

Report of OS Project 6 & 7 & 8

郭嘉宋 517030910374

Assignment

Project 6 — Banker's Algorithm

For this project, you will write a program that implements the banker's algorithm discussed in Section 8.6.3. Customers request and release resources from the bank. The banker will grant a request only if it leaves the system in a safe state. A request that leaves the system in an unsafe state will be denied. Although the code examples that describe this project are illustrated in C, you may also develop a solution using Java.

The banker will consider requests from n customers for m resources types, as outlined in Section 8.6.3. The banker will keep track of the resources using the following data structures:

```
#define NUMBER OF CUSTOMERS 5
#define NUMBER OF RESOURCES 4
/* the available amount of each resource */
int available[NUMBER OF RESOURCES];
/*the maximum demand of each customer */
int maximum[NUMBER OF CUSTOMERS][NUMBER OF RESOURCES];
/* the amount currently allocated to each customer */
int allocation[NUMBER OF CUSTOMERS][NUMBER OF RESOURCES];
/* the remaining need of each customer */
int need[NUMBER OF CUSTOMERS][NUMBER OF RESOURCES];
```

The banker will grant a request if it satisfies the safety algorithm outlined in Section 8.6.3.1. If a request does not leave the system in a safe state, the banker will deny it. Function prototypes for requesting and releasing resources are as follows:

```
int request resources(int customer num, int request[]);
void release resources(int customer num, int release[]);
```

The request resources() function should return 0 if successful and -1 if unsuccessful.

Project 7 — Contiguous Memory Allocation

In Section 9.2, we presented different algorithms for contiguous memory allocation. This project will involve managing a contiguous region of memory of size MAX where addresses may range from 0 ... MAX - 1. Your program must respond to four different requests:

1. Request for a contiguous block of memory

2. Release of a contiguous block of memory
3. Compact unused holes of memory into one single block
4. Report the regions of free and allocated memory

Your program will allocate memory using one of the three approaches highlighted in Section 9.2.2, depending on the flag that is passed to the RQ command. The flags are:

- F — first fit
- B — best fit
- W — worst fit

This will require that your program keep track of the different holes representing available memory. When a request for memory arrives, it will allocate the memory from one of the available holes based on the allocation strategy. If there is insufficient memory to allocate to a request, it will output an error message and reject the request. Your program will also need to keep track of which region of memory has been allocated to which process. This is necessary to support the STAT The first parameter to the RQ command is the new process that requires the memory, followed by the amount of memory being requested, and finally the strategy. (In this situation, "W" refers to worst fit.) Once your program has started, it will present the user with the following prompt:

```
allocator>
```

Your program will also need to keep track of which region of memory has been allocated to which process. This is necessary to support the STAT command and is also needed when memory is released via the RL command, as the process releasing memory is passed to this command. If a partition being released is adjacent to an existing hole, be sure to combine the two holes into a single hole. If the user enters the C command, your program will compact the set of holes into one larger hole. For example, if you have four separate holes of size 550 KB, 375 KB, 1,900 KB, and 4,500 KB, your program will combine these four holes into one large hole of size 7,325 KB. There are several strategies for implementing compaction, one of which is suggested in Section 9.2.3. Be sure to update the beginning address of any processes that have been affected by compaction.

Project 8 — Designing a Virtual Memory Manager

This project consists of writing a program that translates logical to physical addresses for a virtual address space of size $2^{16} = 65,536$ bytes. Your program will read from a file containing logical addresses and, using a TLB and a page table, will translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address. Your learning goal is to use simulation to understand the steps involved in translating logical to physical addresses. This will include resolving page faults using demand paging, managing a TLB, and implementing a page-replacement algorithm. Your program will read a file containing several 32-bit integer numbers that represent logical addresses. However, you need only be concerned with 16-bit addresses, so you must mask the rightmost 16 bits of each logical address.

- 2^8 entries in the page table
- Page size of 2^8 bytes

- 16 entries in the TLB
- Frame size of 2^8 bytes
- 256 frames
- Physical memory of 65,536 bytes (256 frames \times 256-byte frame size)

Additionally, your program need only be concerned with reading logical addresses and translating them to their corresponding physical addresses. You do not need to support writing to the logical address space. Your program is to output the following values:

- The logical address being translated (the integer value being read from addresses.txt).
- The corresponding physical address (what your program translates the logical address to).
- The signed byte value stored in physical memory at the translated physical address.

