OS Project-7 Report

Contiguous Memory Allocation

Zhou Yiyuan

516021910270
1025152261@qq.com

1 Data Structure

I use the following data structure to record the memory:

```
typedef struct block
{
    int start;
    int end;
    int size;
    char name[MAX_SIZE];
}Block;

typedef struct block_node
{
    Block block;
    struct block_node *next;
}Node;

Node *memory;
```

A struct Block represent a hole or a used block. The name of a hole is ”unused” while the name of a used block is the process which it belongs to. The memory is a link list, organized by the order of physical addresses.

2 main()

The steps in the main() are shown as following:

(a) Initialize the memory as an unused block, whose size is defined by the command line argument.

(b) Read a command from the command line.

(c) Analyze the command and execute the different functions:

- RQ: execute allocate()
- RL: execute release()
- C: execute compact()
- STAT : output the values of the different data structures
- exit: break

(d) Jump back to step (b).

3 allocate()

There are three allocation modes: W(worst), B(best), F(first). Firstly, find the right hole according the allocation mode. If the hole's size is equal to the request size, just set the hole's name as the process name. Else, a new block should be created and inserted in front of the hole and then resize the hole's size. Three allocation modes are as following:

3.1 First fit

```c
Node *pre = NULL;
Node *tmp = memory;
if(mode == 'F'){
    while(1){
        if(tmp == NULL){
            return -1;
        }
        if(strcmp(tmp->block.name, UNUSED) == 0){
            if(tmp->block.size > size){
                Block new_block;
                strcpy(new_block.name, name);
                new_block.start = tmp->block.start;
                new_block.size = size;
                new_block.end = new_block.start + new_block.size - 1;

                Node *new_node = (Node *)malloc(sizeof(Node));
                new_node->block = new_block;
                new_node->next = tmp;

                tmp->block.start = new_block.end + 1;
                tmp->block.size = tmp->block.end - tmp->block.start + 1;

                if(pre == NULL){
                    memory = new_node;
                }else{
                    pre->next = new_node;
                }
                break;
            }else if(tmp->block.size == size){
                strcpy(tmp->block.name, name);
                break;
            }
        }
        pre = tmp;
        tmp = tmp->next;
    }
```

3.2 Worst fit

```c
Node *pre = NULL;
Node *tmp = memory;
if(mode == 'W'){
    Node *biggest = NULL;
    while(tmp != NULL){
        if(strcmp(tmp->block.name, UNUSED) == 0){
            if(biggest == NULL){
                biggest = tmp;
                continue;
            }else if(tmp->block.size > biggest->block.size){
                biggest = tmp;
            }
        }
        tmp = tmp->next;
    }
    if(biggest == NULL || biggest->block.size < size){
        return -1;
    }else if(biggest->block.size == size){
        strcpy(biggest->block.name, name);
    }else{
        Block new_block;
        strcpy(new_block.name, UNUSED);
        new_block.start = biggest->block.start + size;
        new_block.size = biggest->block.size - size;
        new_block.end = biggest->block.end;

        Node *new_node = (Node *)malloc(sizeof(Node));
        new_node->block = new_block;
        new_node->next = biggest->next;

        biggest->next = new_node;
        biggest->block.size = size;
        biggest->block.end = biggest->block.start + size - 1;
        strcpy(biggest->block.name, name);
    }
```

3.3 Best fit

```
Node *pre = NULL;
Node *tmp = memory;
if(mode == 'B'){
    Node *best = NULL;

    while(tmp != NULL){
        if(strcmp(tmp->block.name, UNUSED) == 0 && tmp->block.size >= size){
            if(best == NULL){
                best = tmp;
                continue;
            }else if(tmp->block.size < best->block.size){
                best = tmp;
            }
        }
        tmp = tmp->next;
    }

    if(best == NULL){
        return -1;
    }else if(best->block.size == size){
        strcpy(best->block.name, name);
    }else{
        Block new_block;
        strcpy(new_block.name, UNUSED);
        new_block.start = best->block.start + size;
        new_block.size = best->block.size - size;
        new_block.end = best->block.end;

        Node *new_node = (Node *)malloc(sizeof(Node));
        new_node->block = new_block;
        new_node->next = best->next;

        best->next = new_node;
        best->block.size = size;
        best->block.end = best->block.start + size - 1;
        strcpy(best->block.name, name);
    }
```

4 compact()

