

# 北京交通大学实验报告

课程名称 : 操作系统  
实验题目 : 动态可重定位分区内存管理模拟设计与实现  
学号 : 21281280  
姓名 : 柯劲帆  
班级 : 物联网2101班  
指导老师 : 何永忠  
报告日期 : 2023年11月26日

## 目录

目录

### 1. 开发运行环境和工具

### 2. 实验过程、分析和结论

2.1. 初始化

2.2. 申请内存分配操作

2.3. 回收

2.4. 其他代码

附录

## 1. 开发运行环境和工具

| 操作系统           | Linux内核版本                     | 处理器                                         | GCC版本                                 |
|----------------|-------------------------------|---------------------------------------------|---------------------------------------|
| Deepin<br>20.9 | 5.18.17-amd64-<br>desktop-hwe | Intel(R) Core(TM) i3-2330M<br>CPU @ 2.20GHz | gcc (Uos 8.3.0.3-<br>3+rebuild) 8.3.0 |

- 其他工具:
  - 编辑: VSCode
  - 编译和运行: Terminal

## 2. 实验过程、分析和结论

### 2.1. 初始化

```
1 #include <stdio.h>
2 #include <malloc.h>
3 #include <string.h>
4 #include <stdlib.h>
5
```

```

6  #define MAX_NUM_TASKS 128
7
8  struct Block {
9      int start;
10     int size;
11     int pid;
12     struct Block* next, * last;
13 };
14
15 struct Task {
16     int start;
17     int size;
18     int pid;
19     struct Block* block;
20 };
21
22 struct Request {
23     int size;
24     struct Request* next;
25 };
26
27
28 int preprocess_task_size(int);
29 int find_free_block_ff(int, struct Block**);
30 int find_free_block_bf(int, struct Block**);
31 void queue_push(int);
32 int queue_pop();
33 int get_command(FILE*);
34 int fork_process(int);
35 void kill_process(int);
36 int run(char*);
37 void init();
38 void clean_stdin();
39 void execute_wait_list();
40 void print();
41
42 int sys_size = 128 << 20, usr_size = 384 << 20;
43 struct Block* sys_space = NULL, * usr_space = NULL;
44 struct Task* tasks = NULL;
45 struct Request* waitlist_begin = NULL, * waitlist_end = NULL;
46 int waitlist_size = 0;
47 int last_pid = -1;
48 int match_method = -1;
49
50
51 int main() {
52     init();
53     while (get_command(stdin) >= 0);
54     return 0;
55 }
56
57 void init() {
58     sys_space = (struct Block*)malloc(sizeof(struct Block));
59     sys_space->pid = -1;
60     sys_space->start = 0;
61     sys_space->size = sys_size;

```

```

62     sys_space->next = NULL;
63     sys_space->last = NULL;
64
65     usr_space = (struct Block*)malloc(sizeof(struct Block));
66     usr_space->pid = -1;
67     usr_space->start = sys_size;
68     usr_space->size = usr_size;
69     usr_space->next = NULL;
70     usr_space->last = NULL;
71
72     tasks = (struct Task*)malloc(MAX_NUM_TASKS * sizeof(struct Task));
73     for (int i = 0; i < MAX_NUM_TASKS; i++) tasks[i].pid = -1;
74
75 ask_match_method:
76     printf("Choose a matching method:\n 1. First Fit\n 2. Best Fit\n");
77     printf("choose> ");
78     scanf("%d", &match_method);
79     clean_stdin();
80     if (match_method != 1 && match_method != 2) {
81         printf("Bad input. Choose again.\n");
82         goto ask_match_method;
83     }
84
85     waitlist_begin = (struct Request*)malloc(sizeof(struct Request));
86     waitlist_end = waitlist_begin;
87     waitlist_begin->next = NULL;
88     waitlist_begin->size = -1;
89 }

```

## 2.2. 申请内存分配操作

```

1  int fork_process(int size) {
2      int pid;
3      int task_index = -1;
4      repeat:
5          last_pid++;
6          if (last_pid < 0) last_pid = 0;
7          pid = last_pid;
8          for (int i = 0; i < MAX_NUM_TASKS; i++) {
9              if (tasks[i].pid >= 0 && tasks[i].pid == pid) goto repeat;
10         }
11         for (int i = 0; i < MAX_NUM_TASKS; i++) {
12             if (tasks[i].pid >= 0) continue;
13             task_index = i;
14             break;
15         }
16         if (task_index == -1) return -1;
17
18         size = preprocess_task_size(size);
19         tasks[task_index].size = size;
20         int start_loc = -1;
21         if (match_method == 1) start_loc = find_free_block_ff(size,
&tasks[task_index].block);

