Programming Report

Description:

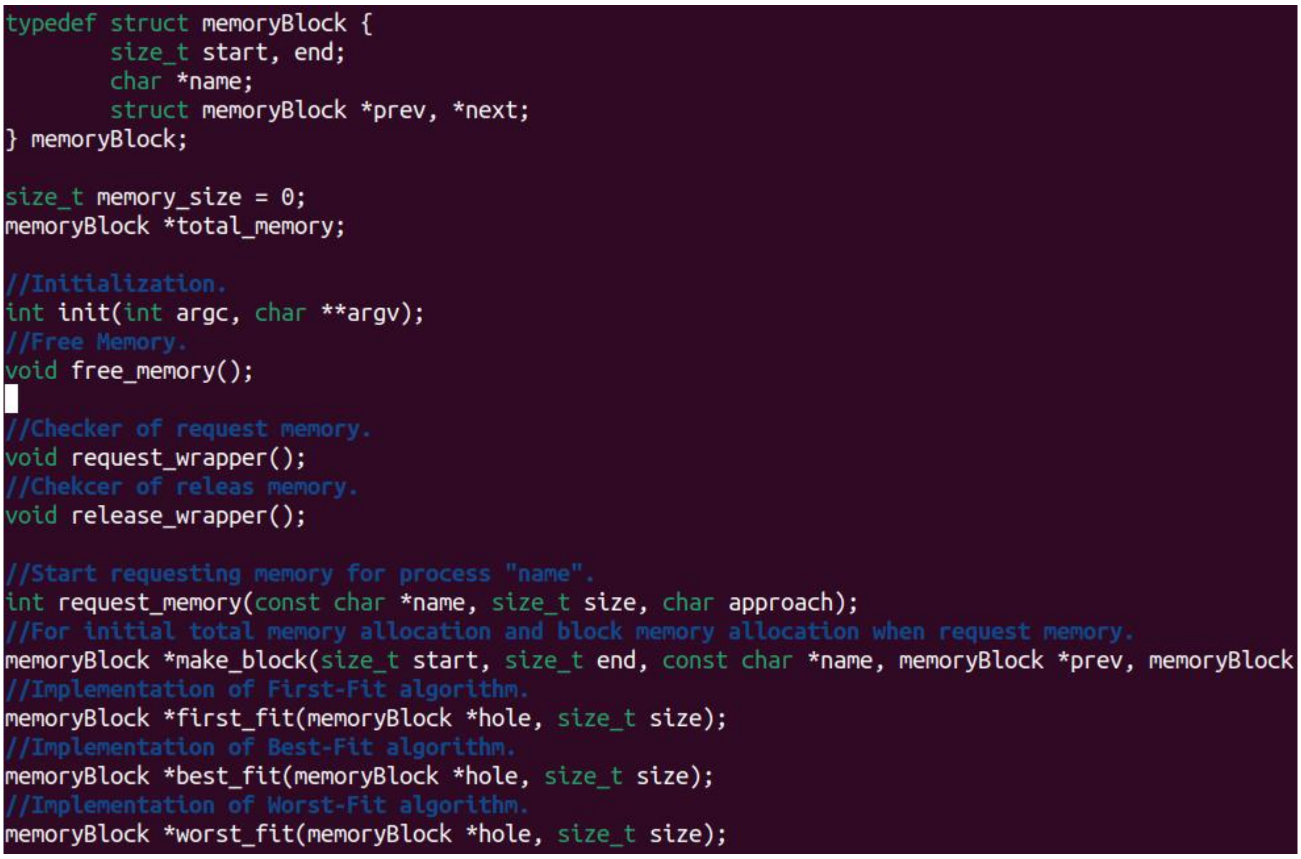
There are 2 projects:
Project 1: Contiguous Memory Allocation. P1-P9
Project 2: Design a Virtual Memory Manager. P10-P16
And I will show you the codes, results, and explanation of each Project.

Project 1: Contiguous Memory Allocation.

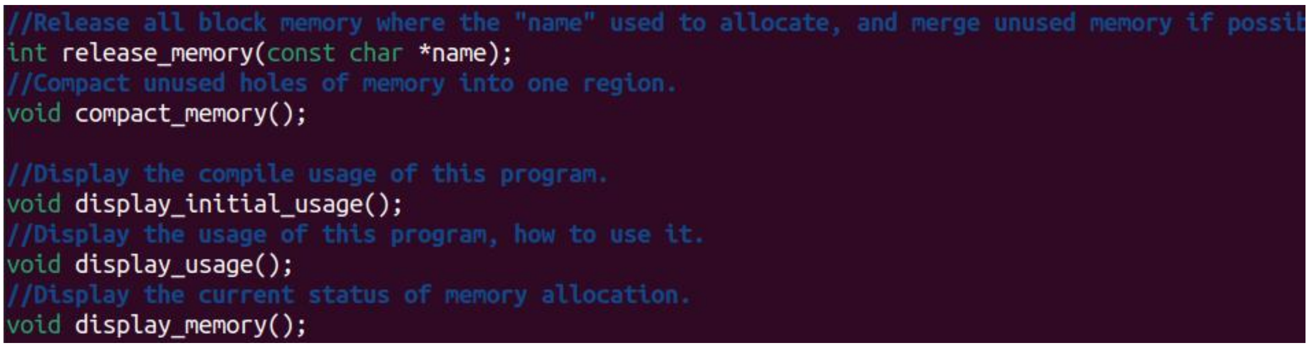
Description:

1) In this project, I need to implement a memory allocator with 4 operations (Request Memory, Release Memory, Compact Unused Memory, and Display the Current Situation of Allocation) to simulate managing a contiguous region of memory from a Operating System.
2) To run this program, you need to involve an argument represents the initial amount of memory at startup. For my program, you can simply run "./run.sh", then it will help you to set the initial amount to 1000 as default: "./allocator 1000". You can also use "./allocator <number>" with your argument.

Functions:



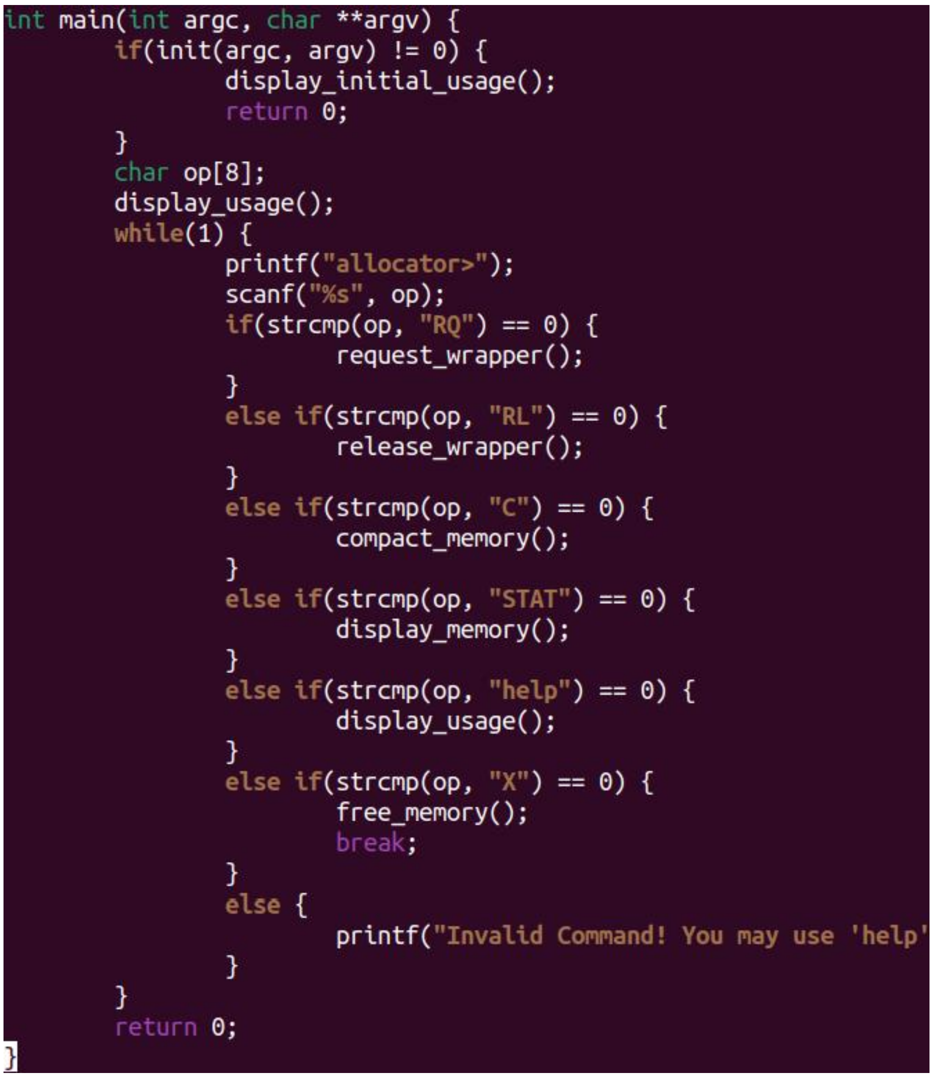
```c
typedef struct memoryBlock {
        size_t start, end;
        char *name;
        struct memoryBlock *prev, *next;
} memoryBlock;

size_t memory_size = 0;
memoryBlock *total_memory;

//Initialization.
int init(int argc, char **argv);
//Free Memory.
void free_memory();

//Checker of request memory.
void request_wrapper();
//Chekcer of releas memory.
void release_wrapper();

//Start requesting memory for process "name".
int request_memory(const char *name, size_t size, char approach);
//For initial total memory allocation and block memory allocation when request memory.
memoryBlock *make_block(size_t start, size_t end, const char *name, memoryBlock *prev, memoryBlock
//Implementation of First-Fit algorithm.
memoryBlock *first_fit(memoryBlock *hole, size_t size);
//Implementation of Best-Fit algorithm.
memoryBlock *best_fit(memoryBlock *hole, size_t size);
//Implementation of Worst-Fit algorithm.
memoryBlock *worst_fit(memoryBlock *hole, size_t size);
```



```
//Release all block memory where the "name" used to allocate, and merge unused memory if possi
int release_memory(const char *name);
//Compact unused holes of memory into one region.
void compact_memory();

//Display the compile usage of this program.
void display_initial_usage();
//Display the usage of this program, how to use it.
void display_usage();
//Display the current status of memory allocation.
void display_memory();
```

1) I first implement the function “make_block()” for block memory allocation.
2) The operation “Request Memory” is implemented by: “request_wrapper()”, “request_memory()”, 3 algorithms: “first_fit()”, “best_fit()”, and “worst_fit()”.
3) The operation “Compact” is implemented by “compact_memory()”, the operation “Display” is implemented by “display_memory()”.

