# CS356 Project 7

Name : HaotianXue

Student ID : 518021910506

Email : xavihart@sjtu.edu.cn

**CS356 Project 7**

- Introduction
- General Ideas for Implementation
- User Command
- Detailed Implementation
- Evaluation
- Conclusion

## Introduction

This is the report for project7 for CS356, also for COS book chapter9.

The project encouraged us to implement  a program to simulate the arrangement for contiguous memory, including managing request from process for memory, releasing of memory space, compacting to remove holes and presentation of status of current memory states.

When considering the request for memory space, we have strategies like Best First, Best First and Worst First.

## General Ideas for Implementation

We use linked list to represent a contiguous space in the memory which is a hole or is occupied by the same process. For each node in the linked, we have properties like `used`, `start`, `end`, `name`. `used` is used to represent if the space is a hole, the next two is used to determine the range for the space. And `last` is used to save the name of the process:

```
typedef struct Block{
    int used;
    int start;
    int end;
    char name[5];
}block;

typedef struct Node{
   block *hole;
   struct Node* next;
   struct Node* pre;
}node;
```

We have many functions in this implementation:

```
/* allocate memory for certain process in certain memory
blocks*/
void insert_process(node* , int,char*);
/* traverse the whole memory and output*/
void traverse();
/* this function will generate a new array with new head
and tail based on the original mem*/
void compact();
/*free all the dynamic allocated memory*/
void free_all();
/*init linked list*/
void init();
/*release process into unused blocks*/
void release_process(char*);
/*merge the contiguous unused blocks*/
void merge_left(node*);
void merge_right(node*);
void merge_twosides(node*);
/*different fitting */
int bestFit(int, char*);
int worstFit(int, char*);
int firstFit(int, char*);
```

The first function `insert_process` is used to insert a process into a hole. And the `merge_left(right/twosides)` is used to deal with the releasing operations, since if a process is released, it may make the hole bigger, and we should merge the holes if necessary as a result. Also we have other basic functions such as the three fitting algorithms and the compact operations.

To make the code more readable, we used two nodes `head` and `tail` as extra nodes to represent the head and tail of the list. And the space is initialized like :

$$head \rightarrow hole(0, 0, MAX, NULL) \rightarrow tail$$

## User Command

Users are allowed to input the command and they include:

- RQ [x] [y] [z] : request memory space sized y for x using strategy z
- RL [x] : release process x
- C : compact holes
- STAT : show current states for memory
- X : exit the program

## Detailed Implementation

Since we used linked list to implement it, the program may be a little longer(404 lines in total), and it takes me almost one day to debug it:

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

typedef struct Block{
    int used;
    int start;
    int end;
    char name[5];
}block;

typedef struct Node{
   block *hole;
   struct Node* next;
   struct Node* pre;
}node;

node* head;
node* tail;
/* allocate memory for certain process in certain memory
blocks*/
void insert_process(node* , int,char*);
/* traverse the whole memory and output*/
void traverse();
/* this function will generate a new array with new head
and tail based on the original mem*/
void compact();
/*free all the dynamic allocated memory*/
void free_all();
/*init linked list*/
void init();
/*release process into unused blocks*/
void release_process(char*);
/*merge the contiguous unused blocks*/
void merge_left(node*);
void merge_right(node*);
void merge_twosides(node*);

/*different fitting */
int bestFit(int, char*);
int worstFit(int, char*);
int firstFit(int, char*);

int max_size;

int main(int argc, char* argv[]){
    assert(argc == 2);
    max_size = atoi(argv[1]);
    init();

    traverse();
```

```
    /*init the linked list*/
    char command[5];

    while(1){
        printf("allocator>>");
        scanf("%s", command);
        if(strcmp(command, "X") == 0){
            printf("Bye!\n");
            break;
        }
        else if(strcmp(command, "RQ") == 0){
            char name[5];
            scanf("%s", name);
            int proc_size;
            char method;
            scanf("%d %c", &proc_size, &method);
            if(method == 'W'){
                int f = worstFit(proc_size, name);
                if(f < 0) printf("process insertion
failed!\n");
            }
            else if(method == 'B'){
                int f = bestFit(proc_size, name);
                if(f < 0) printf("process insertion
failed!\n");
            }
            else if(method == 'F'){
                int f = firstFit(proc_size, name);
                if(f < 0) printf("process insertion
failed!\n");
            }
            continue;
        }
        else if(strcmp(command, "RL") == 0){
            char name[5];
            scanf("%s", name);
            release_process(name);
            continue;
        }
        else if(strcmp(command, "STAT") == 0){
            traverse();
            continue;
        }
        else if(strcmp(command, "C") == 0){
            compact();
            continue;
        }else{
            printf("Wrong Command!----\n");
        }

