Project 6: Banker's Algorithm

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 Banker's Algorithm

1.1 Requirements

For this project, we are required to write a program that implements the banker's algorithm discussed in Section 8.6.3 in textbook. Customers request and release resources from the bank. The banker will grant a request only if it leaves the system in a safe state. A request that leaves the system in an unsafe state will be denied. Although the code examples that describe this project are illustrated in C, you may also develop a solution using Java.

The banker will consider requests from n customers for m resources types, as outlined in Section 8.6.3. The banker will keep track of the resources using `available`, `allocation`, `maximum` and `need` arrays. The banker will grant a request if it satisfies the safety algorithm outlined in Section 8.6.3.1 in textbook. If a request does not leave the system in a safe state, the banker will deny it. Function prototypes for requesting and releasing resources are as follows.

```
1 int request_resources(int customer_id, int request[]);
2 void release_resources(int customer_id, int request[]);
```

The `request_resources()` function should return 0 if successful and -1 if unsuccessful.
We need to implement the following functions.

- Use instruction `*` to output the values of the different tracking arrays.
- Use instruction `RQ ...` to request the resources;
- Use instruction `RL ...` to release the resources;
- Use instruction `exit` to exit the banker program.

1.2 Methods

Here are some specific methods of the program.

- We use the Banker's Algorithm to implement the full program. We use `available`, `allocation`, `maximum` and `need` arrays to keep track of the resources.
- For every request, we first suppose that we grant the request. Then we will use the banker-algorithm based safe-unsafe checking algorithm to check whether the state is safe. If the state is safe, we will grant the request and update the arrays; if the state is unsafe, we will deny the request. We also need to check some special cases of the input to prevent invalid inputs.
- For every release operation, we will release the resources immediately, unless the release instruction is invalid.
- For every printing operation (`*`), we will print the arrays in the screen immediately.
- For exit operation, we will exit the program.
- You need to put the maximum data in the `maximum.txt`, and put available data in the arguments to start the program. If the data is invalid, the program will terminate automatically.