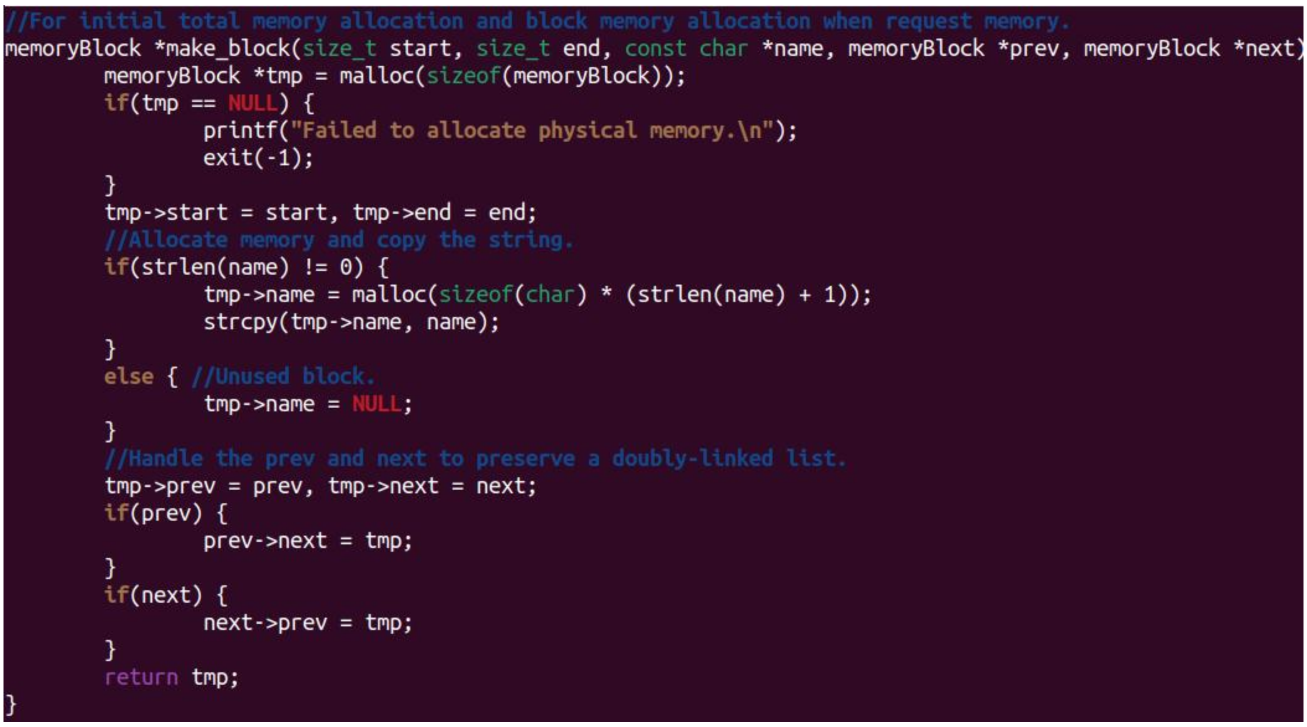
Main Function:



```
int main(int argc, char **argv) {
        if(init(argc, argv) != 0) {
                display_initial_usage();
                return 0;
        }
        char op[8];
        display_usage();
        while(1) {
                printf("allocator>");
                scanf("%s", op);
                if(strcmp(op, "RQ") == 0) {
                        request_wrapper();
                }
                else if(strcmp(op, "RL") == 0) {
                        release_wrapper();
                }
                else if(strcmp(op, "C") == 0) {
                        compact_memory();
                }
                else if(strcmp(op, "STAT") == 0) {
                        display_memory();
                }
                else if(strcmp(op, "help") == 0) {
                        display_usage();
                }
                else if(strcmp(op, "X") == 0) {
                        free_memory();
                        break;
                }
                else {
                        printf("Invalid Command! You may use 'help'
                }
        }
        return 0;
}
```

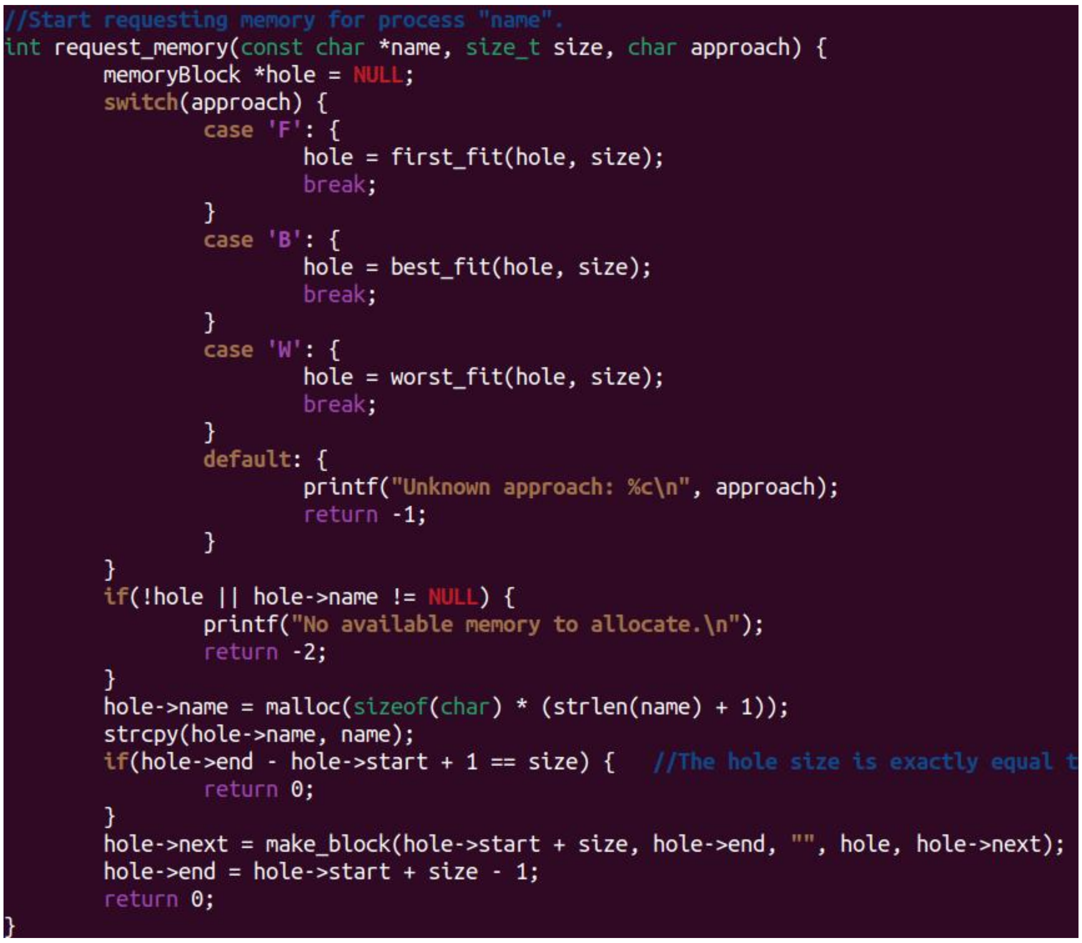
The main function constructed the terminal shell to accept command.

Make Block:



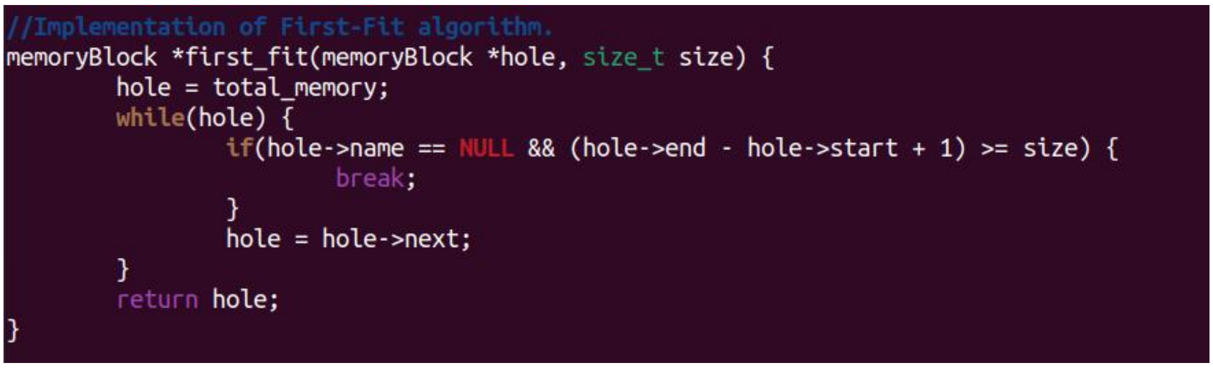
```c
//For initial total memory allocation and block memory allocation when request memory.
memoryBlock *make_block(size_t start, size_t end, const char *name, memoryBlock *prev, memoryBlock *next)
        memoryBlock *tmp = malloc(sizeof(memoryBlock));
        if(tmp == NULL) {
                printf("Failed to allocate physical memory.\n");
                exit(-1);
        }
        tmp->start = start, tmp->end = end;
        //Allocate memory and copy the string.
        if(strlen(name) != 0) {
                tmp->name = malloc(sizeof(char) * (strlen(name) + 1));
                strcpy(tmp->name, name);
        }
        else { //Unused block.
                tmp->name = NULL;
        }
        //Handle the prev and next to preserve a doubly-linked list.
        tmp->prev = prev, tmp->next = next;
        if(prev) {
                prev->next = tmp;
        }
        if(next) {
                next->prev = tmp;
        }
        return tmp;
}
```

Request Memory:



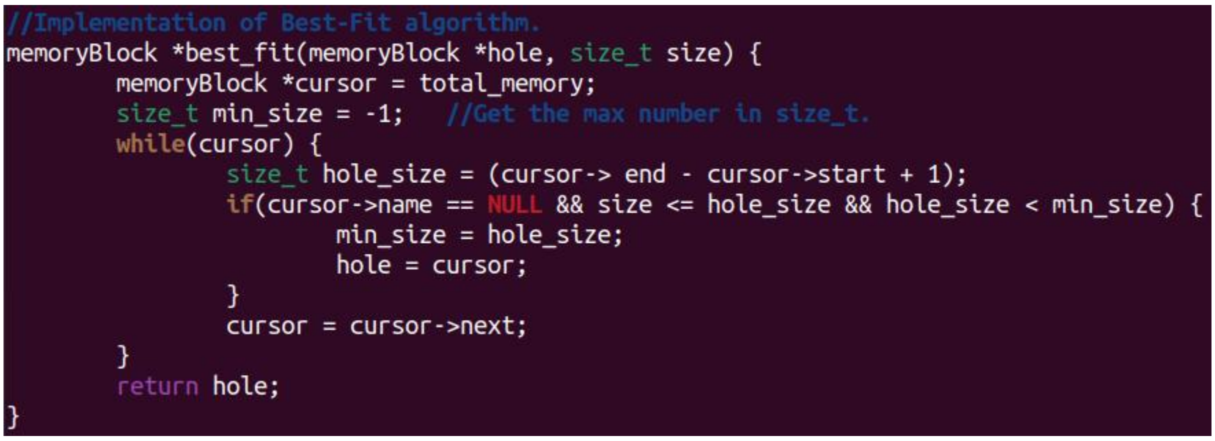
```c
//Start requesting memory for process "name".
int request_memory(const char *name, size_t size, char approach) {
        memoryBlock *hole = NULL;
        switch(approach) {
                case 'F': {
                        hole = first_fit(hole, size);
                        break;
                }
                case 'B': {
                        hole = best_fit(hole, size);
                        break;
                }
                case 'W': {
                        hole = worst_fit(hole, size);
                        break;
                }
                default: {
                        printf("Unknown approach: %c\n", approach);
                        return -1;
                }
        }
        if(!hole || hole->name != NULL) {
                printf("No available memory to allocate.\n");
                return -2;
        }
        hole->name = malloc(sizeof(char) * (strlen(name) + 1));
        strcpy(hole->name, name);
        if(hole->end - hole->start + 1 == size) {   //The hole size is exactly equal t
                return 0;
        }
        hole->next = make_block(hole->start + size, hole->end, "", hole, hole->next);
        hole->end = hole->start + size - 1;
        return 0;
}
```

First Fit:



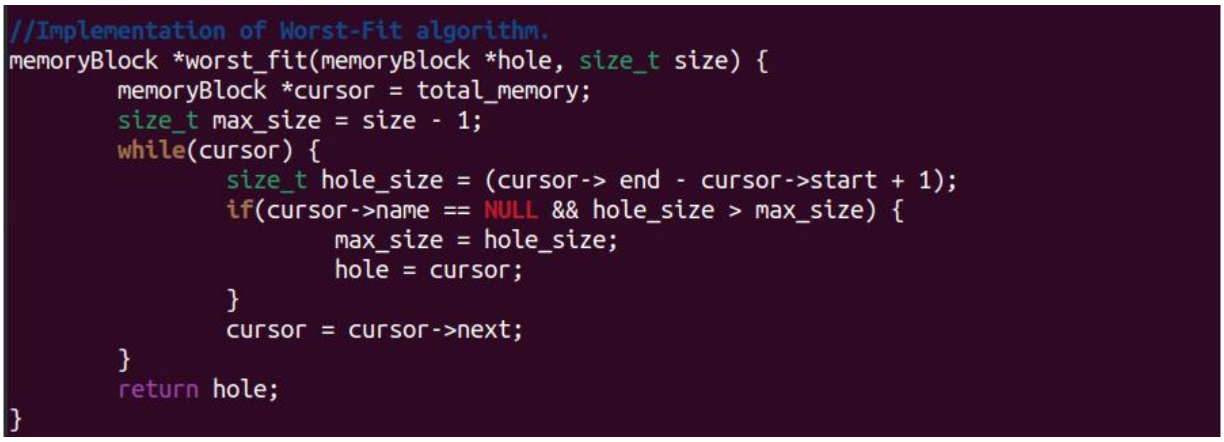
```c
//Implementation of First-Fit algorithm.
memoryBlock *first_fit(memoryBlock *hole, size_t size) {
        hole = total_memory;
        while(hole) {
                if(hole->name == NULL && (hole->end - hole->start + 1) >= size) {
                        break;
                }
                hole = hole->next;
        }
        return hole;
}
```

Best Fit:



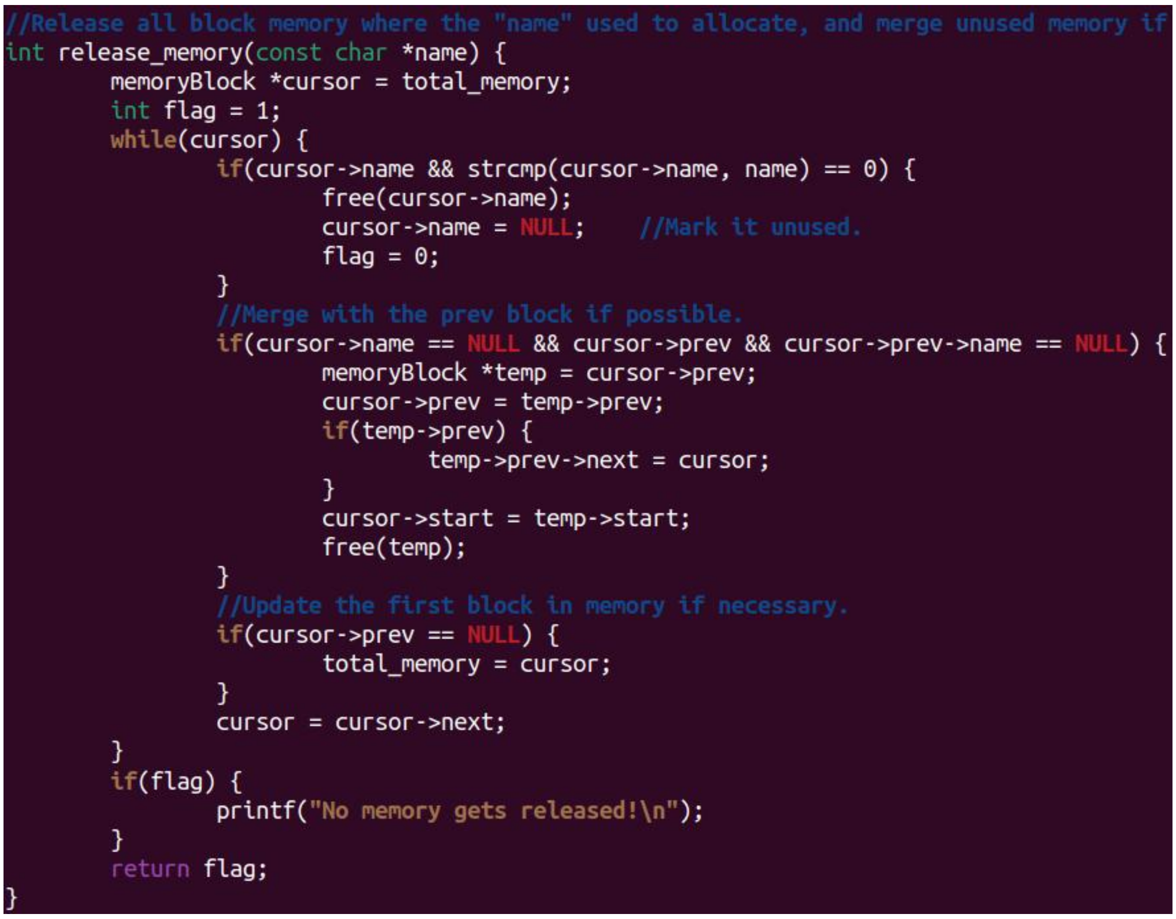
```c
//Implementation of Best-Fit algorithm.
memoryBlock *best_fit(memoryBlock *hole, size_t size) {
        memoryBlock *cursor = total_memory;
        size_t min_size = -1;   //Get the max number in size_t.
        while(cursor) {
                size_t hole_size = (cursor-> end - cursor->start + 1);
                if(cursor->name == NULL && size <= hole_size && hole_size < min_size) {
                        min_size = hole_size;
                        hole = cursor;
                }
                cursor = cursor->next;
        }
        return hole;
}
```

Worst Fit:



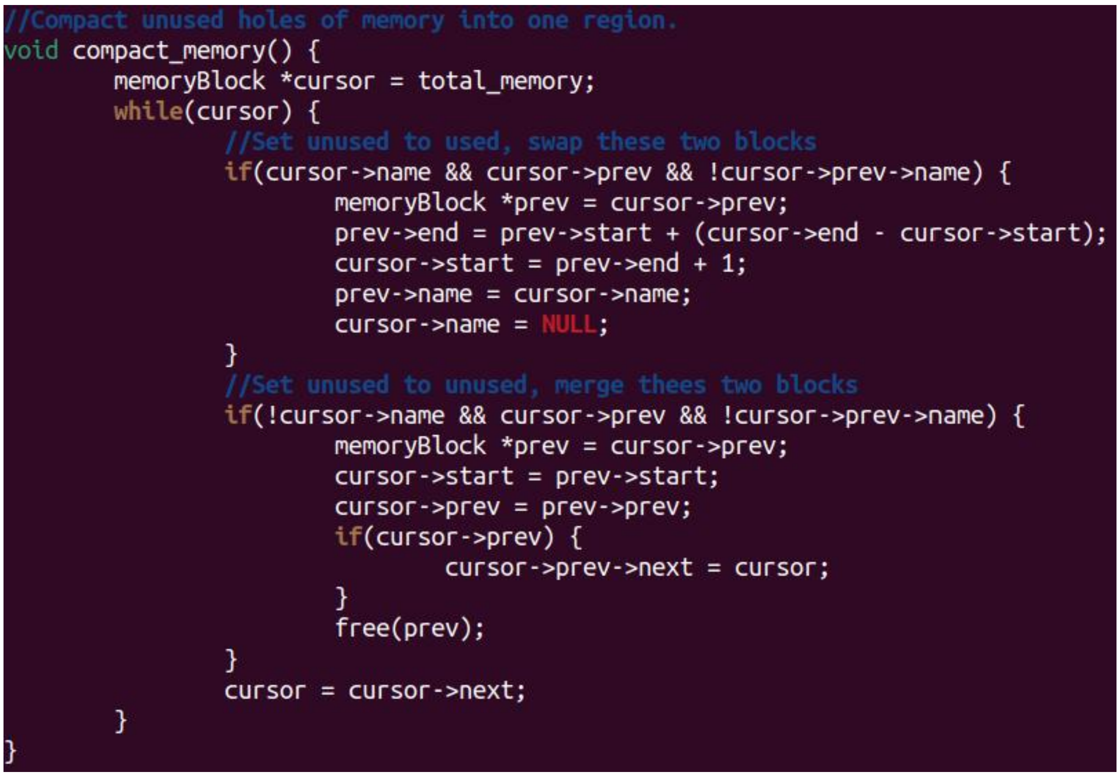
```c
//Implementation of Worst-Fit algorithm.
memoryBlock *worst_fit(memoryBlock *hole, size_t size) {
        memoryBlock *cursor = total_memory;
        size_t max_size = size - 1;
        while(cursor) {
                size_t hole_size = (cursor-> end - cursor->start + 1);
                if(cursor->name == NULL && hole_size > max_size) {
                        max_size = hole_size;
                        hole = cursor;
                }
                cursor = cursor->next;
        }
        return hole;
}
```

Release Memory:



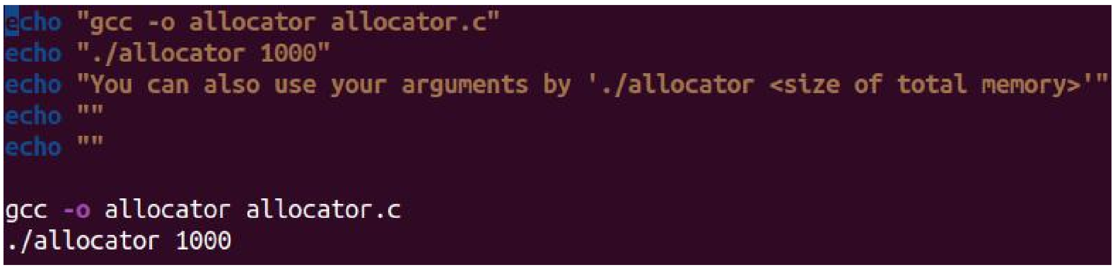
```
//Release all block memory where the "name" used to allocate, and merge unused memory if
int release_memory(const char *name) {
        memoryBlock *cursor = total_memory;
        int flag = 1;
        while(cursor) {
                if(cursor->name && strcmp(cursor->name, name) == 0) {
                        free(cursor->name);
                        cursor->name = NULL;    //Mark it unused.
                        flag = 0;
                }
                //Merge with the prev block if possible.
                if(cursor->name == NULL && cursor->prev && cursor->prev->name == NULL) {
                        memoryBlock *temp = cursor->prev;
                        cursor->prev = temp->prev;
                        if(temp->prev) {
                                temp->prev->next = cursor;
                        }
                        cursor->start = temp->start;
                        free(temp);
                }
                //Update the first block in memory if necessary.
                if(cursor->prev == NULL) {
                        total_memory = cursor;
                }
                cursor = cursor->next;
        }
        if(flag) {
                printf("No memory gets released!\n");
        }
        return flag;
}
```

Compact Memory:



```
//Compact unused holes of memory into one region.
void compact_memory() {
        memoryBlock *cursor = total_memory;
        while(cursor) {
                //Set unused to used, swap these two blocks
                if(cursor->name && cursor->prev && !cursor->prev->name) {
                        memoryBlock *prev = cursor->prev;
                        prev->end = prev->start + (cursor->end - cursor->start);
                        cursor->start = prev->end + 1;
                        prev->name = cursor->name;
                        cursor->name = NULL;
                }
                //Set unused to unused, merge thees two blocks
                if(!cursor->name && cursor->prev && !cursor->prev->name) {
                        memoryBlock *prev = cursor->prev;
                        cursor->start = prev->start;
                        cursor->prev = prev->prev;
                        if(cursor->prev) {
                                cursor->prev->next = cursor;
                        }
                        free(prev);
                }
                cursor = cursor->next;
        }
}
```