    }
```

```
    return 0;
}

void init(){
    head  = (node*) malloc(sizeof(node));
    tail = (node*) malloc(sizeof(node));
    head -> hole = (block*)malloc(sizeof(block));
    tail -> hole = (block*)malloc(sizeof(block));
    head -> hole -> end = 0;
    node *first_block = (node*) malloc(sizeof(node));
    first_block -> hole = (block*)malloc(sizeof(block));
    first_block -> hole -> start = 0;
    first_block -> hole -> end = max_size - 1;
    first_block -> pre = head;
    first_block -> next = tail;
    head -> next = first_block;
    head -> pre = NULL;
    tail -> next = NULL;
    tail -> pre = first_block;
}

void insert_process(node* insblock, int process_size,
char*name){
    //l (proc) ins r
    node* proc = (node*) malloc(sizeof(node));
    proc -> hole = (block*)malloc(sizeof(block));
    proc -> hole -> used = 1;
    proc -> hole -> start = insblock -> hole -> start;
    proc -> hole -> end = proc -> hole -> start +
process_size - 1;
    strcpy(proc -> hole -> name, name);
    node* l  = insblock -> pre;
    l -> next = proc;
    proc -> pre = l;
    proc -> next = insblock;
    insblock -> pre = proc;
    insblock -> hole -> start = proc -> hole -> end + 1;
    if(insblock -> hole -> start >= insblock -> hole ->
end){
        // remove the hole
        node*r = insblock -> next;
        proc -> next = r;
        r -> pre = proc;
        free(insblock -> hole);
        free(insblock);
    }

}

void release_process(char* name){
    // l - proc -left
    node*proc = head;
```

```
    int f = 0;
    while(proc != tail){
        if(proc -> hole -> used == 1 && strcmp(proc ->
hole -> name, name) == 0){
            f = 1;
            break;
        }else{
            proc = proc -> next;
        }
    }
    if(f == 0){
        printf("There is no such process!\n");
        return;
    }

    proc -> hole -> used = 0;
    int l,r;
    l = (proc -> pre -> hole -> used == 0) && (proc -> pre
!= head);
    r = (proc -> next -> hole -> used == 0) && (proc ->
pre != tail);
    if(l == 0 && r == 0) return;
    if(l && r){
        merge_twosides(proc);
        return;
    }
    else if(l){
        merge_left(proc);
        return;
    }
    else{
        merge_right(proc);
        return;
    }
}

void merge_left(node* s){
      // a - left - s - b;
      // assume that before this operation, s is unused
      node*a = s -> pre -> pre;
      node* left = s -> pre;
      s -> hole -> start = s -> pre -> hole -> start;
      a -> next = s;
      s -> pre = a;
      free(left -> hole);
      free(left);
}

void merge_right(node* s){
    // a - s - right
    node* right = s -> next;
    node* a = s -> pre;
```

```
    right -> hole -> start = s -> hole -> start;
    a -> next = right;
    right -> pre = a;
    free(s -> hole);
    free(s);
}

void merge_twosides(node* s){
    // a-l-s-r-b
    node*l = s -> pre;
    node*r = s -> next;
    node*a = l -> pre;
    node*b = r -> next;

    s -> hole -> start = l -> hole -> start;
    s -> hole -> end = r -> hole -> end;
    a -> next = s;
    s -> pre = a;
    s -> next = b;
    b -> pre = s;

    free(l -> hole);
    free(l);
    free(r -> hole);
    free(r);
}


/*different fit strategies, return 0 is successfully, else
-1*/

int worstFit(int process_size, char* name){
    // find the largest hole;
    int max_hole_size = -1;
    node*tmp = head -> next;
    while(tmp != tail){
       if(tmp -> hole -> used){
           tmp = tmp -> next;
       }else{
           int hole_size = tmp -> hole -> end - tmp ->
hole -> start;
           max_hole_size = (hole_size > max_hole_size) ?
hole_size : max_hole_size;
           tmp = tmp -> next;
       }
    }
    tmp = head -> next;
    //printf("%d", process_size);
    if(max_hole_size + 1 < process_size) return -1;
    while(1){
         if(tmp -> hole -> used){
           tmp = tmp -> next;
```

