Project 4 Report

Litao Zhou, 518030910407

Project Description

In this project, we implement a memory allocation algorithm.

The source code of this project can be found here.

Contents

Algorithm Implementation

The C code of the signatures and data structures of the algorithm is designed as follows.

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

// maximum length of the input name of a process
#define PROC_NAME_SIZE 20

typedef struct MemBlock {
    int beg;
    int size;
    struct MemBlock* next;
    char name[PROC_NAME_SIZE];
    int status; // 0 unused, 1 in use
} mem_block;

mem_block* mem_head = NULL;

int request_memory(char* proc_name, int request_size, char mode);
// depend on the input mode, request_memory()
// will call the following three methods
int request_memory_first_fit(char* proc_name, int request_size);
int request_memory_best_fit(char* proc_name, int request_size);
int request_memory_worst_fit(char* proc_name, int request_size);

int release_memory(char* proc_name);

void print_status();

void compact_mem();
```

We use a linked list of a struct called “mem_block” to maintain the global memory information. For every memory block, whether it is used or allocated to a process. Initially, there will be only a single memory block, with beg at 0 and size equal to the global memory size. As the requests are processed, the block will be split or joined according to the allocation strategy.

The implementation of the print_status function is trivial, so we will focus on the remaining three functions and the command line interface implementation.

Request Memory

When dealing with a request, we will delegate the request to three separate methods.

```
int request_memory(char* proc_name, int request_size, char mode){
    switch (mode)
    {
    case 'F':
```

```
    case 'f':
        return request_memory_first_fit(proc_name, request_size);
    case 'B':
    case 'b':
        return request_memory_best_fit(proc_name, request_size);
    case 'W':
    case 'w':
        return request_memory_worst_fit(proc_name, request_size);
    default:
        return -1;
    }
    return -1;
}
```

The structure of every methods are similar, it will first choose an unused block that satisfies the specified allocation strategy, then it will split the block into two parts. One is for allocation and the other part remains unused. The implementation of split_block is listed as follows. It first check whether the request_size is equal to the chosen block size , since no splitting is required for this case. Then it will split the block into two parts.

```
void split_block(char* proc_name, int request_size,
                 mem_block* current_block){
    if (request_size == current_block->size){
        // If the memory exactly fits the block, just rename and allocate it
        current_block->status = 1;
        strcpy(current_block->name, proc_name);
        return;
    }
    // otherwise, the block needs to be split
    mem_block* new_hole = malloc(sizeof(mem_block));
    new_hole->next = current_block->next;
    new_hole->size = current_block->size - request_size;
    new_hole->beg = current_block->beg + request_size;
    strcpy(new_hole->name, "Unused");
    new_hole->status = 0;

    current_block->next = new_hole;
    current_block->size = request_size;
    strcpy(current_block->name, proc_name);
    current_block->status = 1;
}
```

Now it falls on the allocation strategy to choose which block should be split. For all methods, a traverse of the memory block list is required, except that in first-fit strategy, the traverse can end on finding the first fit block.

The first fit strategy is implemented as follows.

```
int request_memory_first_fit(char* proc_name, int request_size){
    mem_block* current_block = mem_head;
    while (current_block != NULL){
        if (current_block->status == 0 &&
             current_block->size >= request_size){
            // block is unused and fit
            break;
        }
        current_block = current_block -> next;
    }
    if (current_block == NULL){
        return -1;
    }
    split_block(proc_name, request_size, current_block);
    return 0;
}
```

The best_fit strategy is implemented as follows.

```
int request_memory_best_fit(char* proc_name, int request_size){
    mem_block* best_block = NULL;
    mem_block* current_block = mem_head;
    while(current_block != NULL){
        if (current_block->status == 0 &&
             current_block->size >= request_size){
            // block is unused and fit
            if (best_block == NULL){
                best_block = current_block;
            } else if (best_block->size > current_block->size){
                // update the best block
                best_block = current_block;
            }
        }
        current_block = current_block -> next;
    }
    if (best_block == NULL){
        return -1;
    }
    split_block(proc_name, request_size, best_block);
    return 0;
}
```

The worst fit strategy is implemented as follows.