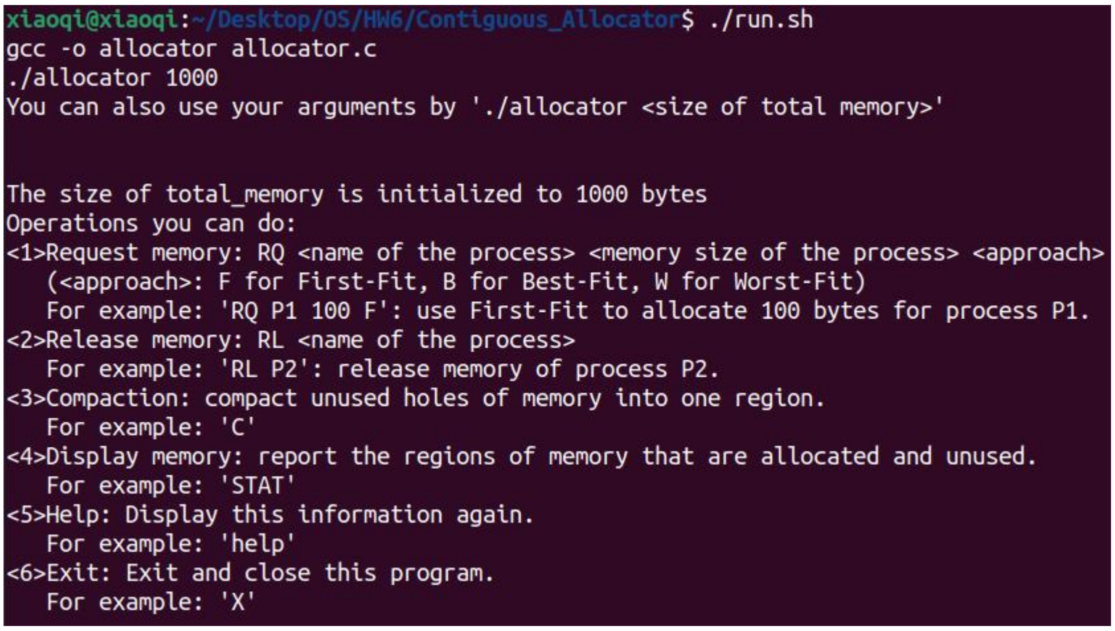
Compile (run.sh):



```
echo "gcc -o allocator allocator.c"
echo "./allocator 1000"
echo "You can also use your arguments by './allocator <size of total memory>'"
echo ""
echo ""

gcc -o allocator allocator.c
./allocator 1000
```

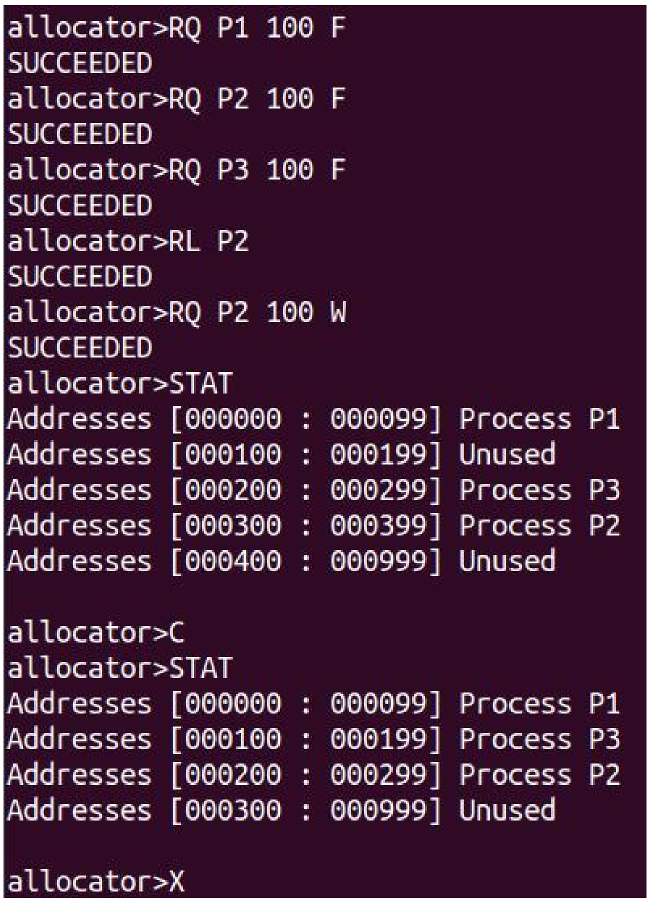
If you run “run.sh” program directly, I set the default number of the length of memory to 1000, you can also use your argument of length of memory you want.

Results:



```
xiaoqi@xiaoqi:~/Desktop/OS/HW6/Contiguous_Allocator$ ./run.sh
gcc -o allocator allocator.c
./allocator 1000
You can also use your arguments by './allocator <size of total memory>'


The size of total_memory is initialized to 1000 bytes
Operations you can do:
<1>Request memory: RQ <name of the process> <memory size of the process> <approach>
   (<approach>: F for First-Fit, B for Best-Fit, W for Worst-Fit)
   For example: 'RQ P1 100 F': use First-Fit to allocate 100 bytes for process P1.
<2>Release memory: RL <name of the process>
   For example: 'RL P2': release memory of process P2.
<3>Compaction: compact unused holes of memory into one region.
   For example: 'C'
<4>Display memory: report the regions of memory that are allocated and unused.
   For example: 'STAT'
<5>Help: Display this information again.
   For example: 'help'
<6>Exit: Exit and close this program.
   For example: 'X'
```

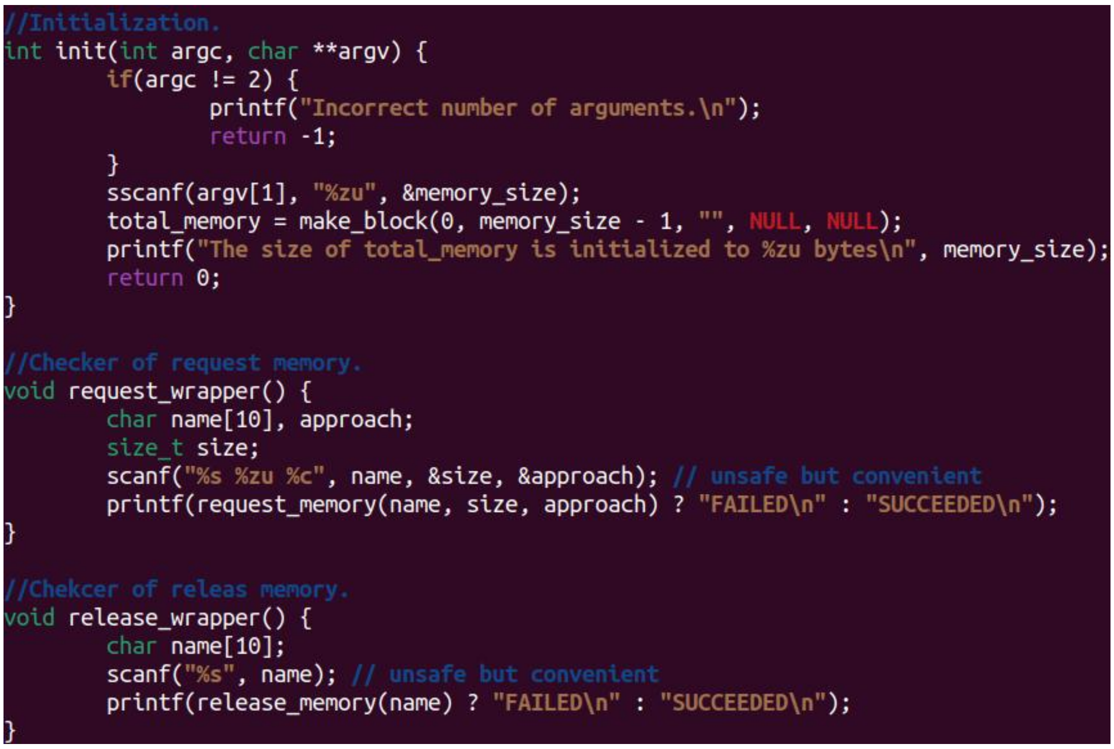
You can see once you run this program, it will confirm how many memory you set first, and then show you all the operations you can enter to interact with this program. (Request Memory - RQ, Release Memory - RL, Compaction - C, Display Memory - STAT, Help - help, and Exit - X)



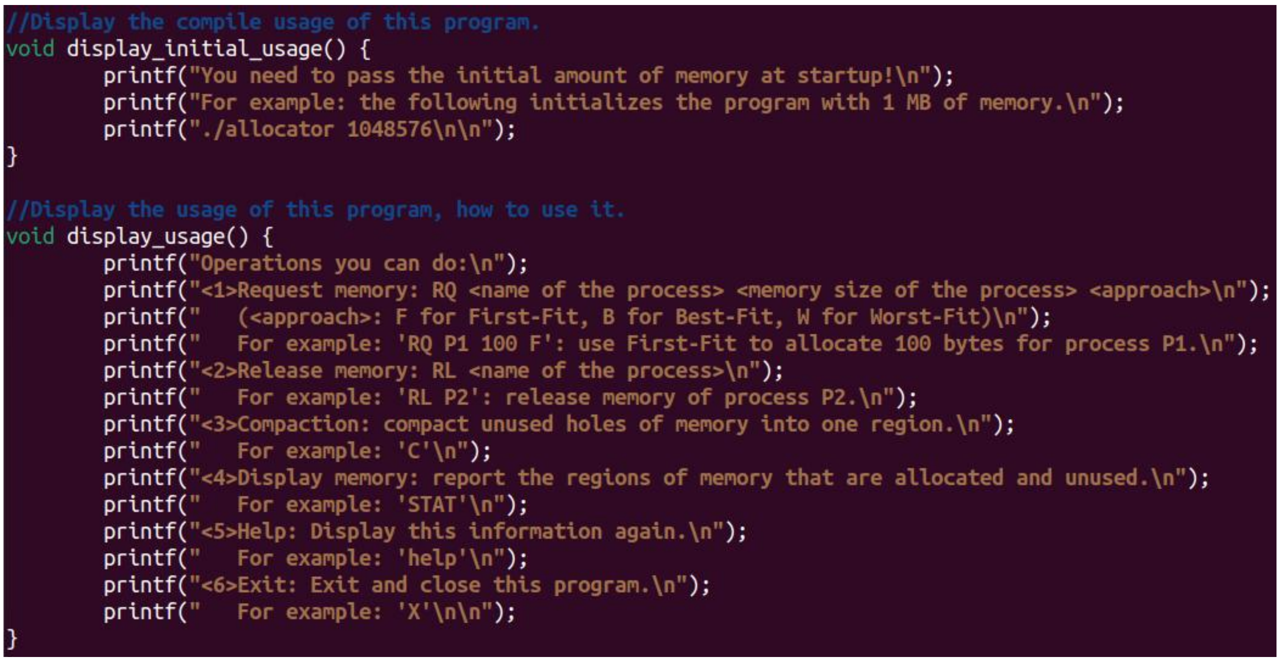
```
allocator>RQ P1 100 F
SUCCEEDED
allocator>RQ P2 100 F
SUCCEEDED
allocator>RQ P3 100 F
SUCCEEDED
allocator>RL P2
SUCCEEDED
allocator>RQ P2 100 W
SUCCEEDED
allocator>STAT
Addresses [000000 : 000099] Process P1
Addresses [000100 : 000199] Unused
Addresses [000200 : 000299] Process P3
Addresses [000300 : 000399] Process P2
Addresses [000400 : 000999] Unused

allocator>C
allocator>STAT
Addresses [000000 : 000099] Process P1
Addresses [000100 : 000199] Process P3
Addresses [000200 : 000299] Process P2
Addresses [000300 : 000999] Unused

allocator>X
```

1) You can see after I request memory for 3 processes and release the memory of P2, the allocation status will be: P1[0-99], Unused[100-199], P3[200-299].
2) Then, I used Worst Fit to allocate the memory of P2 again, and the STAT shows P2 is after P3.
3) Then, I used Compaction to compact unused memory, and the Unused[100-199] was joined into Unused[400-999].
4) Finally, I used X to exit and close this program.