```
         }else{
           int hole_size = tmp -> hole -> end - tmp ->
hole -> start;
           if(hole_size == max_hole_size){
              // printf("insert");
               insert_process(tmp, process_size, name);
               break;
           }
           tmp = tmp -> next;
       }
    }
    return 0;

}

int bestFit(int process_size, char* name){
    int min_fit_hole_size = 1e9;
    node*tmp = head -> next;
    while(tmp != tail){
       if(tmp -> hole -> used){
           tmp = tmp -> next;
           continue;
       }
       int hole_size = tmp -> hole -> end - tmp -> hole ->
start + 1;
       if(hole_size < process_size){
           tmp = tmp -> next;
           continue;
       }
       else{
           min_fit_hole_size = (hole_size <
min_fit_hole_size) ? hole_size : min_fit_hole_size;
           tmp = tmp -> next;
       }
    }

    tmp = head -> next;
    if(min_fit_hole_size == -1) return -1;
    while(1){
         if(tmp -> hole -> used){
           tmp = tmp -> next;
         }else{
           int hole_size = tmp -> hole -> end - tmp ->
hole -> start + 1;
           if(hole_size == min_fit_hole_size){
               insert_process(tmp, process_size, name);
               break;
           }
           tmp = tmp -> next;
       }
    }
    return 0;
```

```
}

int firstFit(int process_size, char *name){
    node*tmp = head -> next;
    while(tmp != tail){
        if(tmp -> hole -> used){
           tmp = tmp -> next;
         }else{
           int hole_size = tmp -> hole -> end - tmp ->
hole -> start + 1;
           if(hole_size >= process_size){
                insert_process(tmp, process_size, name);
                return 0;
           }else{
                tmp = tmp -> next;
                continue;
           }
         }
    }

    return -1;
}

void compact(){
    /*to do*/
    node*new_head=(node*)malloc(sizeof(node));
    node*new_tail=(node*)malloc(sizeof(node));
    new_head -> hole = (block*)malloc(sizeof(block));
    new_tail -> hole = (block*)malloc(sizeof(block));
    new_head -> next = new_tail;
    new_head -> pre = NULL;
    new_tail -> next = NULL;
    new_tail -> pre = new_head;

    node*tmp = head -> next;
    int free_size=0;
    node*tmp_t = new_head;
    while(tmp != tail){
        if(tmp -> hole ->used == 0){
            free_size += (tmp -> hole -> end - tmp -> hole
-> start + 1);
            //printf("FREE%d %d\n", tmp -> hole -> start,
tmp -> hole -> end);
            tmp = tmp -> next;
            continue;
        }else{
            node* proc = (node*)malloc(sizeof(node));
            proc -> hole = (block*)malloc(sizeof(block));
            int proc_size = tmp -> hole -> end - tmp ->
hole -> start + 1;
            proc ->  hole -> start = tmp_t -> hole -> end
+ 1;
```

```
            proc -> hole -> end  = proc -> hole -> start +
(proc_size - 1);
            proc -> hole -> used = 1;
            strcpy(proc -> hole -> name, tmp -> hole -
>name);
            tmp_t -> next = proc;
            proc -> next = new_tail;
            proc -> pre = tmp_t;
            new_tail -> pre = proc;
            tmp_t = proc;
        }
        tmp = tmp -> next;
    }

    node* free_sp = (node*)malloc(sizeof(node));
    free_sp -> hole = (block*)malloc(sizeof(block));
    free_sp -> hole -> start = (tmp_t -> hole -> end) + 1;
    if(free_sp -> hole -> start  == 1) free_sp -> hole ->
start  =0;
    free_sp -> hole -> end = free_sp -> hole -> start +
(free_size - 1);
    free_sp -> hole -> used = 0;
    free_sp -> pre = tmp_t;