1.3 Implementation

Here is the specific implementation of the program (`banker.c`).

```
1  # include <stdio.h>
2  # include <stdlib.h>
3  # include <string.h>
4  # include <unistd.h>
5
6  # define MAX_LINE_BUF 500
7  # define TRUE 1
8
9  // resource number
10 int resource_num;
11
12 // customer number
13 int customer_num;
14
15 // the available amount of each resource
16 int * available;
17 // the maximum demand of each customer
18 int ** maximum;
19 // the amount currently allocated to each other
20 int ** allocation;
21 // the remaining need of each customer
22 int ** need;
23 // current capacity
24 int cur_capacity;
25
26
27 void initialize(int argc, char *argv[]);
28 void print_state(int op);
29 void destroy(void);
30 int parse(char *buf, char *op, int *arg, int * argn);
31 void upd_need(int ** need, int ** maximum, int ** allocation);
32 int check_initial_safe(void);
33 int request_resources(int customer_id, int request[]);
34 void release_resources(int customer_id, int release[]);
35
36
37 int main(int argc, char *argv[]) {
38   initialize(argc, argv);
39
40   print_state(0);
41
42   int res = check_initial_safe();
43   if (res) {
44     fprintf(stdout, "[Err] Initial state is unsafe! Process will end.\n");
45     exit(1);
46   }
47
```

```
48    char buf[MAX_LINE_BUF], op[MAX_LINE_BUF];
49    int *arg = (int *) malloc (sizeof(int) * (1 + resource_num)), argn;
50
51    while(TRUE) {
52      fprintf(stdout, "Banker >> ");
53      fgets(buf, MAX_LINE_BUF, stdin);
54
55      int err = parse(buf, op, arg, &argn);
56      if (err) {
57        fprintf(stdout, "[Err] Invalid instruction. This instruction will be ignored
              .\n");
58        continue;
59      }
60
61      if (strcmp(op, "exit") == 0 && argn == 0) break;
62      else if (strcmp(op, "*") == 0 && argn == 0) print_state(2);
63      else if (strcmp(op, "RQ") == 0 && argn == resource_num + 1) {
64        if (request_resources(arg[0], arg + 1) == -1) fprintf(stdout, "[Log] Request
               command denied.\n");
65        else fprintf(stdout, "[Log] Request command accepted.\n");
66      }
67      else if (strcmp(op, "RL") == 0 && argn == resource_num + 1) release_resources(
           arg[0], arg + 1);
68      else {
69        fprintf(stdout, "[Err] Invalid instruction. This instruction will be ignored
              .\n");
70        continue;
71      }
72    }
73
74    free(arg);
75
76    destroy();
77    return 0;
78  }
79
80
81  // Initialize the arrays according to the input
82  void initialize(int argc, char *argv[]) {
83    // read the initial available resource from the arguments
84    resource_num = argc - 1;
85    if (resource_num == 0) {
86      fprintf(stderr, "Error: no resource!\n");
87      exit(1);
88    }
89
90    available = (int *) malloc (sizeof(int) * resource_num);
91
92    for (int i = 1; i < argc; ++ i)
93      available[i - 1] = atoi(argv[i]);
```

```
94
95    // initial array
96    customer_num = 0;
97    cur_capacity = 100;
98    maximum = (int **) malloc (sizeof(int *) * cur_capacity);
99
100   // read the maximum demand data from file ‘‘maximum.txt’’
101   FILE *fp = fopen("maximum.txt", "r");
102   static int dat;
103   while(~fscanf(fp, "%d", &dat)) {
104     // if already full, then double the array.
105     if (customer_num == cur_capacity) {
106       int ** tem;
107       tem = (int **) malloc (sizeof(int *) * cur_capacity * 2);
108       for (int i = 0; i < cur_capacity; ++ i) {
109         tem[i] = (int *) malloc (sizeof(int) * resource_num);
110         for (int j = 0; j < resource_num; ++ j)
111           tem[i][j] = maximum[i][j];
112         free(maximum[i]);
113       }
114       free(maximum);
115       maximum = tem;
116       cur_capacity <<= 1;
117     }
118     // read the data
119     maximum[customer_num] = (int *) malloc (sizeof(int) * resource_num);
120     maximum[customer_num][0] = dat;
121     for (int i = 1; i < resource_num; ++ i) {
122       fscanf(fp, ",%d", &dat);
123       maximum[customer_num][i] = dat;
124     }
125     customer_num ++;
126   }
127   fclose(fp);
128
129   // allocate the array
130   allocation = (int **) malloc (sizeof(int *) * cur_capacity);
131   for (int i = 0; i < customer_num; ++ i)
132     allocation[i] = (int *) malloc (sizeof(int) * resource_num);
133   need = (int **) malloc (sizeof(int *) * cur_capacity);
134   for (int i = 0; i < customer_num; ++ i)
135     need[i] = (int *) malloc (sizeof(int) * resource_num);
136
137   // initialize the array
138   for (int i = 0; i < customer_num; ++ i)
139     for (int j = 0; j < resource_num; ++ j)
140       allocation[i][j] = 0;
141
142   upd_need(need, maximum, allocation);
143 }
```

```
144
145 // Print the current state
146 // op = 0: just output available & maximum; op = 1: also output allocation; op =
       2: also output allocation & need
147 void print_state(int op) {
148   fprintf(stdout, "[Log] Current State: \n");
149   fprintf(stdout, " Customer Number = %d\n Resource Number = %d\n", customer_num,
        resource_num);
150   fprintf(stdout, " Available = [");
151   for (int i = 0; i < resource_num; ++ i)
152     fprintf(stdout, "%d%c%c", available[i], (i == resource_num - 1) ? ']' : ',', (
          i == resource_num - 1) ? '\n' : ' ');
153
154   fprintf(stdout, " Maximum = [\n");
155   for (int i = 0; i < customer_num; ++ i) {
156     fprintf(stdout, "  [");
157     for (int j = 0; j < resource_num; ++ j)
158       fprintf(stdout, "%d%c%c", maximum[i][j], (j == resource_num - 1) ? ']' :
            ',', (j == resource_num - 1) ? '\n' : ' ');
159   }
160   fprintf(stdout, " ]\n");
161
162   if (op >= 1) {
163     fprintf(stdout, " Allocation = [\n");
164     for (int i = 0; i < customer_num; ++ i) {
165       fprintf(stdout, "  [");