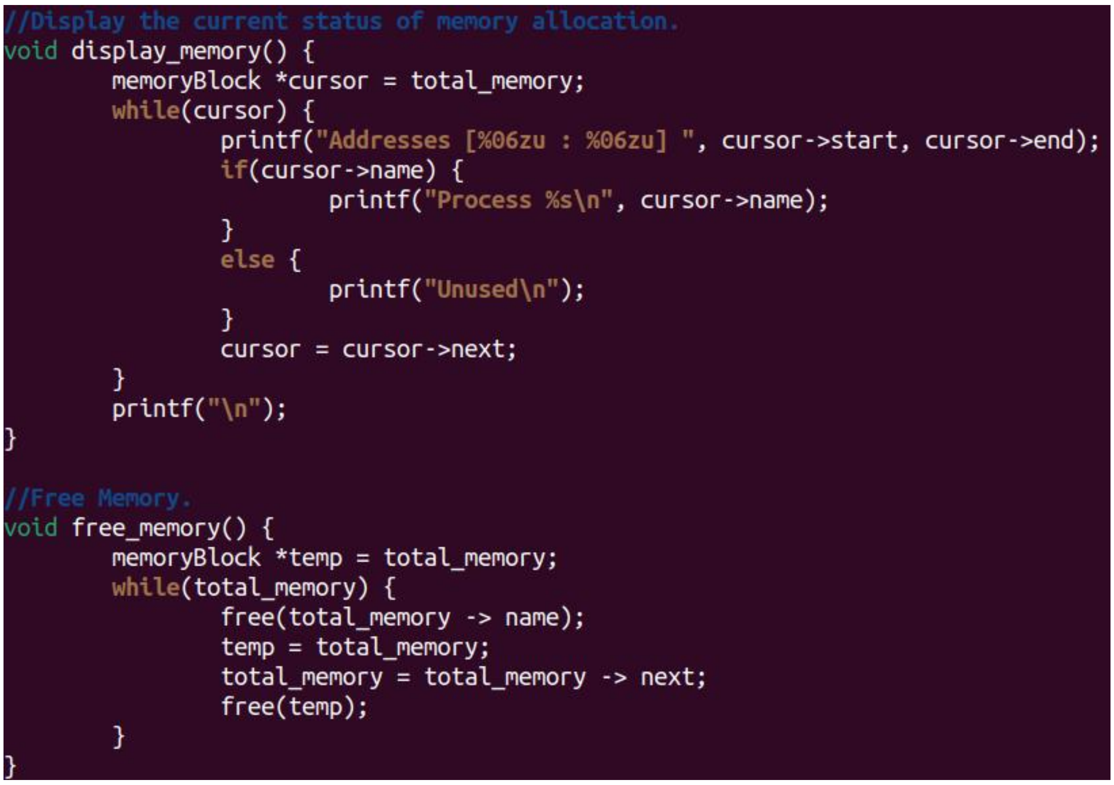
Other Codes:



```c
//Initialization.
int init(int argc, char **argv) {
        if(argc != 2) {
                printf("Incorrect number of arguments.\n");
                return -1;
        }
        sscanf(argv[1], "%zu", &memory_size);
        total_memory = make_block(0, memory_size - 1, "", NULL, NULL);
        printf("The size of total_memory is initialized to %zu bytes\n", memory_size);
        return 0;
}

//Checker of request memory.
void request_wrapper() {
        char name[10], approach;
        size_t size;
        scanf("%s %zu %c", name, &size, &approach); // unsafe but convenient
        printf(request_memory(name, size, approach) ? "FAILED\n" : "SUCCEEDED\n");
}

//Chekcer of releas memory.
void release_wrapper() {
        char name[10];
        scanf("%s", name); // unsafe but convenient
        printf(release_memory(name) ? "FAILED\n" : "SUCCEEDED\n");
}
```



```c
//Display the compile usage of this program.
void display_initial_usage() {
        printf("You need to pass the initial amount of memory at startup!\n");
        printf("For example: the following initializes the program with 1 MB of memory.\n");
        printf("./allocator 1048576\n\n");
}

//Display the usage of this program, how to use it.
void display_usage() {
        printf("Operations you can do:\n");
        printf("<1>Request memory: RQ <name of the process> <memory size of the process> <approach>\n");
        printf("    (<approach>: F for First-Fit, B for Best-Fit, W for Worst-Fit)\n");
        printf("    For example: 'RQ P1 100 F': use First-Fit to allocate 100 bytes for process P1.\n");
        printf("<2>Release memory: RL <name of the process>\n");
        printf("    For example: 'RL P2': release memory of process P2.\n");
        printf("<3>Compaction: compact unused holes of memory into one region.\n");
        printf("    For example: 'C'\n");
        printf("<4>Display memory: report the regions of memory that are allocated and unused.\n");
        printf("    For example: 'STAT'\n");
        printf("<5>Help: Display this information again.\n");
        printf("    For example: 'help'\n");
        printf("<6>Exit: Exit and close this program.\n");
        printf("    For example: 'X'\n\n");
}
```



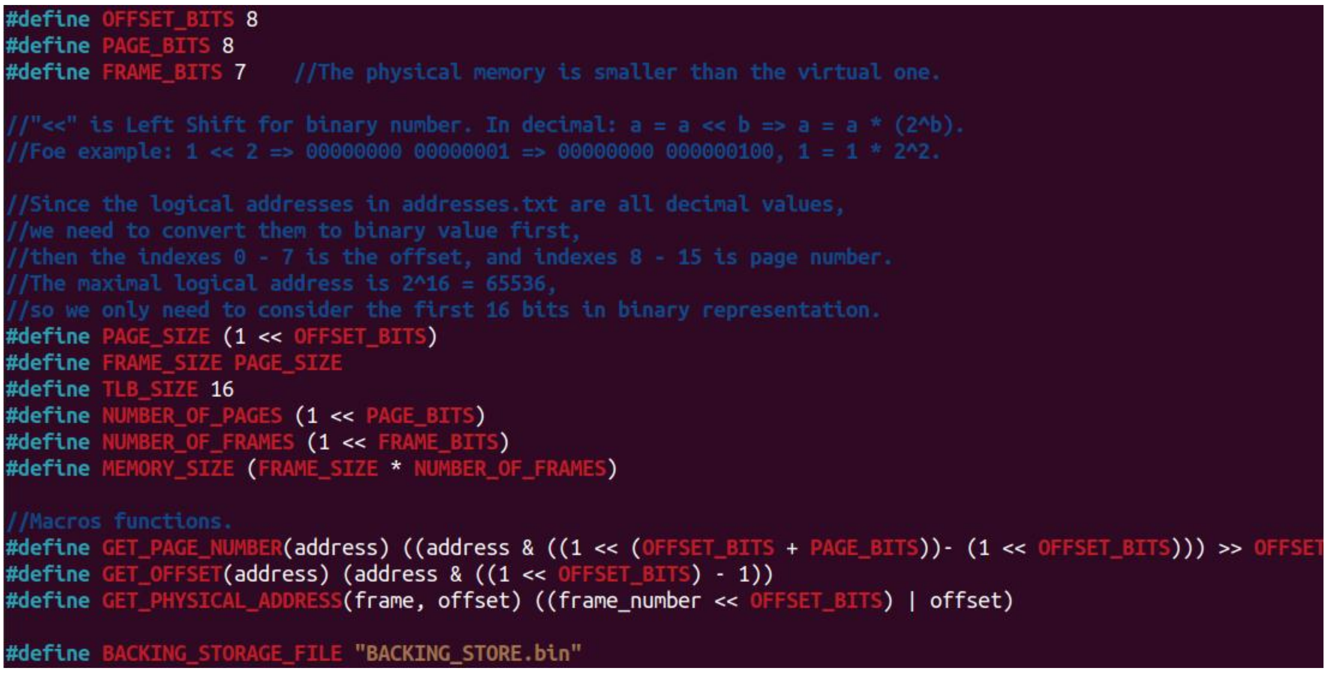
```
//Display the current status of memory allocation.
void display_memory() {
        memoryBlock *cursor = total_memory;
        while(cursor) {
                printf("Addresses [%06zu : %06zu] ", cursor->start, cursor->end);
                if(cursor->name) {
                        printf("Process %s\n", cursor->name);
                }
                else {
                        printf("Unused\n");
                }
                cursor = cursor->next;
        }
        printf("\n");
}

//Free Memory.
void free_memory() {
        memoryBlock *temp = total_memory;
        while(total_memory) {
                free(total_memory -> name);
                temp = total_memory;
                total_memory = total_memory -> next;
                free(temp);
        }
}
```

Project 2: Design a Virtual Memory Manager.

Description:

1) This project asks us using TLB and a page table to translate logical address to its corresponding physical address, and output the value of the byte stored at the translated physical address.
2) This project has 3 source files: “BACKING_STORE.bin” - a binary file representing the backing store to for “Demand Paging”, “addresses.txt” - contains 1000 logical addresses ranging from 0 to 65535, and “correct.txt” contains all the correct output values for the logical addresses from “addresses.txt”, I can use it to determine whether my program works correctly.