```

```

22     else start_loc = find_free_block_bf(size, &tasks[task_index].block);
23     if (start_loc >= 0) {
24         tasks[task_index].start = start_loc;
25         tasks[task_index].pid = pid;
26         tasks[task_index].size = size;
27         tasks[task_index].block->pid = pid;
28         printf("Fork succeed. PID is %d . Block size is %d bits.\n", pid,
size);
29         return pid;
30     }
31     printf("Fork failed. Out of Memory. Trying to push into wait list.\n");
32     queue_push(size);
33     return -1;
34 }
35
36 int find_free_block_ff(int size, struct Block** block_p) {
37     struct Block* now = usr_space;
38     int cnt = 1;
39     while (1) {
40         if (now == NULL) return -1;
41         if (now->pid < 0 && now->size >= size) break;
42         now = now->next;
43         cnt++;
44     }
45     printf("Use %d times to find free block.\n", cnt);
46     *block_p = now;
47     if (now->size == size) return now->start;
48     struct Block* free_space = (struct Block*)malloc(sizeof(struct Block));
49     free_space->pid = -1;
50     free_space->last = now;
51     free_space->next = now->next;
52     free_space->size = now->size - size;
53     free_space->start = now->start + size;
54
55     now->pid = -2;
56     now->next = free_space;
57     now->size = size;
58     return now->start;
59 }
60
61 int find_free_block_bf(int size, struct Block** block_p) {
62     struct Block* now = usr_space, * best_block;
63     int best_size = 0x7fffffff;
64     int cnt = 1;
65     while (1) {
66         if (now == NULL) break;
67         if (now->pid < 0 && now->size >= size && best_size >= now->size) {
68             best_block = now;
69             best_size = now->size;
70         }
71         now = now->next;
72         cnt++;
73     }
74     if (best_size == 0x7fffffff) return -1;
75     printf("Use %d times to find free block.\n", cnt);
76     *block_p = best_block;

```

```

77     if (best_block->size == size) return best_block->start;
78     struct Block* free_space = (struct Block*)malloc(sizeof(struct Block));
79     free_space->pid = -1;
80     free_space->last = best_block;
81     free_space->next = best_block->next;
82     free_space->size = best_block->size - size;
83     free_space->start = best_block->start + size;
84
85     best_block->pid = -2;
86     best_block->next = free_space;
87     best_block->size = size;
88     return best_block->start;
89 }
90
91 int run(char* path) {
92     FILE* f = NULL;
93     f = fopen(path, "r+");
94     if (f == NULL) {
95         printf("Error: Invalid file path.\n");
96         return 0;
97     }
98     int result = 0;
99     while (!feof(f) && result != -1) result = get_command(f);
100    fclose(f);
101    return result;
102 }
103
104 int preprocess_task_size(int size) {
105     int lower = size & 0x3ff;
106     if (lower == 0) return size;
107     int upper = size - lower;
108     size = upper + 0x400;
109     return size;
110 }
111
112 void queue_push(int size) {
113     waitlist_begin->next = (struct Request*)malloc(sizeof(struct Request));
114     waitlist_begin = waitlist_begin->next;
115     waitlist_begin->next = NULL;
116     waitlist_begin->size = size;
117     waitlist_size++;
118 }
119
120 int queue_pop() {
121     if (!waitlist_size) {
122         printf("Wait list empty.\n");
123         return -1;
124     }
125     struct Request* tmp = waitlist_end;
126     waitlist_end = waitlist_end->next;
127     free(tmp);
128     waitlist_size--;
129     return waitlist_end->size;
130 }