166       for (int j = 0; j < resource_num; ++ j)
167         fprintf(stdout, "%d%c%c", allocation[i][j], (j == resource_num - 1) ? ']'
              : ',', (j == resource_num - 1) ? '\n' : ' ');
168     }
169     fprintf(stdout, " ]\n");
170   }
171
172   if (op >= 2) {
173     fprintf(stdout, " Need = [\n");
174     for (int i = 0; i < customer_num; ++ i) {
175       fprintf(stdout, "  [");
176       for (int j = 0; j < resource_num; ++ j)
177         fprintf(stdout, "%d%c%c", need[i][j], (j == resource_num - 1) ? ']' : ',',
              (j == resource_num - 1) ? '\n' : ' ');
178     }
179     fprintf(stdout, " ]\n");
180   }
181 }
182
183 // De-allocate the array
184 void destroy(void) {
185   free(available);
186   for (int i = 0; i < customer_num; ++ i) {
187     free(maximum[i]);
```

```
188     free(allocation[i]);
189     free(need[i]);
190   }
191   free(maximum);
192   free(allocation);
193   free(need);
194 }
195 
196 // Parse the buffer
197 int parse(char * buf, char * op, int * arg, int * argn) {
198   int hv, tmp, opcnt = 0;
199   hv = 0; tmp = 0;
200   (*argn) = -1;
201   for (int i = 0; buf[i]; ++ i) {
202     if (buf[i] == ' ' || buf[i] == '\t' || buf[i] == '\n') {
203       if (!hv) continue;
204       hv = 0;
205       if (*argn != -1) {
206         if (*argn == resource_num + 1) return 1;
207         arg[*argn] = tmp;
208         tmp = 0;
209       }
210       (*argn) ++;
211     } else {
212       hv = 1;
213       if(*argn == -1) op[opcnt ++] = buf[i];
214       else {
215         if (buf[i] >= '0' && buf[i] <= '9') tmp = tmp * 10 + buf[i] - '0';
216         else return 1;
217       }
218     }
219   }
220   op[opcnt] = 0;
221   if(hv) {
222     if (*argn != -1) {
223       if (*argn == resource_num + 1) return 1;
224       arg[*argn] = tmp;
225       tmp = 0;
226     }
227     (*argn) ++;
228   }
229   return 0;
230 }
231 
232 
233 // Update the need matrix
234 void upd_need(int ** need, int ** maximum, int ** allocation) {
235   for (int i = 0; i < customer_num; ++ i)
236     for (int j = 0; j < resource_num; ++ j)
237       need[i][j] = maximum[i][j] - allocation[i][j];
```

```
238 }
239
240 // Check whether the initial state is safe
241 int check_initial_safe(void) {
242   // |-- if maximum > initial avaible, unsafe.
243   for (int i = 0; i < customer_num; ++ i)
244     for (int j = 0; j < resource_num; ++ j)
245       if(maximum[i][j] > available[j]) return 1;
246   // |-- otherwise, safe.
247   return 0;
248 }
249
250 // Request the resources
251 int request_resources(int customer_id, int request[]) {
252   int * available_t, * served;
253
254   // |-- Check special cases.
255   for (int i = 0; i < resource_num; ++ i)
256     if (request[i] > need[customer_id][i]) {
257       fprintf(stdout, "[Err] The request should not be greater than need. This
             request will be ignored.");
258       return -1;
259     }
260   for (int i = 0; i < resource_num; ++ i)
261     if (request[i] > available[i]) {
262       fprintf(stdout, "[Log] Request CANNOT be granted, because there is not
             enough available resources.\n");
263       return -1;
264     }
265
266   // |-- Suppose we grant the request, then check the state.
267   available_t = (int *) malloc (sizeof(int) * resource_num);
268   served = (int *) malloc (sizeof(int) * customer_num);
269   for (int i = 0; i < customer_num; ++ i)
270     served[i] = 0;
271   for (int i = 0; i < resource_num; ++ i) {
272     available_t[i] = available[i] - request[i];
273     allocation[customer_id][i] += request[i];
274   }
275   upd_need(need, maximum, allocation);
276
277   // |-- Implement the situation.
278   int res = 1;
279   for (int step = 0; step < customer_num; ++ step) {
280     // |---- Find a unserved, feasible customer.
281     int pos = -1;
282     for (int i = 0; i < customer_num; ++ i) {
283       if (served[i]) continue;
284       int flag = 1;
285       for (int j = 0; j < resource_num; ++ j)
```

```
286         if (need[i][j] > available_t[j]) {
287           flag = 0;
288           break;
289         }
290       if (flag) {
291         pos = i;
292         break;
293       }
294     }
295     // |---- Not found, then unsafe.
296     if(pos == -1) {
297       res = 0;
298       break;
299     }
300     // |---- Serve the customer.
301     served[pos] = 1;
302     for (int i = 0; i < resource_num; ++ i)
303       available_t[i] += allocation[pos][i];
304   }
305
306   // |-- res = 1, then safe; res = 0, then unsafe.
307   if (res) {
308     fprintf(stdout, "[Log] Request is granted.\n");
309     for (int i = 0; i < resource_num; ++ i)
310       available[i] -= request[i];
311     free(available_t);
312     free(served);
313     return 0;
314   } else {
315     fprintf(stdout, "[Log] Request CANNOT be granted, or the system will become
          unsafe.\n");
316     for (int i = 0; i < resource_num; ++ i)
317       allocation[customer_id][i] -= request[i];
318     upd_need(need, maximum, allocation);
319     free(available_t);
320     free(served);
321     return -1;
322   }
323 }
324
325 // Release the resources
326 void release_resources(int customer_id, int release[]) {
327   // |-- Check special cases.
328   for (int i = 0; i < resource_num; ++ i)
329     if (release[i] > allocation[customer_id][i]) {
330       fprintf(stdout, "[Err] The release should not be greater than allocation.
            This release will be ignored.\n");
331       return ;
332     }
333   // |-- Release the resources.
```

```
  for (int i = 0; i < resource_num; ++ i) {
    available[i] += release[i];
    allocation[customer_id][i] -= release[i];
  }
  upd_need(need, maximum, allocation);
  fprintf(stdout, "[Log] The resources are released.\n");
  return ;
}
```