Create a new link list firstly. Then traverse the original link list. Skip all holes, and append the used blocks to the rail of the new link list by order, unchanging their size but changing their addresses. Then, if the memory is not full, create a new hole whose size equals to the remaining memory and append it to the new link list's rail. At last, set the new link list as the memory.

```
void compact(){
    Node *new_memory = NULL;
    Node *tmp = memory;
    Node *new_tmp = NULL;

    while(tmp != NULL){
        if(strcmp(tmp->block.name, UNUSED) != 0){
            //first used block
            if(new_memory == NULL){
                new_memory = tmp;
                tmp->block.start = 0;
                tmp->block.end = tmp->block.size - 1;
                new_tmp = new_memory;
            }else{
                new_tmp->next = tmp;
                tmp->block.start = new_tmp->block.end + 1;
                tmp->block.end = tmp->block.start + tmp->block.size - 1;
                new_tmp = tmp;
            }
        }
        tmp = tmp->next;
    }
    //the memory is empty
    if(new_memory == NULL){
        return;
    //the memory is full
    }else if(new_tmp->block.end == memory_size - 1){
        new_tmp->next = NULL;
        memory = new_memory;
    }else{//create a new hole
        Block new_block;
        strcpy(new_block.name, UNUSED);
        new_block.start = new_tmp->block.end + 1;
        new_block.end = memory_size - 1;
        new_block.size = new_block.start - new_block.end + 1;

        Node *new_node = (Node *)malloc(sizeof(Node));
        new_node->block = new_block;
        new_node->next = NULL;

        new_tmp->next = new_node;

        memory = new_memory;
    }
}
```

5 release()

If two holes are neighbors, they should be merged into one hole. Considering the previous block and the next block, there are four situations when releasing a block:

- Both of the previous block and the next block are used.
  - Set the block unused directly.
- The previous block is used and the next block is unused.
  - Delete the next block and set the block unused and enlarge the block's size.
- The previous block is unused and the next block is used.

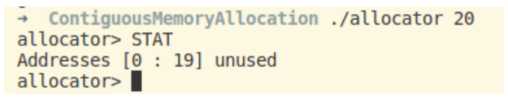
  - Delete the block and enlarge the previous block's size.

- Both of the previous block and the next block are unused.
  - Delete the block and the next block, then enlarge the previous block's size.

```c
void release(char *name){
    Node *tmp = memory;
    Node *pre = NULL;
    while(tmp != NULL){
        //find the block to release
        if(strcmp(tmp->block.name, name) == 0){
            // if the next block is unused
            if(tmp->next != NULL && strcmp(tmp->next->block.name, UNUSED) == 0){
                //if the next block and the last block are both unused, merge
                 ↪  three unused blocks
                if(pre != NULL && strcmp(pre->block.name, UNUSED) == 0){
                    pre->block.size += tmp->block.size + tmp->next->block.size;
                    pre->block.end = tmp->next->block.end;
                    pre->next = tmp->next->next;
                }else{ // if only the next block is unused, merge two unused
                 ↪  blocks
                    strcpy(tmp->block.name, UNUSED);
                    tmp->block.size += tmp->next->block.size;
                    tmp->block.end = tmp->next->block.end;
                    tmp->next = tmp->next->next;
                }
            //if only the last block is unused, merge two unused blocks
            }else if(pre != NULL && strcmp(pre->block.name, UNUSED) == 0){
                pre->block.size += tmp->block.size;
                pre->block.end = tmp->block.end;
                pre->next = tmp->next;
            }else{//if neither the last block or the next block is unused, just
             ↪  change the name
                strcpy(tmp->block.name, UNUSED);
            }
            break;
        }
        pre = tmp;
        tmp = tmp->next;
    }
    if(tmp == NULL){
        printf("fail to find %s\n", name);
    }else{
        printf("succeed to release %s\n", name);
    }
}
```

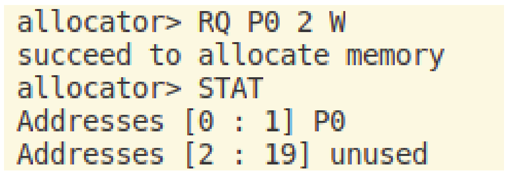
6 Results

The order of time is Figure 1-11, and all kinds of commands are used.