```

## 2.3. 回收

```
1 void kill_process(int pid) {
2     struct Block* block = NULL;
3     for (int i = 0; i < MAX_NUM_TASKS; i++) {
4         if (tasks[i].pid != pid) continue;
5
6         block = tasks[i].block;
7         break;
8     }
9     if (block == NULL) {
10        printf("Error: PID dosen't exist.\n");
11        return;
12    }
13    if (block->last && block->last->pid < 0 && block->next && block->next->pid < 0) {
14        struct Block* free_block = block->last;
15        free_block->next = block->next->next;
16        free_block->size += block->size + block->next->size;
17        if (free_block->next) free_block->next->last = free_block;
18        free(block->next);
19        free(block);
20    }
21    else if (block->last && block->last->pid < 0) {
22        struct Block* free_block = block->last;
23        free_block->next = block->next;
24        free_block->size += block->size;
25        if (free_block->next) free_block->next->last = free_block;
26        free(block);
27    }
28    else if (block->next && block->next->pid < 0) {
29        struct Block* free_block = block;
30        free_block->size += block->next->size;
31        free_block->next = block->next->next;
32        free_block->pid = -1;
33        if (free_block->next) free_block->next->last = free_block;
34        free(block->next);
35    }
36    else {
37        block->pid = -1;
38    }
39
40    execute_wait_list();
41
42    return;
43 }
44
45 void execute_wait_list() {
46     int size = -1, loop_num = waitlist_size;
47     for (int i = 0; i < loop_num; i++) {
48         size = queue_pop();
49         fork_process(size);
50     }
```

## 2.4. 其他代码

```

1  int get_command(FILE* stream) {
2      printf("user> ");
3      char input_str[128] = { 0 };
4      fgets(input_str, 127, stream);
5      if (stream != stdin) printf("%s", input_str);
6
7      int split_index = 0;
8      for (int i = 0; i < 127; i++) {
9          if (input_str[i] == '\0') break;
10         if (input_str[i] == ' ') {
11             input_str[i] = '\0';
12             split_index = i;
13         }
14         if (input_str[i] == '\n') {
15             input_str[i] = '\0';
16             break;
17         }
18     }
19
20     char* command = input_str, * arguments = input_str + split_index + 1;
21
22     if (!strcmp(command, "exit")) {
23         return -1;
24     }
25     else if (strcmp(command, "fork") == 0) {
26         int input_size = atoi(arguments);
27         if (input_size <= 0) {
28             printf("Bad input size. Input number must larger than
29 ZERO.\n");
30             return 0;
31         }
32         else if (input_size > usr_size) {
33             printf("Bad input size. Input number must smaller than %d
34 bits.\n", usr_size);
35             return 0;
36         }
37         int pid = fork_process(input_size);
38         return 1;
39     }
40     else if (!strcmp(command, "kill")) {
41         int input_pid = atoi(arguments);
42         if (input_pid < 0) {
43             printf("Bad input PID. Input number must not smaller than
44 ZERO.\n");
45             return 0;
46         }
47         kill_process(input_pid);
48         return 1;
49     }
50 }

```

```

47     else if (!strcmp(input_str, "print")) {
48         print();
49         return 1;
50     }
51     else if (!strcmp(input_str, "run")) {
52         int result = run(arguments);
53         return result;
54     }
55     else if (input_str[0] == '\0');
56     else {
57         printf("Command not found.\n");
58         return 0;
59     }
60 }
61
62 void print() {
63     char str[2][13] = { {"          True"}, {"          False"} };
64     char* s;
65     struct Block* p = sys_space;
66     printf("+-----+
67     ++\n");
68     printf("|                               System Space
69     |\n");
70     printf("+-----+-----+-----+-----+
71     ++\n");
72     printf("|      Free      |      Begin      |      Size      |      End      |      PID
73     |\n");
74     printf("+-----+-----+-----+-----+
75     ++\n");
76     while (p != NULL) {
77         s = p->pid < 0 ? str[0] : str[1];
78         printf("|%s|%12d|%12d|%12d|%12d|\n", s, p->start, p->size, p->start
79         + p->size - 1, p->pid);
80         printf("+-----+-----+-----+-----+
81         +-----+\n");
82         p = p->next;
83     }
84     printf("+-----+
85     ++\n\n");
86
87     p = usr_space;
88     printf("+-----+
89     ++\n");
90     printf("|                               User Space
91     |\n");
92     printf("+-----+-----+-----+-----+
93     ++\n");
94     printf("|      Free      |      Begin      |      Size      |      End      |      PID
95     |\n");
96     printf("+-----+-----+-----+-----+
97     ++\n");
98     while (p != NULL) {
99         s = p->pid < 0 ? str[0] : str[1];
100        printf("|%s|%12d|%12d|%12d|%12d|\n", s, p->start, p->size, p->start
101        + p->size - 1, p->pid);