```
int request_memory_worst_fit(char* proc_name, int request_size){
    mem_block* worst_block = NULL;
    mem_block* current_block = mem_head;
    while(current_block != NULL){
        if (current_block->status == 0 &&
             current_block->size >= request_size){
            // block is unused and fit
            if (worst_block == NULL){
                worst_block = current_block;
            } else if (worst_block->size < current_block->size){
                // update the worst block
                worst_block = current_block;
            }
        }
        current_block = current_block -> next;
    }
    if (worst_block == NULL){
        return -1;
    }
    split_block(proc_name, request_size, worst_block);
    return 0;
}
```

All the three methods above will return -1 if no fit block is found, otherwise, it will return 0 indicating success.

Release Memory

There are two passes in the release memory implementation. First the blocks will be released. In our simulator, the program will simply relabel the mem_block struct that matches the given process name. In practice, the real allocation phase will also happen here.

In the second pass, the memory blocks will be checked again, and consecutive unused blocks will be merged together.

```
int release_memory(char* proc_name){
    int release_cnt = 0;
    mem_block* current_block = mem_head;
    while (current_block != NULL){
        // mark released block as unused
        if (strcmp(current_block->name,proc_name) == 0){
            current_block->status = 0;
            strcpy(current_block->name,"Unused");
            release_cnt += 1;
        }
        current_block = current_block -> next;
    }
    current_block = mem_head;
    while (current_block->next != NULL){
        // compact consecutive blocks
        if (current_block->status == 0 && current_block->next->status == 0){
            current_block->size += current_block->next->size;
            mem_block* tmp = current_block->next;
            current_block->next = current_block->next->next;
            free(tmp);
            continue; /* don't step forward, check again for consecutive */
        }
        current_block = current_block -> next;
    }
    return release_cnt;
}
```

The function will return the number of blocks that match the given process name.

Compact Memory

The implementation of compact memory function is listed as follows.

We use “used_mem” to count the amount of space for the blocks that have been moved to the head so that the next block to be moved can know where it should be moved. We use “free_mem” to count the amount of unused space that has been discovered up to now, so that we can ensure the size of the last unused block is consistent with the total memory of the system.

```
void compact_mem(){
    mem_block* current_block = mem_head;
    int free_mem = 0;
    int used_mem = 0;
    // Release the (consecutive) unused blocks at the head
    while (current_block != NULL && current_block->status == 0){
        mem_head = current_block->next;
        free_mem += current_block->size;
        free(current_block);
        current_block = mem_head;
    }
    // Now the head of the linked-list is the first allocated
    // block in the original list

    // Move the first block to the head of the actual memory
    mem_block* prev_block = current_block;
    current_block->beg = used_mem;
    used_mem += current_block->size;
    current_block = current_block->next;

