocator> STAT
Address [0 : 9999] Process P1
Address [10000 : 19999] Unused
Address [20000 : 29999] Process P8
Address [30000 : 59999] Unused
Address [60000 : 99999] Process P4
Address [100000 : 120000] Unused
Address [120001 : 130000] Process P7
Address [130001 : 1048575] Unused
allocator> C
allocator> STAT
Address [0 : 9999] Process P1
Address [10000 : 19999] Process P8
Address [20000 : 59999] Process P4
Address [60000 : 69999] Process P7
Address [70000 : 1048575] Unused
allocator> EXIT
```

Figure 1: The execution result of the memory allocation program

## 2 Personal Thoughts

The project helps me understand the contiguous memory allocation algorithm better, including the first-fit algorithm, the best-fit algorithm and the worst-fit algorithm. The project also improves my understanding of memory compaction. The project trains our coding skills since we also need to implement parsing process and the shell-like windows except the algorithm. The implementation of the algorithms is quite simple but it needs patience. I enjoy the process of writing this program.

By the way, you can find all the source codes in the “src” folder. You can also refer to [my github](#) to see my codes of this project, and they are in the **Project7** folder.