```

```

88     printf("+-----+-----+-----+-----+-----+
-----+\n");
89     p = p->next;
90 }
91     printf("+-----+-----+-----+-----+-----+
--+\\n");
92 }
93
94 void clean_stdin() {
95     int c;
96     do {
97         c = getchar();
98     } while (c != '\\n' && c != EOF);
99 }
100
101 int run(char* path) {
102     FILE* f = NULL;
103     f = fopen(path, "r+");
104     if (f == NULL) {
105         printf("Error: Invalid file path.\\n");
106         return 0;
107     }
108     int result = 0;
109     while (!feof(f) && result != -1) result = get_command(f);
110     fclose(f);
111     return result;
112 }

```

## 附录

```

1  #include <stdio.h>
2  #include <malloc.h>
3  #include <string.h>
4  #include <stdlib.h>
5
6  #define MAX_NUM_TASKS 128
7
8  struct Block {
9      int start;
10     int size;
11     int pid;
12     struct Block* next, * last;
13 };
14
15 struct Task {
16     int start;
17     int size;
18     int pid;
19     struct Block* block;
20 };
21
22 struct Request {

```

```

23     int size;
24     struct Request* next;
25 };
26
27
28 int preprocess_task_size(int);
29 int find_free_block_ff(int, struct Block**);
30 int find_free_block_bf(int, struct Block**);
31 void queue_push(int);
32 int queue_pop();
33 int get_command(FILE*);
34 int fork_process(int);
35 void kill_process(int);
36 int run(char*);
37 void init();
38 void clean_stdin();
39 void execute_wait_list();
40 void print();
41
42 int sys_size = 128 << 20, usr_size = 384 << 20;
43 struct Block* sys_space = NULL, * usr_space = NULL;
44 struct Task* tasks = NULL;
45 struct Request* waitlist_begin = NULL, * waitlist_end = NULL;
46 int waitlist_size = 0;
47 int last_pid = -1;
48 int match_method = -1;
49
50
51 int main() {
52     init();
53     while (get_command(stdin) >= 0);
54     return 0;
55 }
56
57 int get_command(FILE* stream) {
58     printf("user> ");
59     char input_str[128] = { 0 };
60     fgets(input_str, 127, stream);
61     if (stream != stdin) printf("%s", input_str);
62
63     int split_index = 0;
64     for (int i = 0; i < 127; i++) {
65         if (input_str[i] == '\0') break;
66         if (input_str[i] == ' ') {
67             input_str[i] = '\0';
68             split_index = i;
69         }
70         if (input_str[i] == '\n') {
71             input_str[i] = '\0';
72             break;
73         }
74     }
75
76     char* command = input_str, * arguments = input_str + split_index + 1;
77
78     if (!strcmp(command, "exit")) {

```

```

79         return -1;
80     }
81     else if (strcmp(command, "fork") == 0) {
82         int input_size = atoi(arguments);
83         if (input_size <= 0) {
84             printf("Bad input size. Input number must larger than
ZERO.\n");
85             return 0;
86         }
87         else if (input_size > usr_size) {
88             printf("Bad input size. Input number must smaller than %d
bits.\n", usr_size);
89             return 0;
90         }
91         int pid = fork_process(input_size);
92         return 1;
93     }
94     else if (!strcmp(command, "kill")) {
95         int input_pid = atoi(arguments);
96         if (input_pid < 0) {
97             printf("Bad input PID. Input number must not smaller than
ZERO.\n");
98             return 0;
99         }
100         kill_process(input_pid);
101         return 1;
102     }
103     else if (!strcmp(input_str, "print")) {
104         print();
105         return 1;
106     }
107     else if (!strcmp(input_str, "run")) {
108         int result = run(arguments);
109         return result;
110     }
111     else if (input_str[0] == '\0');
112     else {
113         printf("Command not found.\n");
114         return 0;
115     }
116 }
117
118 void kill_process(int pid) {
119     struct Block* block = NULL;
120     for (int i = 0; i < MAX_NUM_TASKS; i++) {
121         if (tasks[i].pid != pid) continue;
122
123         block = tasks[i].block;
124         break;
125     }
126     if (block == NULL) {
127         printf("Error: PID dosen't exist.\n");
128         return;
129     }
130     if (block->last && block->last->pid < 0 && block->next && block->next-
>pid < 0) {