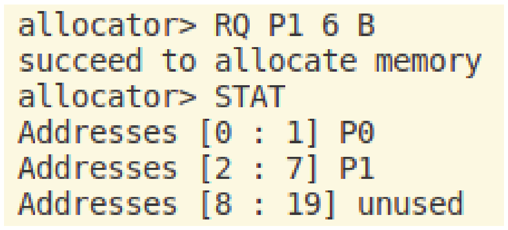
```
→ ContiguousMemoryAllocation ./allocator 20
allocator> STAT
Addresses [0 : 19] unused
allocator> 
```

Figure 1: Initialize the size of the memory as 20.



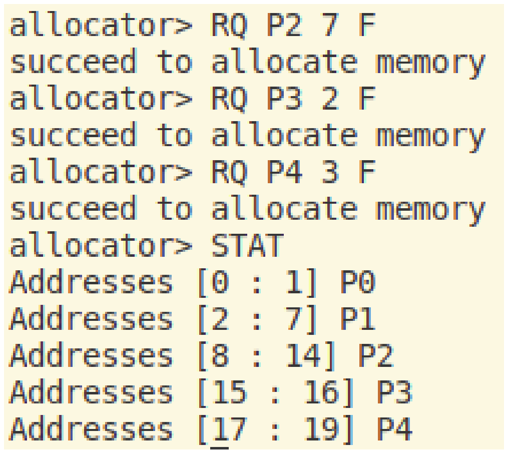
```
allocator> RQ P0 2 W
succeed to allocate memory
allocator> STAT
Addresses [0 : 1] P0
Addresses [2 : 19] unused
```

Figure 2: Allocate P0 with 2 units of memory with Worst Fit.



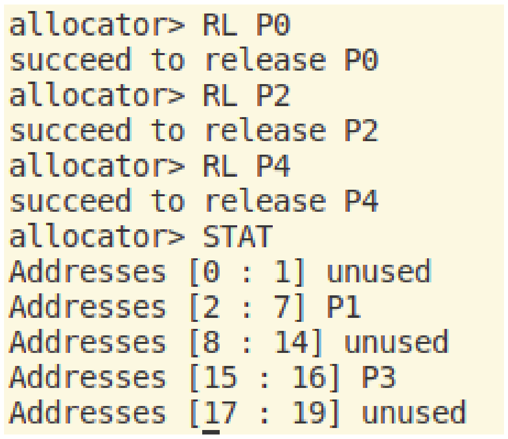
```
allocator> RQ P1 6 B
succeed to allocate memory
allocator> STAT
Addresses [0 : 1] P0
Addresses [2 : 7] P1
Addresses [8 : 19] unused
```

Figure 3: Allocate P1 with 2 units of memory with Best Fit.



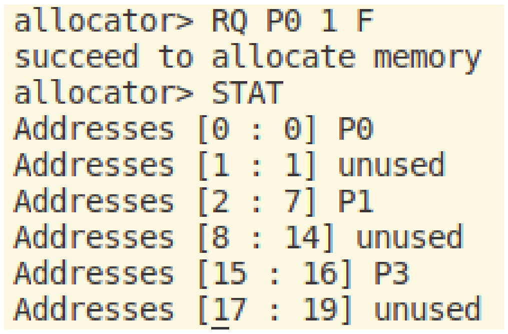
```
allocator> RQ P2 7 F
succeed to allocate memory
allocator> RQ P3 2 F
succeed to allocate memory
allocator> RQ P4 3 F
succeed to allocate memory
allocator> STAT
Addresses [0 : 1] P0
Addresses [2 : 7] P1
Addresses [8 : 14] P2
Addresses [15 : 16] P3
Addresses [17 : 19] P4
```

Figure 4: Allocate P2, P3, P4 with First Fit.



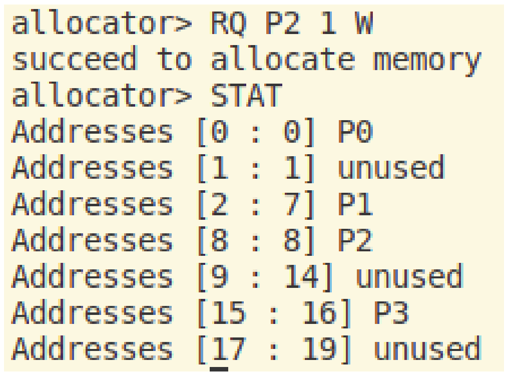
```
allocator> RL P0
succeed to release P0
allocator> RL P2
succeed to release P2
allocator> RL P4
succeed to release P4
allocator> STAT
Addresses [0 : 1] unused
Addresses [2 : 7] P1
Addresses [8 : 14] unused
Addresses [15 : 16] P3
Addresses [17 : 19] unused
```

Figure 5: Release P0, P2, P4.