After completion, your program is to report the following statistics:

- Page-fault rate—The percentage of address references that resulted in page faults.
- TLB hit rate—The percentage of address references that were resolved in the TLB.

Environment

- Virtual platform: VMware Workstation Pro 15.0
- Operating system: Ubuntu 18.04
- Kernel: Linux 5.3.1
- Local system: Windows 10

Solution

Project 6:

I use C to implement this project. Input list shows the available resources. The key function is `whether_safe()` which is to judge whether this allocation is a safe state. I implement this function as follows:

```
// 1 if safe, 0 if unsafe
int whether_safe() {
    int work[NUMBER_OF_RESOURCES];
    int finish[NUMBER_OF_CUSTOMERS] = {0};
    int flag = 0;
    int finished = 0;
    int i, j;
    int judge;
    copy_vector(work, available);
    while (1) {
        for (i = 0; i < NUMBER_OF_CUSTOMERS; i++) {
            if (finish[i] == 0 && less_or_equal(need[i], work)) {
                flag = 1;
                add_or_sub(work, allocation[i], 1);
                finish[i] = 1;
                finished++;
            }
        }
    }
}
```

```

    }
    if (!flag) {
        judge = 1;
        for (j = 0; j < NUMBER_OF_CUSTOMERS; j++) {
            if (finish[j] == 0)
                judge = 0;
        }
        if (!judge)
            return 0;
        else
            return 1;
    }
    flag = 0;
}
}

```

- I use the safety algorithm outlined in Section 8.6.3.1:
 1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available and Finish[i] = false for i = 0, 1, ..., n - 1.
 2. Find an index i such that both
 - Finish[i] == false
 - Need[i] ≤ Work If no such i exists, go to step 4.
 3. Work = Work + Allocation[i] Finish[i] = true Go to step 2.
 4. If Finish[i] == true for all i, then the system is in a safe state
- This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

```

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

#define NUMBER_OF_CUSTOMERS 5
#define NUMBER_OF_RESOURCES 4

/* the available amount of each resource */
int available[NUMBER_OF_RESOURCES];
/*the maximum demand of each customer */
int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
/* the amount currently allocated to each customer */
int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
/* the remaining need of each customer */
int need[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

void copy_vector(int *new_vec, int *old_vec){
    int i;
    for(i=0; i<NUMBER_OF_RESOURCES; i++)
        new_vec[i] = old_vec[i];
}

void add_or_sub(int *old_vec, int *new_vec, int sign){

```

```
//sign is +1 then add, sign is -1 then sub
int i;
for(i=0; i<NUMBER_OF_RESOURCES; i++)
    old_vec[i] += new_vec[i] * sign;
}

// compare vector
int less_or_equal(int *left, int *right){
    int i;
    int flag = 0;
    for(i=0; i<NUMBER_OF_RESOURCES; i++)
        flag += (left[i] <= right[i]);
    return (flag == NUMBER_OF_RESOURCES);
}

// 1 if safe, 0 if unsafe
int whether_safe() {
    int work[NUMBER_OF_CUSTOMERS];
    int finish[NUMBER_OF_RESOURCES] = {0};
    int flag = 0;
    int finished = 0;
    int i, j;
    int judge;
    copy_vector(work, available);
    while (1) {
        for (i = 0; i < NUMBER_OF_CUSTOMERS; i++) {
            if (finish[i] == 0 && less_or_equal(need[i], work)) {
                flag = 1;
                add_or_sub(work, allocation[i], 1);
                finish[i] = 1;
                finished++;
            }
        }
        if (!flag) {
            judge = 1;
            for (j = 0; j < NUMBER_OF_CUSTOMERS; j++) {
                if (finish[j] == 0)
                    judge = 0;
            }
            if (!judge)
                return 0;
            else
                return 1;
        }
        flag = 0;
    }
}

// update resources
void update(int customer_num, int release[], int sign){
    add_or_sub(available, release, sign);
    add_or_sub(allocation[customer_num], release, -sign);
}
```

```

    add_or_sub(need[customer_num], release, sign);
}

// request resources
int request_resources(int customer_num, int request[]){
    int flag;
    if(!less_or_equal(request, need[customer_num]))
    {
        printf("Denied, request exceeds need\n");
        return -1;
    }
    if(!less_or_equal(request, available))
    {
        printf("Denied, request larger than available\n");
        return -1;
    }
    update(customer_num, request, -1);
    if(whether_safe()) return 0;
    update(customer_num, request, 1);
    printf("Denied, it will cause deadlock\n");
    return -1;
}

// release resources
void release_resources(int customer_num, int release[]){
    update(customer_num, release, 1);
}

// print a vector
void print_vec(int v[NUMBER_OF_RESOURCES])
{
    int i;
    for(i=0; i<NUMBER_OF_RESOURCES; i++)
        printf("%d ", v[i]);
    printf("\n");
}

// print a matrix
void print_matrix(int m[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES])
{
    int i, j;
    for(i=0; i<NUMBER_OF_CUSTOMERS; i++)
    {
        for(j=0; j<NUMBER_OF_RESOURCES; j++)
            printf("%d ", m[i][j]);
        printf("\n");
    }
}

int main(int argc, char **args){
    FILE *file = fopen("max_table.txt", "r");

    int i, j;
    int request[NUMBER_OF_RESOURCES];