    while (current_block != NULL){
        if (current_block->status == 0){
            // For unused blocks, free them
            free_mem += current_block->size;
            mem_block* tmp = current_block;
            current_block = current_block->next;
            free(tmp);
        } else {
            // For used blocks, move them to the left side
            current_block->beg = used_mem;
            used_mem += current_block->size;
            prev_block->next = current_block;
            prev_block = current_block;
            current_block = current_block->next;
        }
    }
    // Noe all rhe remaining space on the right is a single unused block
    prev_block->next = malloc(sizeof(mem_block));
    prev_block->next->beg = used_mem;
    strcpy(prev_block->next->name,"Unused");
    prev_block->next->next = NULL;
    prev_block->next->size = free_mem;
    prev_block->next->status = 0;
    return;
}
```

The function will move all the separate used blocks to the head of the memory space. The remaining space will be marked as a large unused block.

Command Line Interface

The command line interface is simple to implement. According to the user input, the CLI will call the corresponding methods above to complete the function. Note that we also add some extra codes to deal with invalid inputs and exception cases.

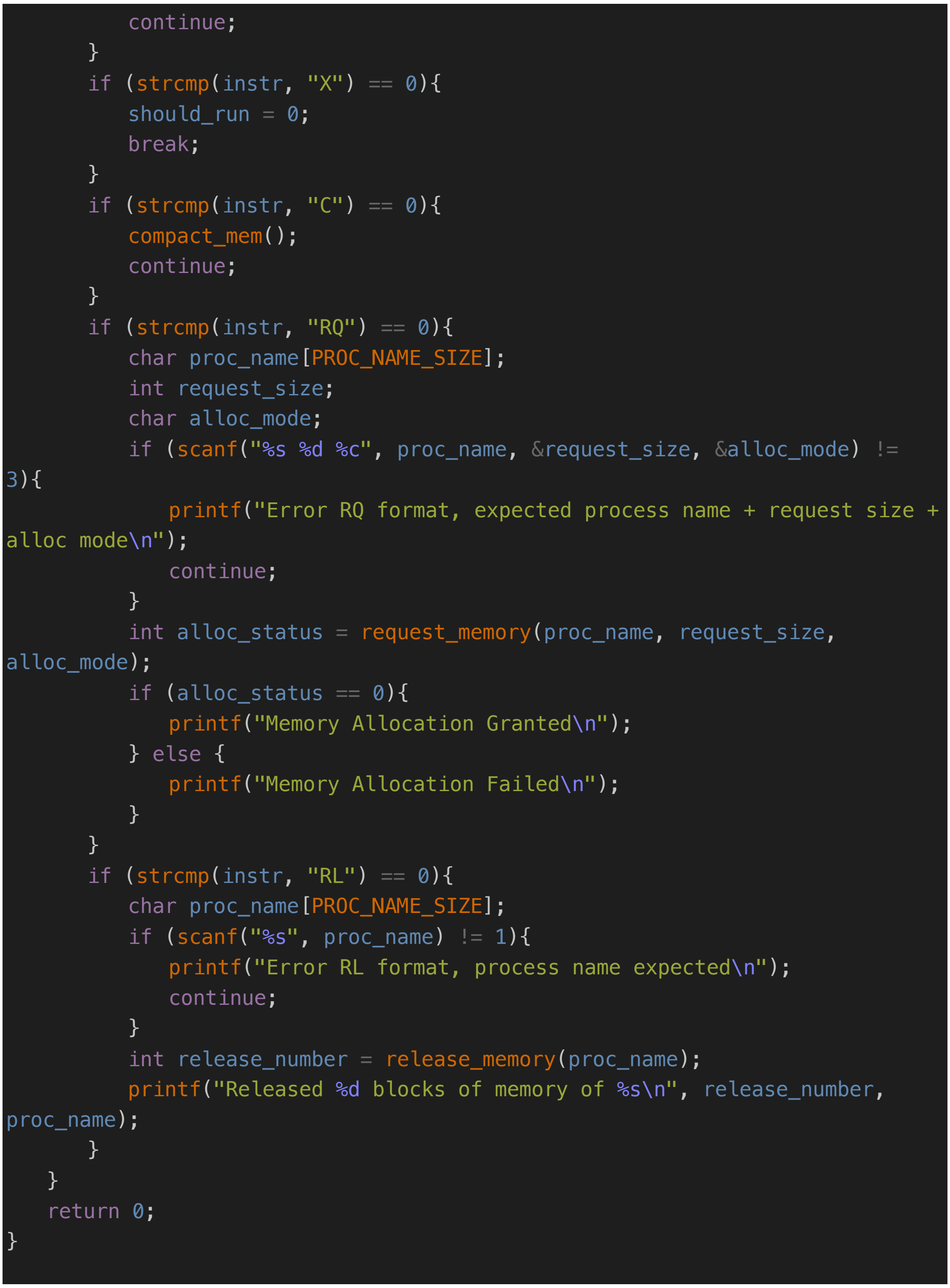
```
int main(int argc, char **argv){
    if (argc != 2){
        printf("Wrong Argument Number, input %d, but 1 (memory size) expected\n",argc-1);
        return -1;
    }

    /* initialize */
    mem_head = malloc(sizeof(mem_block));
    mem_head->beg = 0;
    if (sscanf(argv[1],"%d",&(mem_head->size)) != 1){
        printf("Input Size should be a number\n");
        return -1;
    };
    mem_head->next = NULL;
    strcpy(mem_head->name,"Unused");
    mem_head->status = 0;

    // init_data();

    int should_run = 1;

    while (should_run){
        char instr[10];
        printf("allocator> ");
        fflush(stdin);
        if (scanf("%s", instr) != 1){
            printf("Empty input.\n");
            continue;
        }
        if (strcmp(instr, "STAT") == 0){
            print_status();
```

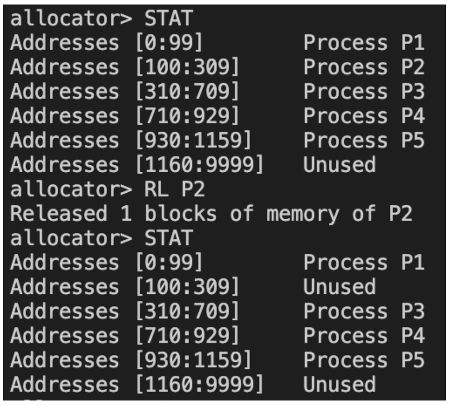