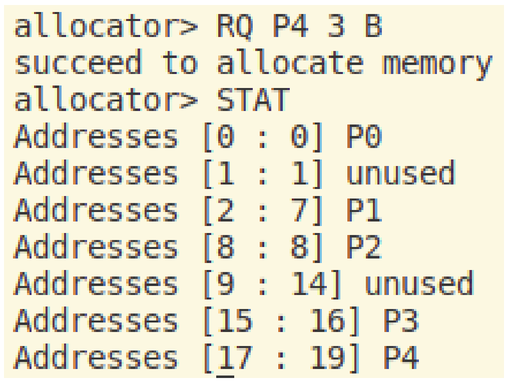
```
allocator> RQ P0 1 F
succeed to allocate memory
allocator> STAT
Addresses [0 : 0] P0
Addresses [1 : 1] unused
Addresses [2 : 7] P1
Addresses [8 : 14] unused
Addresses [15 : 16] P3
Addresses [17 : 19] unused
```

Figure 6: Allocate P0 with 1 unit of memory with First Fit.



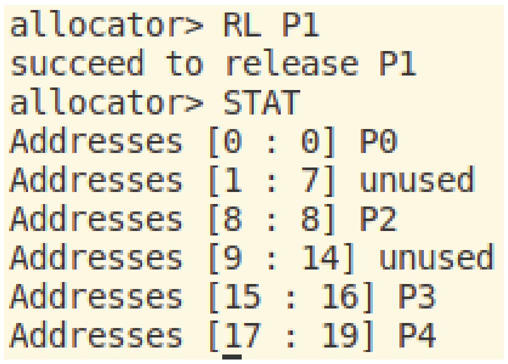
```
allocator> RQ P2 1 W
succeed to allocate memory
allocator> STAT
Addresses [0 : 0] P0
Addresses [1 : 1] unused
Addresses [2 : 7] P1
Addresses [8 : 8] P2
Addresses [9 : 14] unused
Addresses [15 : 16] P3
Addresses [17 : 19] unused
```

Figure 7: Allocate P2 with 1 unit of memory with Worst Fit.



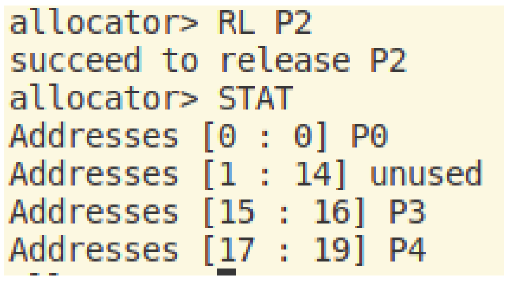
```
allocator> RQ P4 3 B
succeed to allocate memory
allocator> STAT
Addresses [0 : 0] P0
Addresses [1 : 1] unused
Addresses [2 : 7] P1
Addresses [8 : 8] P2
Addresses [9 : 14] unused
Addresses [15 : 16] P3
Addresses [17 : 19] P4
```

Figure 8: Allocate P4 with 3 units of memory with First Fit.



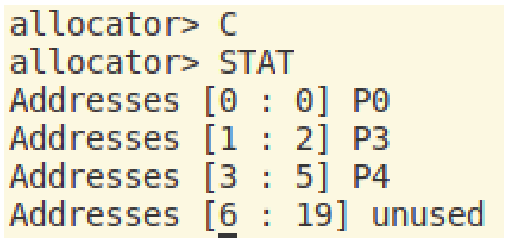
```
allocator> RL P1
succeed to release P1
allocator> STAT
Addresses [0 : 0] P0
Addresses [1 : 7] unused
Addresses [8 : 8] P2
Addresses [9 : 14] unused
Addresses [15 : 16] P3
Addresses [17 : 19] P4
```

Figure 9: Release P1 and the hole is merged with the previous hole.



```
allocator> RL P2
succeed to release P2
allocator> STAT
Addresses [0 : 0] P0
Addresses [1 : 14] unused
Addresses [15 : 16] P3
Addresses [17 : 19] P4
```

Figure 10: Release P2 and the hole is merged with the previous hole and the next hole.



```
allocator> C
allocator> STAT
Addresses [0 : 0] P0
Addresses [1 : 2] P3
Addresses [3 : 5] P4
Addresses [6 : 19] unused
```

Figure 11: Compact the memory.