    tail -> pre = free_sp;
    tmp_t -> next = free_sp;
    free_sp -> pre = tmp_t;
    free_sp -> next = new_tail;

    free_all(head);
    head = new_head;
    tail = new_tail;
    return;
}

void traverse(){
    node* tmp = head;
    tmp = tmp -> next;
    while(tmp != tail){
        if(tmp -> hole -> used){
            printf("Mem[%d]-[%d], [%s]\n", tmp -> hole ->
start, tmp -> hole -> end, tmp -> hole -> name);
        }else{
            printf("Mem[%d]-[%d], unused\n", tmp -> hole -
> start, tmp -> hole -> end);
        }
        tmp = tmp -> next;
    }
    return;
}
```

```
void free_all(){
    node* tmp = head;
    node* t;
    while(tmp != tail){
        t = tmp -> next;
        free(tmp->hole);
        free(tmp);
        tmp = t;
    }
    free(head -> hole);
    free(head);
    free(tail -> hole);
    free(tail);
}
```

## Evaluation

For RQ command:

```
chrisxue@chrisxue-VirtualBox:~/OS-Lab/proj7$ ./run 9999
Mem[0]-[9998], unused
allocator>>RQ P1 1000 W
allocator>>RQ P2 1500 F
allocator>>RQ P3 1300 B
allocator>>RQ P4 1100 W
allocator>>RQ P5 1000 F
allocator>>RQ P6 1200 F
allocator>>STAT
Mem[0]-[999], [P1]
Mem[1000]-[2499], [P2]
Mem[2500]-[3799], [P3]
Mem[3800]-[4899], [P4]
Mem[4900]-[5899], [P5]
Mem[5900]-[7099], [P6]
Mem[7100]-[9998], unused
```

Then for RL command, and holes are constructed:

```
allocator>>RL P2
allocator>>RL P4
allocator>>STAT
Mem[0]-[999], [P1]
Mem[1000]-[2499], unused
Mem[2500]-[3799], [P3]
Mem[3800]-[4899], unused
Mem[4900]-[5899], [P5]
Mem[5900]-[7099], [P6]
Mem[7100]-[9998], unused
allocator>>RL P6
allocator>>STAT
Mem[0]-[999], [P1]
Mem[1000]-[2499], unused
Mem[2500]-[3799], [P3]
Mem[3800]-[4899], unused
Mem[4900]-[5899], [P5]
Mem[5900]-[9998], unused
```

Then we test different strategies to arrange hole allocations:

```
allocator>>RQ P7 1000 B
allocator>>STAT
Mem[0]-[999], [P1]
Mem[1000]-[2499], unused
Mem[2500]-[3799], [P3]
Mem[3800]-[4799], [P7]
Mem[4800]-[4899], unused
Mem[4900]-[5899], [P5]
Mem[5900]-[9998], unused
```

```
allocator>>RL P3
allocator>>RL P5
allocator>>STAT
Mem[1]-[1000], [P1]
Mem[1001]-[2300], unused
Mem[2301]-[3300], [P7]
Mem[3301]-[9999], unused
allocator>>RQ P8 1000 W
allocator>>STAT
Mem[1]-[1000], [P1]
Mem[1001]-[2300], unused
Mem[2301]-[3300], [P7]
Mem[3301]-[4300], [P8]
Mem[4301]-[9999], unused
allocator>>RQ P9 1000 F
allocator>>STAT
Mem[1]-[1000], [P1]
Mem[1001]-[2000], [P9]
Mem[2001]-[2300], unused
Mem[2301]-[3300], [P7]
Mem[3301]-[4300], [P8]
Mem[4301]-[9999], unused
allocator>>X
Bye!
```

We also test the compact operation:

```
allocator>>C
allocator>>STAT
Mem[1]-[1000], [P1]
Mem[1001]-[2300], [P3]
Mem[2301]-[3300], [P7]
Mem[3301]-[4300], [P5]
Mem[4301]-[9999], unused
```

## Conclusion

This project helps me have a better understanding of the arrangement for contiguous memory including allocation and release operations. This project also made me more confident writing linked list program, including dynamic space and free operations.

Thanks for the instructions and useful help offered by Prof. Wu and all the TAs!