```
            continue;
        }
        if (strcmp(instr, "X") == 0){
            should_run = 0;
            break;
        }
        if (strcmp(instr, "C") == 0){
            compact_mem();
            continue;
        }
        if (strcmp(instr, "RQ") == 0){
            char proc_name[PROC_NAME_SIZE];
            int request_size;
            char alloc_mode;
            if (scanf("%s %d %c", proc_name, &request_size, &alloc_mode) !=
3){
                printf("Error RQ format, expected process name + request size +
alloc mode\n");
                continue;
            }
            int alloc_status = request_memory(proc_name, request_size,
alloc_mode);
            if (alloc_status == 0){
                printf("Memory Allocation Granted\n");
            } else {
                printf("Memory Allocation Failed\n");
            }
        }
        if (strcmp(instr, "RL") == 0){
            char proc_name[PROC_NAME_SIZE];
            if (scanf("%s", proc_name) != 1){
                printf("Error RL format, process name expected\n");
                continue;
            }
            int release_number = release_memory(proc_name);
            printf("Released %d blocks of memory of %s\n", release_number,
proc_name);
        }
    }
    return 0;
}
```

Experiment Results

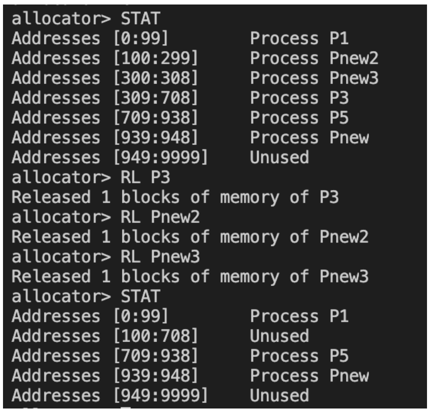
A few test cases are demonstrated below. The correctness of the function can be verified through the difference in STAT before and after the request.

Case 1: Release Block



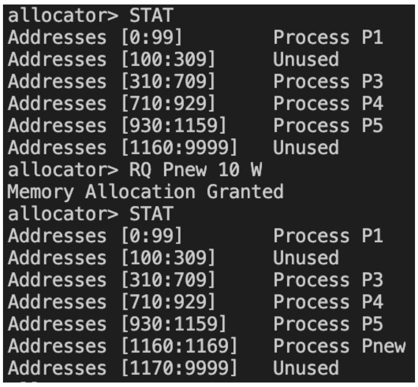
```
allocator> STAT
Addresses [0:99]        Process P1
Addresses [100:309]     Process P2
Addresses [310:709]     Process P3
Addresses [710:929]     Process P4
Addresses [930:1159]    Process P5
Addresses [1160:9999]   Unused
allocator> RL P2
Released 1 blocks of memory of P2
allocator> STAT
Addresses [0:99]        Process P1
Addresses [100:309]     Unused
Addresses [310:709]     Process P3
Addresses [710:929]     Process P4
Addresses [930:1159]    Process P5
Addresses [1160:9999]   Unused
```

Case 2: Release Consecutive Blocks and Merge



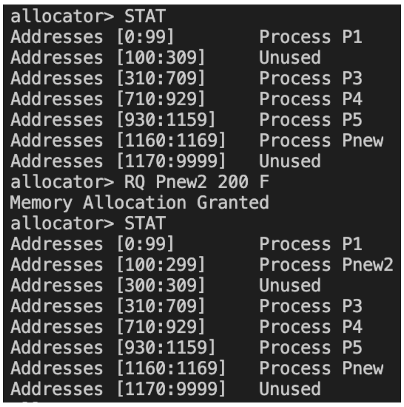
```
allocator> STAT
Addresses [0:99]        Process P1
Addresses [100:299]     Process Pnew2
Addresses [300:308]     Process Pnew3
Addresses [309:708]     Process P3
Addresses [709:938]     Process P5
Addresses [939:948]     Process Pnew
Addresses [949:9999]    Unused
allocator> RL P3
Released 1 blocks of memory of P3
allocator> RL Pnew2
Released 1 blocks of memory of Pnew2
allocator> RL Pnew3
Released 1 blocks of memory of Pnew3
allocator> STAT
Addresses [0:99]        Process P1
Addresses [100:708]     Unused
Addresses [709:938]     Process P5
Addresses [939:948]     Process Pnew
Addresses [949:9999]    Unused
```

Case 3: Request Worst-Fit



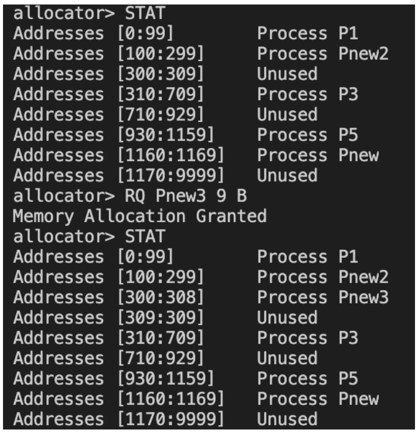
```
allocator> STAT
Addresses [0:99]        Process P1
Addresses [100:309]     Unused
Addresses [310:709]     Process P3
Addresses [710:929]     Process P4
Addresses [930:1159]    Process P5
Addresses [1160:9999]   Unused
allocator> RQ Pnew 10 W
Memory Allocation Granted
allocator> STAT
Addresses [0:99]        Process P1
Addresses [100:309]     Unused
Addresses [310:709]     Process P3
Addresses [710:929]     Process P4
Addresses [930:1159]    Process P5
Addresses [1160:1169]   Process Pnew
Addresses [1170:9999]   Unused
```

Case 4: Request First-Fit



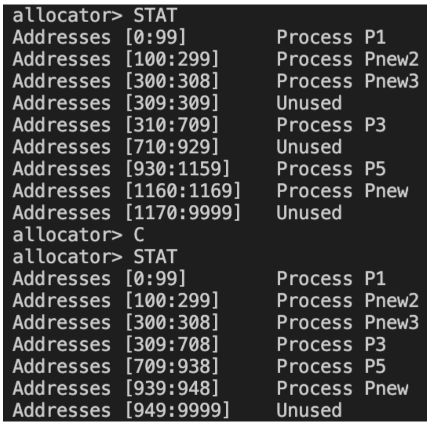
```
allocator> STAT
Addresses [0:99]          Process P1
Addresses [100:309]       Unused
Addresses [310:709]       Process P3
Addresses [710:929]       Process P4
Addresses [930:1159]      Process P5
Addresses [1160:1169]     Process Pnew
Addresses [1170:9999]     Unused
allocator> RQ Pnew2 200 F
Memory Allocation Granted
allocator> STAT
Addresses [0:99]          Process P1
Addresses [100:299]       Process Pnew2
Addresses [300:309]       Unused
Addresses [310:709]       Process P3
Addresses [710:929]       Process P4
Addresses [930:1159]      Process P5
Addresses [1160:1169]     Process Pnew
Addresses [1170:9999]     Unused
```

Case 5: Request Best-Fit