```

```

char s[3];
for(i=0; i<NUMBER_OF_CUSTOMERS; i++){
    for(j=0; j<NUMBER_OF_RESOURCES; j++){
        fscanf(file, ((j==NUMBER_OF_RESOURCES-1)? "%d\n": "%d, "),
            &maximum[i][j]);

        allocation[i][j] = 0;
        need[i][j] = maximum[i][j];

    }
}
fclose(file);
for(j=0; j<NUMBER_OF_RESOURCES; j++)
    available[j] = ((j<argc-1)?atoi(args[j+1]):0);
while(1){
    scanf("%s", s);
    if(s[0] == '*'){
        printf("Available:\n");
        print_vec(available);
        printf("Maximum matrix:\n");
        print_matrix(maximum);
        printf("Allocation matrix:\n");
        print_matrix(allocation);
        printf("Need matrix:\n");
        print_matrix(need);
    }
    else{
        scanf("%d", &j);
        for(i=0; i<NUMBER_OF_RESOURCES; i++)
            scanf("%d", &request[i]);
        if(s[1] == 'Q' || s[1] == 'q'){
            if(request_resources(j, request) == 0){
                printf("Request satisfied\n");
            }
        }
        else if(s[1] == 'L' || s[1] == 'l'){
            if(less_or_equal(request, allocation[j])){
                release_resources(j, request);
                printf("Resource released\n");
            }
            else{
                printf("Denied, release exceeds allocation\n");
            }
        }
    }
}
return 0;
}

```

- The implementation of banker's algorithm are showed above.
- Remember to deny the case that the release sources are more than the allocation sources.
- Deny the case that the request is larger than need or available.

- I use whether_safety() function to avoid deadlock.
 - I read init-max matrix from "max_table.txt"
 - Whole source code is in the package.
-

Project 7

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

#define NAME_LENGTH 32

typedef struct BLOCK{
    int low;
    int high;
    char name[NAME_LENGTH];
    struct BLOCK *next;
}block;

int size(block *node)
{
    return node->high - node->low + 1;
}

block *head;

void init(int size){
    head = (block*)malloc(sizeof(block));
    block *node = (block*)malloc(sizeof(block));
    node->low = 0;
    node->high = size-1;
    strcpy(node->name, "Unused");
    node->next = NULL;
    head->next = node;
}

//delete a node
void delete_node(block **node){
    block *tmp;
    tmp = (*node)->next;
    (*node)->next = tmp->next;
    free(tmp);
}

void free_list(){
    while (head->next != NULL)
    {
        delete_node(&head);
    }
}
```



```

    }
    free(head);
}

void status(){
    block *p = head->next;
    while (p != NULL)
    {
        if(p->next == NULL && p->name[0] == 'U'){
            printf("Addresses [%d] . . .\n", p->low);
            break;
        }
        else{
            printf("Addresses [%d:%d] %s\n", p->low, p->high, p->name);
            p = p->next;
        }
    }
}