```

```

131     struct Block* free_block = block->last;
132     free_block->next = block->next->next;
133     free_block->size += block->size + block->next->size;
134     if (free_block->next) free_block->next->last = free_block;
135     free(block->next);
136     free(block);
137 }
138 else if (block->last && block->last->pid < 0) {
139     struct Block* free_block = block->last;
140     free_block->next = block->next;
141     free_block->size += block->size;
142     if (free_block->next) free_block->next->last = free_block;
143     free(block);
144 }
145 else if (block->next && block->next->pid < 0) {
146     struct Block* free_block = block;
147     free_block->size += block->next->size;
148     free_block->next = block->next->next;
149     free_block->pid = -1;
150     if (free_block->next) free_block->next->last = free_block;
151     free(block->next);
152 }
153 else {
154     block->pid = -1;
155 }
156
157 execute_wait_list();
158
159 return;
160 }
161
162 void init() {
163     sys_space = (struct Block*)malloc(sizeof(struct Block));
164     sys_space->pid = -1;
165     sys_space->start = 0;
166     sys_space->size = sys_size;
167     sys_space->next = NULL;
168     sys_space->last = NULL;
169
170     usr_space = (struct Block*)malloc(sizeof(struct Block));
171     usr_space->pid = -1;
172     usr_space->start = sys_size;
173     usr_space->size = usr_size;
174     usr_space->next = NULL;
175     usr_space->last = NULL;
176
177     tasks = (struct Task*)malloc(MAX_NUM_TASKS * sizeof(struct Task));
178     for (int i = 0; i < MAX_NUM_TASKS; i++) tasks[i].pid = -1;
179
180 ask_match_method:
181     printf("Choose a matching method:\n 1. First Fit\n 2. Best Fit\n");
182     printf("choose> ");
183     scanf("%d", &match_method);
184     clean_stdin();
185     if (match_method != 1 && match_method != 2) {
186         printf("Bad input. Choose again.\n");

```

```

187         goto ask_match_method;
188     }
189
190     waitlist_begin = (struct Request*)malloc(sizeof(struct Request));
191     waitlist_end = waitlist_begin;
192     waitlist_begin->next = NULL;
193     waitlist_begin->size = -1;
194 }
195
196 int fork_process(int size) {
197     int pid;
198     int task_index = -1;
199     repeat:
200         last_pid++;
201         if (last_pid < 0) last_pid = 0;
202         pid = last_pid;
203         for (int i = 0; i < MAX_NUM_TASKS; i++) {
204             if (tasks[i].pid >= 0 && tasks[i].pid == pid) goto repeat;
205         }
206         for (int i = 0; i < MAX_NUM_TASKS; i++) {
207             if (tasks[i].pid >= 0) continue;
208             task_index = i;
209             break;
210         }
211         if (task_index == -1) return -1;
212
213         size = preprocess_task_size(size);
214         tasks[task_index].size = size;
215         int start_loc = -1;
216         if (match_method == 1) start_loc = find_free_block_ff(size,
217 &tasks[task_index].block);
218         else start_loc = find_free_block_bf(size, &tasks[task_index].block);
219         if (start_loc >= 0) {
220             tasks[task_index].start = start_loc;
221             tasks[task_index].pid = pid;
222             tasks[task_index].size = size;
223             tasks[task_index].block->pid = pid;
224             printf("Fork succeed. PID is %d . Block size is %d bits.\n", pid,
225 size);
226             return pid;
227         }
228         printf("Fork failed. Out of Memory. Trying to push into wait list.\n");
229         queue_push(size);
230         return -1;
231     }
232
233 int find_free_block_ff(int size, struct Block** block_p) {
234     struct Block* now = usr_space;
235     int cnt = 1;
236     while (1) {
237         if (now == NULL) return -1;
238         if (now->pid < 0 && now->size >= size) break;
239         now = now->next;
240         cnt++;
241     }
242     printf("Use %d times to find free block.\n", cnt);

