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](https://github.com/ltzone/2020Fall/tree/master/EI338/Project/memory).

Contents

[Project Description 1](#_Toc59541901)

[Algorithm Implementation 2](#_Toc59541902)

[Request Memory 2](#_Toc59541903)

[Release Memory 5](#_Toc59541904)

[Compact Memory 6](#_Toc59541905)

[Command Line Interface 7](#_Toc59541906)

[Experiment Results 9](#_Toc59541907)

[Case 1: Release Block 9](#_Toc59541908)

[Case 2: Release Consecutive Blocks and Merge 9](#_Toc59541909)

[Case 3: Request Worst-Fit 9](#_Toc59541910)

[Case 4: Request First-Fit 10](#_Toc59541911)

[Case 5: Request Best-Fit 10](#_Toc59541912)

[Case 6: Compact Memory 10](#_Toc59541913)

# 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

图片包含 文字, 牌匾, 游戏机

描述已自动生成

## Case 2: Release Consecutive Blocks and Merge

一些文字和图片

描述已自动生成

## Case 3: Request Worst-Fit

图片包含 文字, 游戏机, 牌匾

描述已自动生成

## Case 4: Request First-Fit

一些文字和图片

中度可信度描述已自动生成

## Case 5: Request Best-Fit

一些文字和图片

中度可信度描述已自动生成

## Case 6: Compact Memory

一些文字被放在一起

中度可信度描述已自动生成