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, where `...` stands for the `available` arguments. Remember to put the maximum data in the `maximum.txt` file.

```
make
./banker ...
```

Here is a specific testing examples.

```
make
./banker 10 6 9 7
RQ 0 6 4 7 3
*
RQ 1 5 2 2 2
RQ 1 4 2 2 2
RQ 4 0 0 0 2
RL 0 1 1 1 1
RQ 4 1 1 1 1
*
exit
```

The data in the `maximum.txt` is displayed as follows.

```
6,4,7,3
4,2,3,2
2,5,3,3
6,3,3,2
5,6,7,5
```

Here are the execution results of the program (Fig. 1, Fig. 2 and Fig. 3).

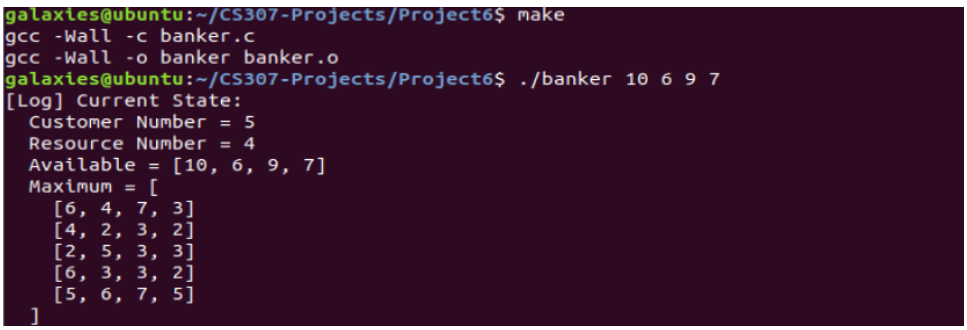

Figure 1: The execution result of the program (1)

```
Banker >> RQ 0 6 4 7 3
[Log] Request is granted.
[Log] Request command accepted.
Banker >> *
[Log] Current State:
  Customer Number = 5
  Resource Number = 4
  Available = [4, 2, 2, 4]
  Maximum = [
    [6, 4, 7, 3]
    [4, 2, 3, 2]
    [2, 5, 3, 3]
    [6, 3, 3, 2]
    [5, 6, 7, 5]
  ]
  Allocation = [
    [6, 4, 7, 3]
    [0, 0, 0, 0]
    [0, 0, 0, 0]
    [0, 0, 0, 0]
    [0, 0, 0, 0]
  ]
  Need = [
    [0, 0, 0, 0]
    [4, 2, 3, 2]
    [2, 5, 3, 3]
    [6, 3, 3, 2]
    [5, 6, 7, 5]
  ]
Banker >> RQ 1 5 2 2 2
[Err] The request should not be greater than need. This request will be ignored.
[Log] Request command denied.
Banker >> RQ 1 4 2 2 2
[Log] Request is granted.
[Log] Request command accepted.
Banker >> RQ 4 0 0 0 2
[Log] Request is granted.
[Log] Request command accepted.
Banker >> RL 0 1 1 1 1
[Log] The resources are released.
Banker >> RQ 4 1 1 1 1
[Log] Request CANNOT be granted, or the system will become unsafe.
[Log] Request command denied.
```

Figure 2: The execution result of the program (2)

```
Banker >> *
[Log] Current State:
  Customer Number = 5
  Resource Number = 4
  Available = [1, 1, 1, 1]
  Maximum = [
    [6, 4, 7, 3]
    [4, 2, 3, 2]
    [2, 5, 3, 3]
    [6, 3, 3, 2]
    [5, 6, 7, 5]
  ]
  Allocation = [
    [5, 3, 6, 2]
    [4, 2, 2, 2]
    [0, 0, 0, 0]
    [0, 0, 0, 0]
    [0, 0, 0, 2]
  ]
  Need = [
    [1, 1, 1, 1]
    [0, 0, 1, 0]
    [2, 5, 3, 3]
    [6, 3, 3, 2]
    [5, 6, 7, 3]
  ]
Banker >> exit
galaxies@ubuntu:~/CS307-Projects/Project6$
```

Figure 3: The execution result of the program (3)

2 Personal Thoughts

The project helps me understand the banker's algorithm better, and I also gain some knowledge about the banker-algorithm-based safe-unsafe checking algorithm. The project also trains our coding skills since we also need to implement parsing process except the algorithm. The implementation of the algorithm is a little bit complicated and 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 `Project6` folder.