Project 7

Name: 刘　涵　之
ID : 519021910102

Project 7: 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, It 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.

Design: My design for this task is:

- The linked list is used to present the memory state. I use char **process** to distinguish the free memory and the allocated memory.
- For `RL` command, **liner_compact()** function is used to combine the free memory into one memory block in the linked list.
- For `RQ` command, first-fit, best-fit and worst-fit algorithm are implemented according to the textbook. The basic idea is to traverse the linked list to find the target free memory.
- For `STAT` command, **output()** function is used to print the current memory state.
- For `C` command, **compact()** function is used to find all free memory node, delete them from the linked list and add them up to one free memory node.

The implementation of the contiguous memory allocator (`allocator.c`) is shown as follows.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>


struct memory {
    char *process;
    int space;
    struct memory * next;
};

struct memory *head;

void initialize(int space) {
    head = (struct memory *) malloc (sizeof(struct memory));
    head -> process = (char *) malloc (sizeof(char) * 20);
    strcpy(head -> process, "Unused");
    head -> space = space;
    head -> next = NULL;
    return;
}

void output() {
    struct memory *temp = head;
    int address = 0;
    while (temp != NULL) {
        fprintf(stdout, "Addresses [%d:%d] %s\n", address, address + temp -> space - 1,
            temp -> process);
        address += temp -> space;
        temp = temp -> next;
    }
    return;
}

void liner_compact() {
    struct memory *temp = head;
    while (temp -> next != NULL) {
        struct memory *n = temp -> next;
        if ((strcmp(temp -> process, "Unused") == 0) && (strcmp(n -> process, "Unused")
            == 0)) {
            temp -> space += n -> space;
            temp -> next = n -> next;
            free(n -> process);
            free(n);
        }else{
            temp = temp -> next;
        }
    }
}
```

```
49
50  void compact() {
51      struct memory *temp = head -> next, *last = head;
52      int free_sapce = 0;
53      if (temp == NULL) return;
54      if (strcmp(head -> process, "Unused") == 0) {
55          struct memory *n = head;
56          free_sapce += head -> space;
57          head = head -> next;
58          free(n -> process);
59          free(n);
60      }
61      while (temp != NULL) {
62          if (strcmp(temp -> process, "Unused") == 0) {
63              free_sapce += temp -> space;
64              last -> next = temp -> next;
65              free(temp -> process);
66              free(temp);
67              temp = last -> next;
68          } else {
69              last = temp;
70              temp = temp -> next;
71          }
72      }
73
74      last -> next = (struct memory *) malloc (sizeof(struct memory));
75      last = last -> next;
76      last -> process = (char *) malloc (sizeof(char) * 20);
77      strcpy(last -> process, "Unused");
78      last -> space = free_sapce;
79      last -> next = NULL;
80      return;
81  }
82
83  int main(int argc, char *argv[]) {
84          if (argc != 2) {
85          fprintf(stdout, "[Error] Wrong input. \n");
86          exit(1);
87      }
88
89      initialize(atoi(argv[1]));
90
91      while (1) {
92          char s1[100];
93          fprintf(stdout, "allocator>>");
94          fscanf(stdin, "%s", s1);
95          if (strcmp(s1, "EXIT") == 0) {
96              break;
97          } else
98          if (strcmp(s1, "STAT") == 0) {
99              output();
```

```
100         } else
101         if (strcmp(s1, "RQ") == 0) {
102             char process_name[100], mode;
103             int process_size;
104             fscanf(stdin, "%s %d %c", process_name, &process_size, &mode);
105             if (mode == 'F') {
106                 struct memory *temp = head;
107                 while (temp != NULL) {
108                     if (strcmp(temp -> process, "Unused") == 0) {
109                         if (temp -> space > process_size) {
110                             struct memory *new = (struct memory *) malloc (sizeof(struct
                                memory));
111                             new -> process = (char *) malloc (sizeof(char) * 20);
112                             strcpy(new -> process, "Unused");
113                             new -> space = temp -> space - process_size;
114                             new -> next = temp -> next;
115 
116                             strcpy(temp -> process, process_name);
117                             temp -> space = process_size;
118                             temp -> next = new;
119 
120                             break;
121                         } else if (temp -> space == process_size) {
122                             strcpy(temp -> process, process_name);
123                             temp -> space = process_size;
124 
125                             break;
126                         }
127                     }
128                     temp = temp -> next;
129                 }
130                 if (temp == NULL) {
131                     fprintf(stdout, "[Error] Out of Memory, you may need to compact!! \n"
                            );
132                 }
133             } else if (mode == 'B') {
134                 struct memory *temp = head, *target = NULL;
135                 while (temp != NULL) {
136                     if (strcmp(temp -> process, "Unused") == 0) {
137                         if (temp -> space >= process_size) {
138                             if (target == NULL) {
139                                 target = temp;
140                             } else if (target -> space > temp -> space) {
141                                 target = temp;
142                             }
143                         }
144                     }
145                     temp = temp -> next;
146                 }
147                 if (target == NULL) {
148                     fprintf(stdout, "[Error] Out of Memory, you may need to compact!! \n"
```