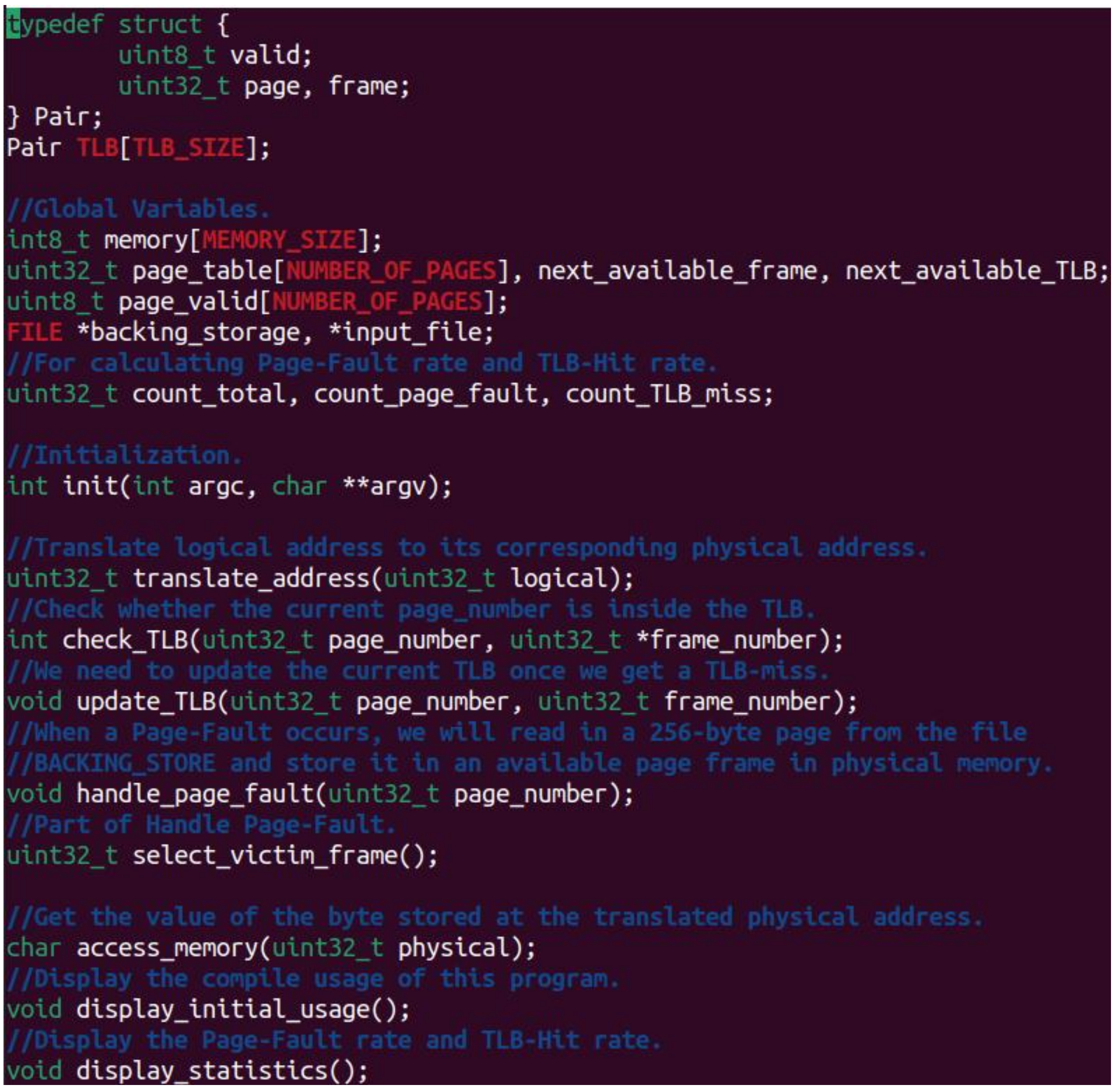
Macros:



```
#define OFFSET_BITS 8
#define PAGE_BITS 8
#define FRAME_BITS 7    //The physical memory is smaller than the virtual one.

//"<<" is Left Shift for binary number. In decimal: a = a << b => a = a * (2^b).
//Foe example: 1 << 2 => 00000000 00000001 => 00000000 000000100, 1 = 1 * 2^2.

//Since the logical addresses in addresses.txt are all decimal values,
//we need to convert them to binary value first,
//then the indexes 0 - 7 is the offset, and indexes 8 - 15 is page number.
//The maximal logical address is 2^16 = 65536,
//so we only need to consider the first 16 bits in binary representation.
#define PAGE_SIZE (1 << OFFSET_BITS)
#define FRAME_SIZE PAGE_SIZE
#define TLB_SIZE 16
#define NUMBER_OF_PAGES (1 << PAGE_BITS)
#define NUMBER_OF_FRAMES (1 << FRAME_BITS)
#define MEMORY_SIZE (FRAME_SIZE * NUMBER_OF_FRAMES)

//Macros functions.
#define GET_PAGE_NUMBER(address) ((address & ((1 << (OFFSET_BITS + PAGE_BITS))- (1 << OFFSET_BITS))) >> OFFSET
#define GET_OFFSET(address) (address & ((1 << OFFSET_BITS) - 1))
#define GET_PHYSICAL_ADDRESS(frame, offset) ((frame_number << OFFSET_BITS) | offset)

#define BACKING_STORAGE_FILE "BACKING_STORE.bin"
```

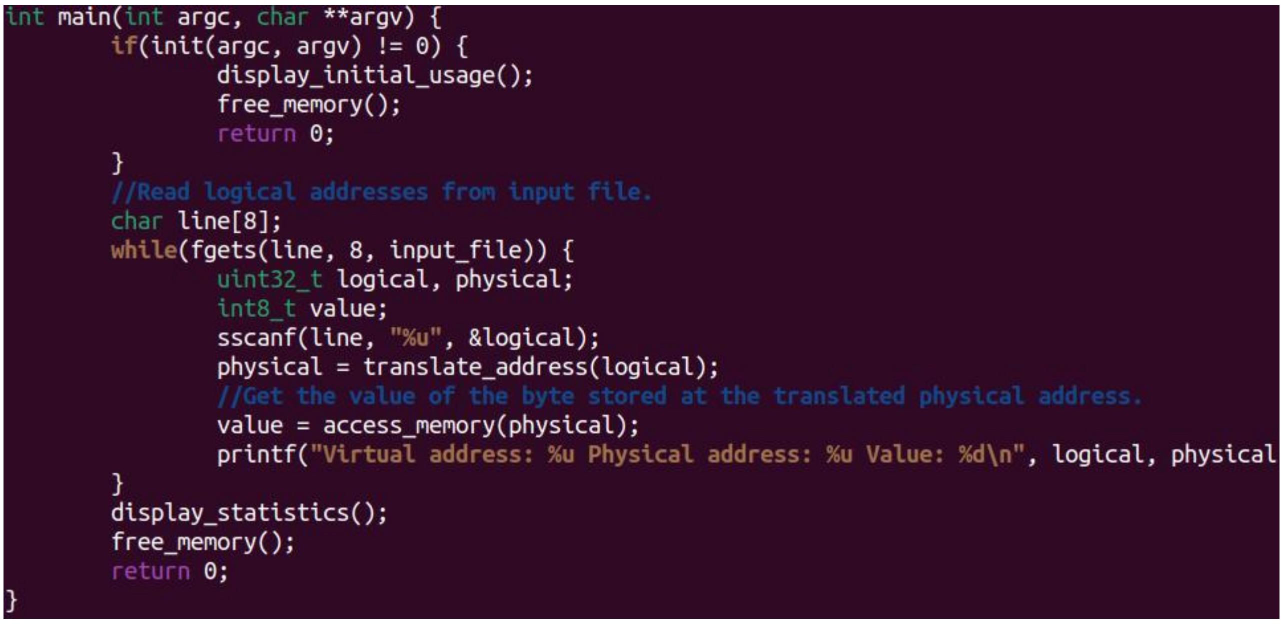
1) Using Macros Functions in C language is easier to implement other functions.
2) "<<" in C is Left Shift for binary number. In decimal: a = a << b => a = a * (2^b).
   Foe example: 1 << 2 => 00000000 00000001 => 00000000 000000100,
   and 1 = 1 * 2^2.
3) Since the logical addresses in “addresses.txt” are all decimal values, we need to convert them to binary value first, then the indexes of the binary representation of the corresponding logical addresses: 0 - 7 is the offset, and 8 - 15 is page number.
4) The maximal logical address is 2^16 = 65536, so we only need to consider the first 16 bits in binary representation.

Functions:


```
typedef struct {
        uint8_t valid;
        uint32_t page, frame;
} Pair;
Pair TLB[TLB_SIZE];

//Global Variables.
int8_t memory[MEMORY_SIZE];
uint32_t page_table[NUMBER_OF_PAGES], next_available_frame, next_available_TLB;
uint8_t page_valid[NUMBER_OF_PAGES];
FILE *backing_storage, *input_file;
//For calculating Page-Fault rate and TLB-Hit rate.
uint32_t count_total, count_page_fault, count_TLB_miss;

//Initialization.
int init(int argc, char **argv);

//Translate logical address to its corresponding physical address.
uint32_t translate_address(uint32_t logical);
//Check whether the current page_number is inside the TLB.
int check_TLB(uint32_t page_number, uint32_t *frame_number);
//We need to update the current TLB once we get a TLB-miss.
void update_TLB(uint32_t page_number, uint32_t frame_number);
//When a Page-Fault occurs, we will read in a 256-byte page from the file
//BACKING_STORE and store it in an available page frame in physical memory.
void handle_page_fault(uint32_t page_number);
//Part of Handle Page-Fault.
uint32_t select_victim_frame();

//Get the value of the byte stored at the translated physical address.
char access_memory(uint32_t physical);
//Display the compile usage of this program.
void display_initial_usage();
//Display the Page-Fault rate and TLB-Hit rate.
void display_statistics();
```

1) The main goal of this program is to translate logical address to physical address. The “Translation” is implemented by “translate_address()”.
2) For translation, we need to deal with “Page Number” and “Offset”, in order to get these variable, we need “Page Table” and “TLB”. “TLB” is implemented by a structure “Pair”, and both of them are global variables.
3) After we get physical address, we want to get the value of byte stored at the corresponding address, then we need to access the binary file “BACKING_STORE.bin” and get the value at the address.

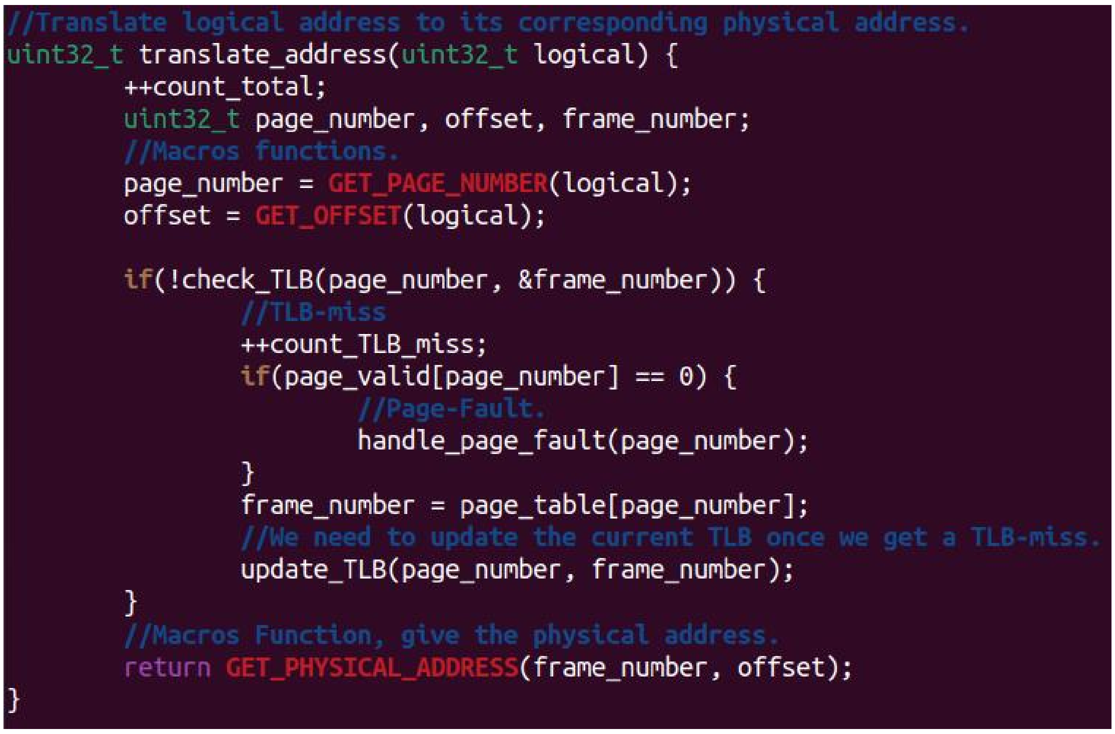
Main Function:



```
int main(int argc, char **argv) {
        if(init(argc, argv) != 0) {
                display_initial_usage();
                free_memory();
                return 0;
        }
        //Read logical addresses from input file.
        char line[8];
        while(fgets(line, 8, input_file)) {
                uint32_t logical, physical;
                int8_t value;
                sscanf(line, "%u", &logical);
                physical = translate_address(logical);
                //Get the value of the byte stored at the translated physical address.
                value = access_memory(physical);
                printf("Virtual address: %u Physical address: %u Value: %d\n", logical, physical
        }
        display_statistics();
        free_memory();
        return 0;
}
```

1) The main function read the logical addresses from the input file “addresses.txt” first.
2) Then, for each logical address, it uses “translate_address()” to translate it to physical address. After we get physical address, it will access the backing storage to get the value of byte stored at it.