```
allocator> STAT
Addresses [0:99]          Process P1
Addresses [100:299]       Process Pnew2
Addresses [300:309]       Unused
Addresses [310:709]       Process P3
Addresses [710:929]       Unused
Addresses [930:1159]      Process P5
Addresses [1160:1169]     Process Pnew
Addresses [1170:9999]     Unused
allocator> RQ Pnew3 9 B
Memory Allocation Granted
allocator> STAT
Addresses [0:99]          Process P1
Addresses [100:299]       Process Pnew2
Addresses [300:308]       Process Pnew3
Addresses [309:309]       Unused
Addresses [310:709]       Process P3
Addresses [710:929]       Unused
Addresses [930:1159]      Process P5
Addresses [1160:1169]     Process Pnew
Addresses [1170:9999]     Unused
```

Case 6: Compact Memory



```
allocator> STAT
Addresses [0:99]          Process P1
Addresses [100:299]       Process Pnew2
Addresses [300:308]       Process Pnew3
Addresses [309:309]       Unused
Addresses [310:709]       Process P3
Addresses [710:929]       Unused
Addresses [930:1159]      Process P5
Addresses [1160:1169]     Process Pnew
Addresses [1170:9999]     Unused
allocator> C
allocator> STAT
Addresses [0:99]          Process P1
Addresses [100:299]       Process Pnew2
Addresses [300:308]       Process Pnew3
Addresses [309:708]       Process P3
Addresses [709:938]       Process P5
Addresses [939:948]       Process Pnew
Addresses [949:9999]      Unused
```