```
                    );
149             } else {
150                 if (target -> space > process_size) {
151                     struct memory *new = (struct memory *) malloc (sizeof(struct
                        memory));
152                     new -> process = (char *) malloc (sizeof(char) * 20);
153                     strcpy(new -> process, "Unused");
154                     new -> space = target -> space - process_size;
155                     new -> next = target -> next;
156 
157                     strcpy(target -> process, process_name);
158                     target -> space = process_size;
159                     target -> next = new;
160 
161                     } else if (target -> space == process_size) {
162                         strcpy(target -> process, process_name);
163                         target -> space = process_size;
164                     }
165                 }
166             } else if (mode == 'W') {
167                 struct memory *temp = head, *target = NULL;
168                 while (temp != NULL) {
169                     if (strcmp(temp -> process, "Unused") == 0) {
170                         if (temp -> space >= process_size) {
171                             if (target == NULL) {
172                                 target = temp;
173                             } else if (target -> space < temp -> space) {
174                                 target = temp;
175                             }
176                         }
177                     }
178                     temp = temp -> next;
179                 }
180                 if (target == NULL) {
181                     fprintf(stdout, "[Error] Out of Memory, you may need to compact!! \n"
                        );
182                 } else {
183                     if (target -> space > process_size) {
184                         struct memory *new = (struct memory *) malloc (sizeof(struct
                            memory));
185                         new -> process = (char *) malloc (sizeof(char) * 20);
186                         strcpy(new -> process, "Unused");
187                         new -> space = target -> space - process_size;
188                         new -> next = target -> next;
189 
190                         strcpy(target -> process, process_name);
191                         target -> space = process_size;
192                         target -> next = new;
193 
194                         } else if (target -> space == process_size) {
195                             strcpy(target -> process, process_name);
```

```
                            target -> space = process_size;
                        }
                }
            } else {
                fprintf(stdout, "[Error] Unexpected mode! \n");
            }
        } else
        if (strcmp(s1, "RL") == 0) {
            char process_name[100];
            fscanf(stdin, "%s", process_name);

            struct memory *temp = head;
            while (temp != NULL) {
                if (strcmp(temp -> process, process_name) == 0) {
                    strcpy(temp -> process, "Unused");
                    liner_compact();
                    break;
                }
                temp = temp -> next;
            }
            if (temp == NULL) {
                fprintf(stdout, "[Error] Unexpected Process name! \n");
            }
        } else
        if (strcmp(s1, "C") == 0) {
            compact();
        } else {
            fprintf(stdout, "[Error] Unexpected input! \n");
        }
    }

    return 0;
}
```

Makefile for Contiguous Memory Allocation is shown as follow.

```
CC=gcc
CFLAGS=-Wall

all: allocator.o
        $(CC) $(CFLAGS) -o allocator allocator.o

allocator.o: allocator.c
        $(CC) $(CFLAGS) -c allocator.c

clean:
        rm -rf *.o
        rm -rf allocator
```

The execution result is shown as follow.

```
misaka@MS-BVZPMBEQIPCD:/mnt/c/Projects/OS_Project/Project7/project7$ make all
gcc -Wall -c allocator.c
gcc -Wall -o allocator allocator.o
.misaka@MS-BVZPMBEQIPCD:/mnt/c/Projects/OS_Project/Project7/project7$ ./allocator 100000
allocator>>RQ P1 10000 F
allocator>>RQ P2 9999 F
allocator>>RQ P3 10000 F
allocator>>RQ P4 10001 F
allocator>>RQ P5 10000 F
allocator>>RQ P6 10000 F
allocator>>RQ P7 23333 F
allocator>>STAT
Addresses [0:9999] P1
Addresses [10000:19998] P2
Addresses [19999:29998] P3
Addresses [29999:39999] P4
Addresses [40000:49999] P5
Addresses [50000:59999] P6
Addresses [60000:83332] P7
Addresses [83333:99999] Unused
```

图 1: Contiguous Memory Allocation

```
allocator>>RL P2
allocator>>RL P4
allocator>>RL P6
allocator>>STAT
Addresses [0:9999] P1
Addresses [10000:19998] Unused
Addresses [19999:29998] P3
Addresses [29999:39999] Unused
Addresses [40000:49999] P5
Addresses [50000:59999] Unused
Addresses [60000:83332] P7
Addresses [83333:99999] Unused
allocator>>RQ P8 10000 B
allocator>>RQ P9 9999 W
allocator>>RQ P10 9998 F
allocator>>STAT
Addresses [0:9999] P1
Addresses [10000:19997] P10
Addresses [19998:19998] Unused
Addresses [19999:29998] P3
Addresses [29999:39999] Unused
Addresses [40000:49999] P5
Addresses [50000:59999] P8
Addresses [60000:83332] P7
Addresses [83333:93331] P9
Addresses [93332:99999] Unused
```

图 2: Contiguous Memory Allocation

```
allocator>>C
allocator>>STAT
Addresses [0:9999] P1
Addresses [10000:19997] P10
Addresses [19998:29997] P3
Addresses [29998:39997] P5
Addresses [39998:49997] P8
Addresses [49998:73330] P7
Addresses [73331:83329] P9
Addresses [83330:99999] Unused
allocator>>RL P9
allocator>>RL P8
allocator>>RL P7
allocator>>RL P5
allocator>>RL P3
allocator>>RL P1
allocator>>RL P10
allocator>>STAT
Addresses [0:99999] Unused
allocator>>EXIT
```

图 3: Contiguous Memory Allocation