Translate:

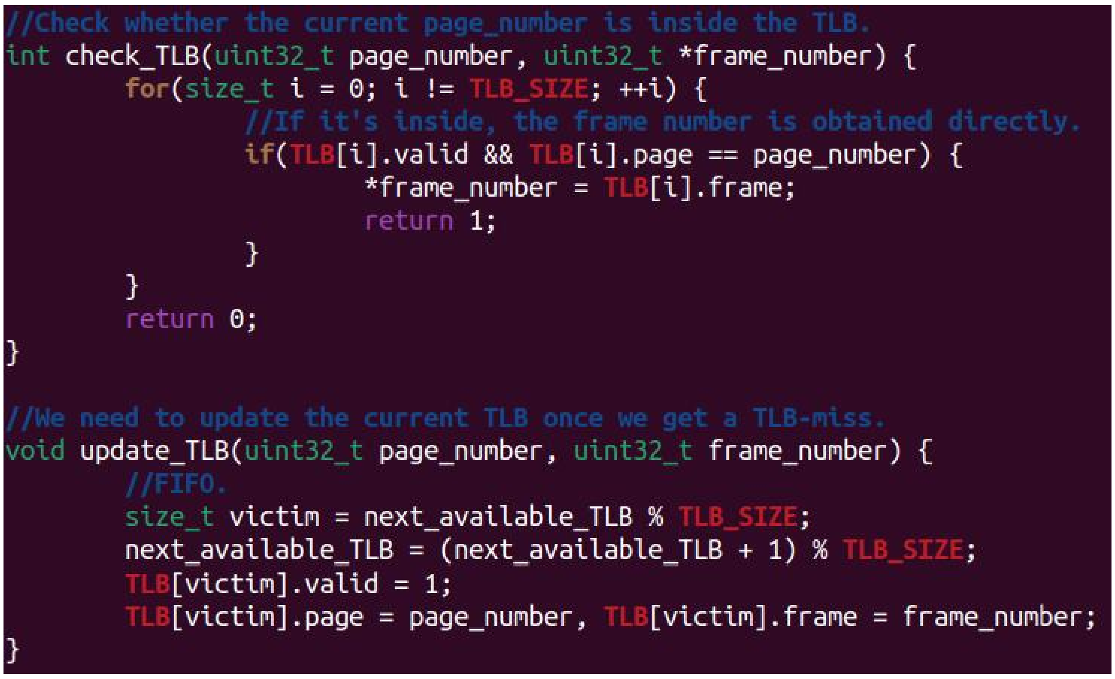


```
//Translate logical address to its corresponding physical address.
uint32_t translate_address(uint32_t logical) {
        ++count_total;
        uint32_t page_number, offset, frame_number;
        //Macros functions.
        page_number = GET_PAGE_NUMBER(logical);
        offset = GET_OFFSET(logical);

        if(!check_TLB(page_number, &frame_number)) {
                //TLB-miss
                ++count_TLB_miss;
                if(page_valid[page_number] == 0) {
                        //Page-Fault.
                        handle_page_fault(page_number);
                }
                frame_number = page_table[page_number];
                //We need to update the current TLB once we get a TLB-miss.
                update_TLB(page_number, frame_number);
        }
        //Macros Function, give the physical address.
        return GET_PHYSICAL_ADDRESS(frame_number, offset);
}
```

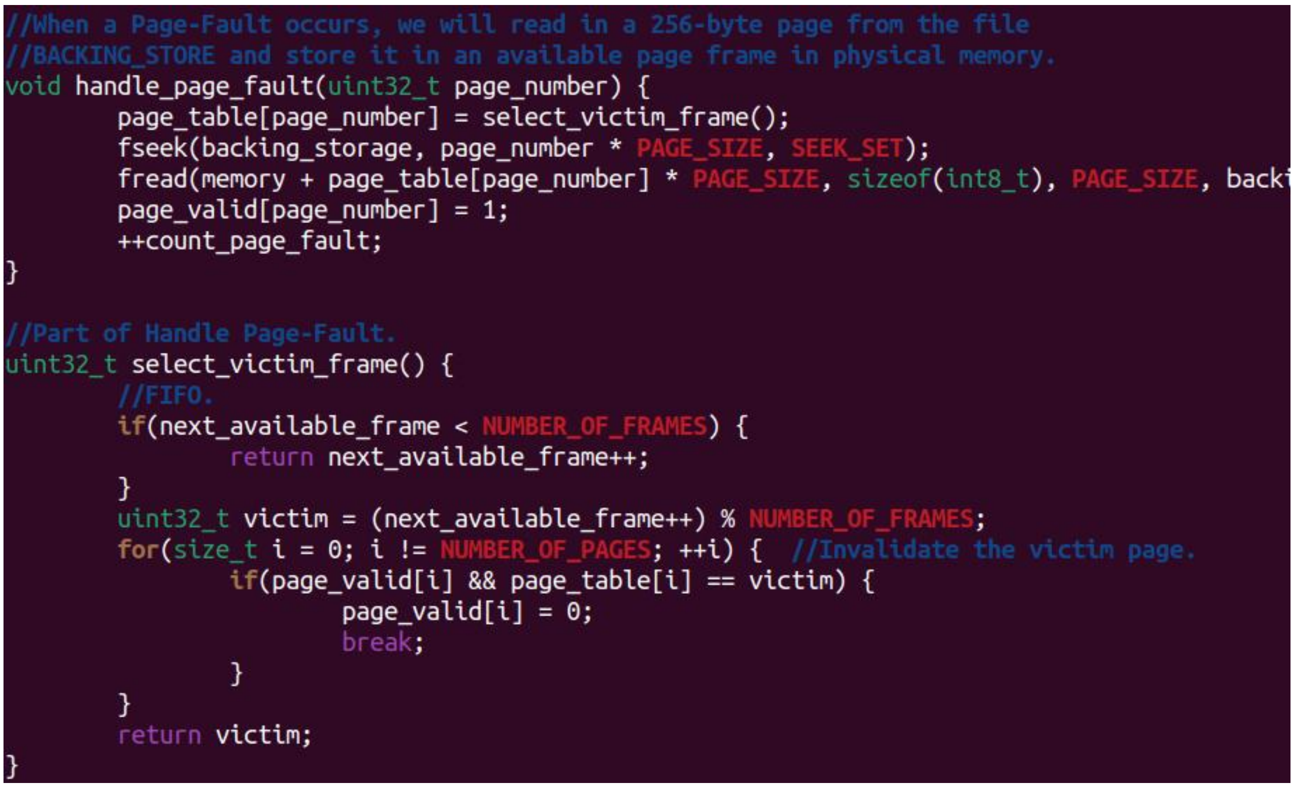
1) This function will check whether the “Page Number” is inside the TLB or not.
2) If it’s inside, we can get the “Frame Number” from the TLB directly. If it’s not

inside, it's a "TLB-Miss", and we will consider whether there is also a "Page Fault", if there is, we need to handle it, and after that, we also need to update the current TLB in order to increase the rate of "TLB-Hit".

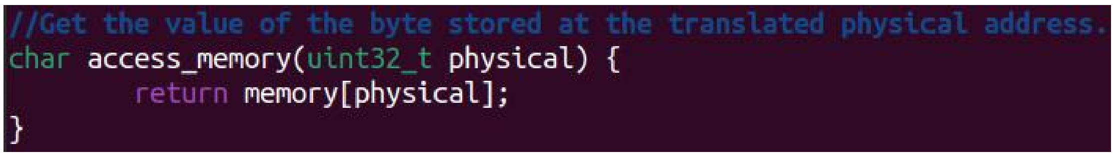
TLB:


```
//Check whether the current page_number is inside the TLB.
int check_TLB(uint32_t page_number, uint32_t *frame_number) {
        for(size_t i = 0; i != TLB_SIZE; ++i) {
                //If it's inside, the frame number is obtained directly.
                if(TLB[i].valid && TLB[i].page == page_number) {
                        *frame_number = TLB[i].frame;
                        return 1;
                }
        }
        return 0;
}

//We need to update the current TLB once we get a TLB-miss.
void update_TLB(uint32_t page_number, uint32_t frame_number) {
        //FIFO.
        size_t victim = next_available_TLB % TLB_SIZE;
        next_available_TLB = (next_available_TLB + 1) % TLB_SIZE;
        TLB[victim].valid = 1;
        TLB[victim].page = page_number, TLB[victim].frame = frame_number;
}
```

Handle Page Fault:


```
//When a Page-Fault occurs, we will read in a 256-byte page from the file
//BACKING_STORE and store it in an available page frame in physical memory.
void handle_page_fault(uint32_t page_number) {
        page_table[page_number] = select_victim_frame();
        fseek(backing_storage, page_number * PAGE_SIZE, SEEK_SET);
        fread(memory + page_table[page_number] * PAGE_SIZE, sizeof(int8_t), PAGE_SIZE, back
        page_valid[page_number] = 1;
        ++count_page_fault;
}

//Part of Handle Page-Fault.
uint32_t select_victim_frame() {
        //FIFO.
        if(next_available_frame < NUMBER_OF_FRAMES) {
                return next_available_frame++;
        }
        uint32_t victim = (next_available_frame++) % NUMBER_OF_FRAMES;
        for(size_t i = 0; i != NUMBER_OF_PAGES; ++i) {  //Invalidate the victim page.
                if(page_valid[i] && page_table[i] == victim) {
                        page_valid[i] = 0;
                        break;
                }
        }
        return victim;
}
```

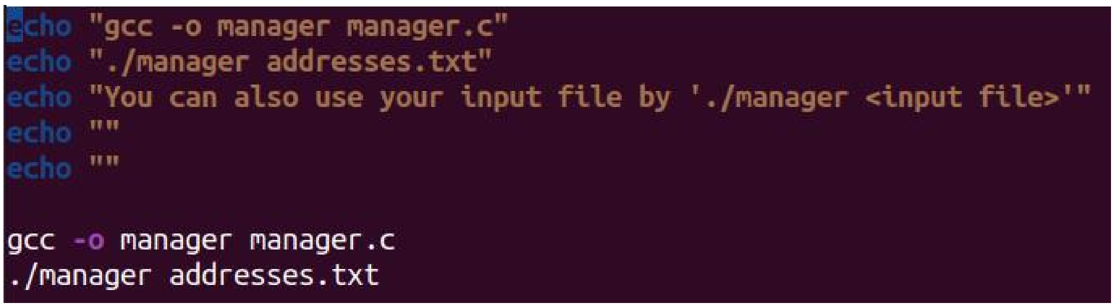
Get Value at the Physical Address:



```c
//Get the value of the byte stored at the translated physical address.
char access_memory(uint32_t physical) {
        return memory[physical];
}
```

After we get physical address, we want to get the value of byte stored at the translated address, then we need to access the binary file “BACKING_STORE.bin” and get the value at the address.