```

```

241     *block_p = now;
242     if (now->size == size) return now->start;
243     struct Block* free_space = (struct Block*)malloc(sizeof(struct Block));
244     free_space->pid = -1;
245     free_space->last = now;
246     free_space->next = now->next;
247     free_space->size = now->size - size;
248     free_space->start = now->start + size;
249
250     now->pid = -2;
251     now->next = free_space;
252     now->size = size;
253     return now->start;
254 }
255
256 int find_free_block_bf(int size, struct Block** block_p) {
257     struct Block* now = usr_space, * best_block;
258     int best_size = 0x7fffffff;
259     int cnt = 1;
260     while (1) {
261         if (now == NULL) break;
262         if (now->pid < 0 && now->size >= size && best_size >= now->size) {
263             best_block = now;
264             best_size = now->size;
265         }
266         now = now->next;
267         cnt++;
268     }
269     if (best_size == 0x7fffffff) return -1;
270     printf("Use %d times to find free block.\n", cnt);
271     *block_p = best_block;
272     if (best_block->size == size) return best_block->start;
273     struct Block* free_space = (struct Block*)malloc(sizeof(struct Block));
274     free_space->pid = -1;
275     free_space->last = best_block;
276     free_space->next = best_block->next;
277     free_space->size = best_block->size - size;
278     free_space->start = best_block->start + size;
279
280     best_block->pid = -2;
281     best_block->next = free_space;
282     best_block->size = size;
283     return best_block->start;
284 }
285
286 int run(char* path) {
287     FILE* f = NULL;
288     f = fopen(path, "r+");
289     if (f == NULL) {
290         printf("Error: Invalid file path.\n");
291         return 0;
292     }
293     int result = 0;
294     while (!feof(f) && result != -1) result = get_command(f);
295     fclose(f);
296     return result;

```

```

297 }
298
299 int preprocess_task_size(int size) {
300     int lower = size & 0x3ff;
301     if (lower == 0) return size;
302     int upper = size - lower;
303     size = upper + 0x400;
304     return size;
305 }
306
307 void queue_push(int size) {
308     waitlist_begin->next = (struct Request*)malloc(sizeof(struct Request));
309     waitlist_begin = waitlist_begin->next;
310     waitlist_begin->next = NULL;
311     waitlist_begin->size = size;
312     waitlist_size++;
313 }
314
315 int queue_pop() {
316     if (!waitlist_size) {
317         printf("Wait list empty.\n");
318         return -1;
319     }
320     struct Request* tmp = waitlist_end;
321     waitlist_end = waitlist_end->next;
322     free(tmp);
323     waitlist_size--;
324     return waitlist_end->size;
325 }
326
327 void clean_stdin() {
328     int c;
329     do {
330         c = getchar();
331     } while (c != '\n' && c != EOF);
332 }
333
334 void execute_wait_list() {
335     int size = -1, loop_num = waitlist_size;
336     for (int i = 0; i < loop_num; i++) {
337         size = queue_pop();
338         fork_process(size);
339     }
340 }
341
342 void print() {
343     char str[2][13] = { {"      True"}, {"      False"} };
344     char* s;
345     struct Block* p = sys_space;
346     printf("+-----+
347     --+\n");
348     printf("|
349     System Space
350     |\n");
351     printf("+-----+-----+-----+-----+-----+
352     --+\n");

```

```

349     printf("|    Free    |    Begin    |    Size    |    End    |    PID\n");
350     printf("+-----+-----+-----+-----+-----\n");
351     while (p != NULL) {
352         s = p->pid < 0 ? str[0] : str[1];
353         printf("|%s|%12d|%12d|%12d|%12d|\n", s, p->start, p->size, p->start
+ p->size - 1, p->pid);
354         printf("+-----+-----+-----+-----+-----\n");
355         p = p->next;
356     }
357     printf("+-----\n");
358
359     p = usr_space;
360     printf("+-----\n");
361     printf("|                                User Space\n");
362     printf("+-----+-----+-----+-----+-----\n");
363     printf("|    Free    |    Begin    |    Size    |    End    |    PID\n");
364     printf("+-----+-----+-----+-----+-----\n");
365     while (p != NULL) {
366         s = p->pid < 0 ? str[0] : str[1];
367         printf("|%s|%12d|%12d|%12d|%12d|\n", s, p->start, p->size, p->start
+ p->size - 1, p->pid);
368         printf("+-----+-----+-----+-----+-----\n");
369         p = p->next;
370     }
371     printf("+-----\n");
372 }

```