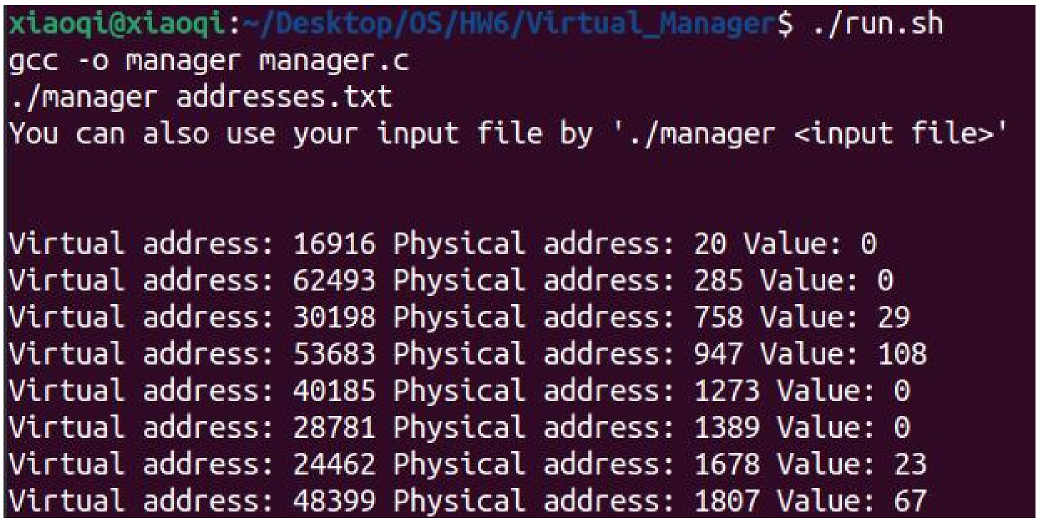
Compile (run.sh):



```sh
echo "gcc -o manager manager.c"
echo "./manager addresses.txt"
echo "You can also use your input file by './manager <input file>'"
echo ""
echo ""

gcc -o manager manager.c
./manager addresses.txt
```

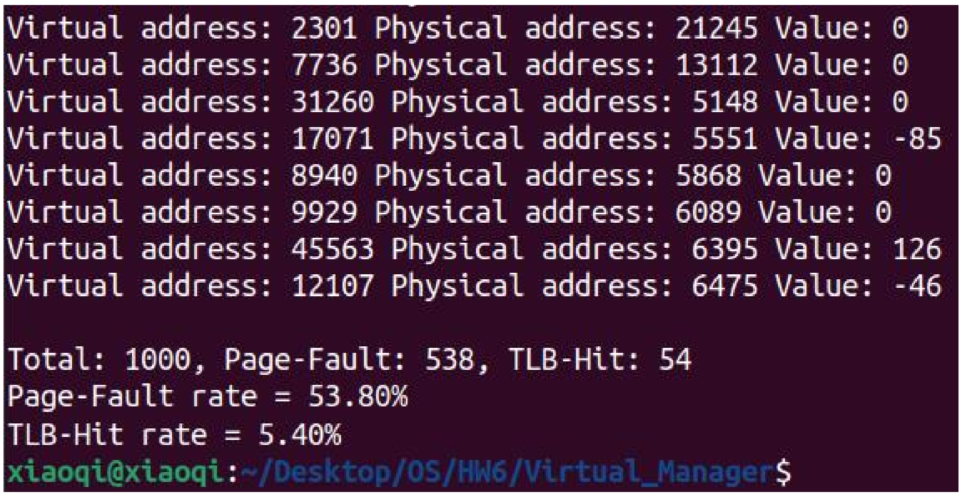
If you run “run.sh” program directly, I set the default input file that contains logical addresses to “addresses.txt”, you can also use your argument of an input file you want.

Results:



```
xiaoqi@xiaoqi:~/Desktop/OS/HW6/Virtual_Manager$ ./run.sh
gcc -o manager manager.c
./manager addresses.txt
You can also use your input file by './manager <input file>'


Virtual address: 16916 Physical address: 20 Value: 0
Virtual address: 62493 Physical address: 285 Value: 0
Virtual address: 30198 Physical address: 758 Value: 29
Virtual address: 53683 Physical address: 947 Value: 108
Virtual address: 40185 Physical address: 1273 Value: 0
Virtual address: 28781 Physical address: 1389 Value: 0
Virtual address: 24462 Physical address: 1678 Value: 23
Virtual address: 48399 Physical address: 1807 Value: 67
```

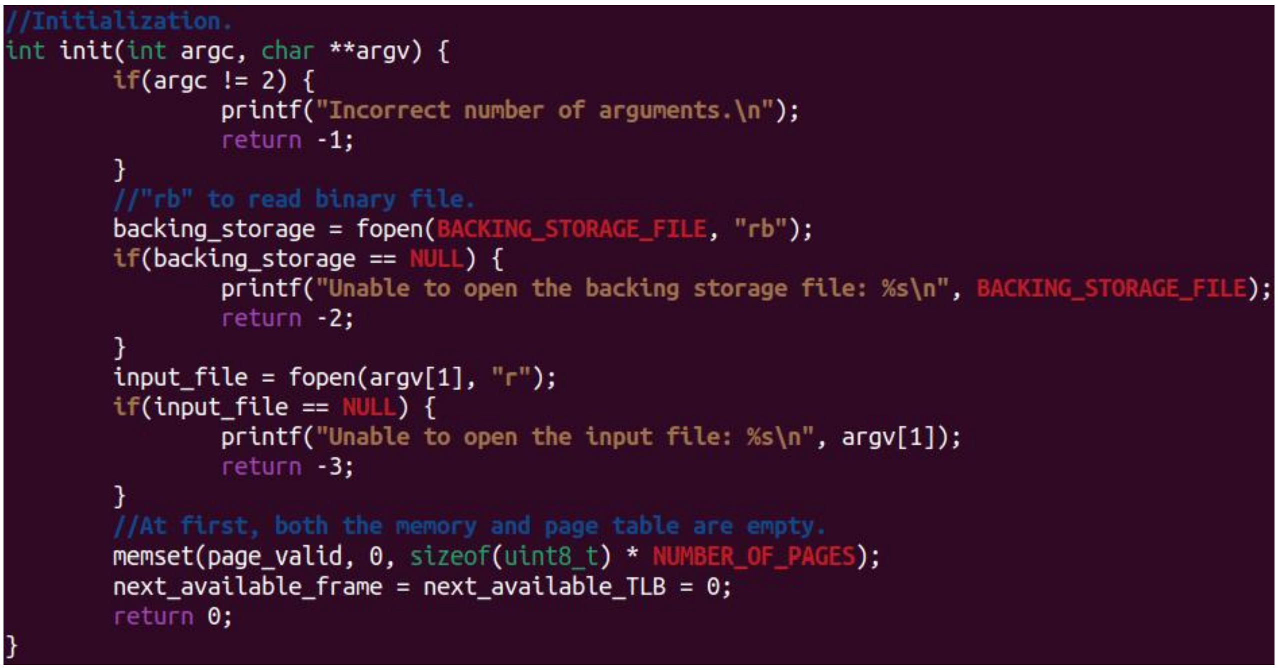
---Other 1000 results of logical addresses.---



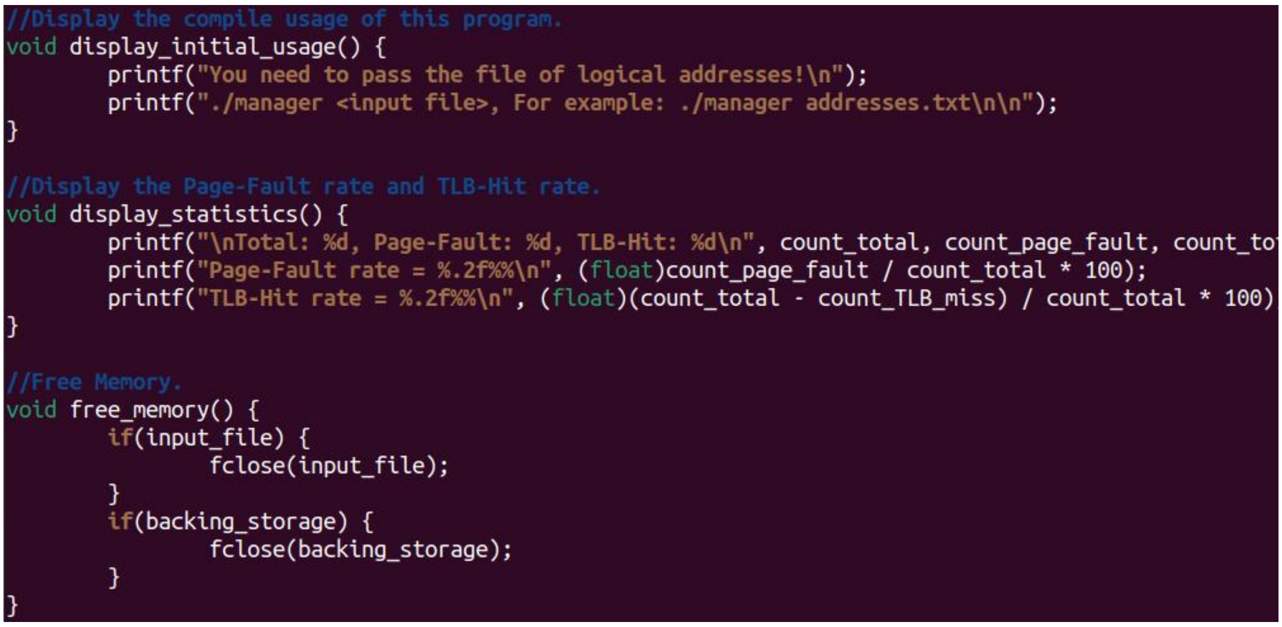
```
Virtual address: 2301 Physical address: 21245 Value: 0
Virtual address: 7736 Physical address: 13112 Value: 0
Virtual address: 31260 Physical address: 5148 Value: 0
Virtual address: 17071 Physical address: 5551 Value: -85
Virtual address: 8940 Physical address: 5868 Value: 0
Virtual address: 9929 Physical address: 6089 Value: 0
Virtual address: 45563 Physical address: 6395 Value: 126
Virtual address: 12107 Physical address: 6475 Value: -46

Total: 1000, Page-Fault: 538, TLB-Hit: 54
Page-Fault rate = 53.80%
TLB-Hit rate = 5.40%
xiaoqi@xiaoqi:~/Desktop/OS/HW6/Virtual_Manager$
```

1) The results are the corresponding physical address and the number of value stored at the physical address in the file BACKING_STORE.bin.
2) After comparing with the resource file "correct.txt", this result is the same with "correct.txt", which means this program works.
3) At the end of this program, it also displays the number of "Page-Fault" and "TLB-Hit", and the rate of them.
4) Since the logical addresses in "addresses.txt" were generated randomly and do not reflect any memory access locality, it's OK that the TLB-Hit Rate here is quite low.

Other Codes:



```
//Initialization.
int init(int argc, char **argv) {
        if(argc != 2) {
                printf("Incorrect number of arguments.\n");
                return -1;
        }
        //"rb" to read binary file.
        backing_storage = fopen(BACKING_STORAGE_FILE, "rb");
        if(backing_storage == NULL) {
                printf("Unable to open the backing storage file: %s\n", BACKING_STORAGE_FILE);
                return -2;
        }
        input_file = fopen(argv[1], "r");
        if(input_file == NULL) {
                printf("Unable to open the input file: %s\n", argv[1]);
                return -3;
        }
        //At first, both the memory and page table are empty.
        memset(page_valid, 0, sizeof(uint8_t) * NUMBER_OF_PAGES);
        next_available_frame = next_available_TLB = 0;
        return 0;
}
```



```
//Display the compile usage of this program.
void display_initial_usage() {
        printf("You need to pass the file of logical addresses!\n");
        printf("./manager <input file>, For example: ./manager addresses.txt\n\n");
}

//Display the Page-Fault rate and TLB-Hit rate.
void display_statistics() {
        printf("\nTotal: %d, Page-Fault: %d, TLB-Hit: %d\n", count_total, count_page_fault, count_to
        printf("Page-Fault rate = %.2f%%\n", (float)count_page_fault / count_total * 100);
        printf("TLB-Hit rate = %.2f%%\n", (float)(count_total - count_TLB_miss) / count_total * 100)
}

//Free Memory.
void free_memory() {
        if(input_file) {
                fclose(input_file);
        }
        if(backing_storage) {
                fclose(backing_storage);
        }
}
```

That’s the end of this report, thank you very much for your attention!
Xiaoqi LIU 999009335