// request memory
// return 0 if succeed, otherwise return -1
int request(char name[], int request_size, char choice) {
    block *p;
    // target->next is the space to insert
    block *target;
    switch (choice) {
        // worst fit
        case 'W':
            target = NULL;
            p = head;
            while (p->next != NULL) {
                if (p->next->name[0] == 'U'
                    && size(p->next) >= request_size &&
                    (target == NULL ||
                     size(p->next) > size(target->next))) {
                    target = p;
                }
                p = p->next;
            }
            break;
        // best fit
        case 'B':
            target = NULL;
            p = head;
            while (p->next != NULL) {
                if (p->next->name[0] == 'U' &&
                    size(p->next) >= request_size &&
                    (target == NULL ||
                     size(p->next) < size(target->next))) {
                    target = p;
                }
                p = p->next;
            }
            break;
    }
}

```

```

        // first fit
        case 'F':
            target = head;
            while (target->next != NULL &&
                    (target->next->name[0] != 'U' ||
                     size(target->next) < request_size))
                target = target->next;
            break;
        default:
            printf("Error! Enter again please\n");
            return -1;
    }
    // failed
    if(target == NULL || target->next == NULL)
//    if (target == NULL) {
        printf("Memory overflow\n");

        return -1;
    }
    else{
        // succeed and just fit
        if (size(target->next) == request_size) {
            strcpy(target->next->name, "Process ");
            strcat(target->next->name, name);
        }
        // succeed but need to split
        else {
            p = (block *) malloc(sizeof(block));
            p->low = target->next->low;
            p->high = p->low + request_size - 1;
            target->next->low += request_size;
            strcpy(p->name, "Process ");
            strcat(p->name, name);
            p->next = target->next;
            target->next = p;
        }
        return 0;
    }
}

// release memory
// return 0 if succeed, otherwise return -1
int release(char process_name[]){
    char name[64] = "Process ";
    block *p = head;
    strcat(name, process_name);
    while(p->next != NULL){
        if(!strcmp(p->next->name, name)){
            int flag = 1;
            // merge backward
            if(p->next->next != NULL && p->next->next->name[0] == 'U')
            {
                p->next->next->low = p->next->low;
                delete_node(&p);
            }
        }
    }
}

```

```

        flag = 0;
    }
    // merge forward
    if(p != head && p->name[0] == 'U'){
        p->high = p->next->high;
        delete_node(&p);
        flag = 0;
    }
    // rename
    if(flag){
        strcpy(p->next->name, "Unused");
    }
    return 0;
}
p = p->next;
}
printf("Process %s is not found\n", process_name);
}

// compact
void compact(){
    int offset = 0;
    block *p = head;
    while (p->next != NULL)
    {
        if(p->next->name[0] == 'U')
        {
            // update last hole
            if(p->next->next == NULL){
                p->next->low -= offset;
                break;
            }
            // delete hole
            else{
                offset += size(p->next);
                delete_node(&p);
            }
        }
        // update address
        else
        {
            p = p->next;
            p->low -= offset;
            p->high -= offset;
        }
    }
}

int main(int argc, char **args){
    int size = 0;
    char command[64];
    char name[NAME_LENGTH];
    char choice[3];
    if(argc > 1){

```

```

        size = atoi(args[1]); //change string to int
    }
    init(size);
    while (1)
    {
        printf("allocator > ");
        scanf("%s", command);
        if(command[0] == 'X') break;
        if (command[0] == 'C')
        {
            // compact
            compact();
            break;
        }
        switch (command[1])
        {
            case 'L':
                scanf("%s", command);
                release(command);
                break;
            // request
            case 'Q':
                scanf("%s%d%s", name, &size, choice);
                request(name, size, choice[0]);
            //if entering more, we take the first
                break;
            // print out
            default:
                status();
                break;
        }
    }
    return 0;
}

```

- I use linklist to implement it because delete and add command are frequent.
- I use "low", "high", "name" to present a block of memory.
- Worst fit just finds the maximum free Memory and best fit finds the minimum.
- "target" is the location where to insert a new node.
- Note that using 'strcat' will make the output of process name become easier.
- Remember to deny the case that the releasing process does not exist.
- Whole source code is in the package.

Project 8

```

# define mask = 255
// address >> 8 can take the 15 - 8 bit of address.
int get_page(int address) {

```

```

    return (address>>8) & mask;
}

int get_offset(int address) {
    return (address) & mask;
}

```

- First, we use right shift to get page and offset.
- "address>>8" can take 15 - 8 bits of address.

```

while (fscanf(address_file, "%d", &address) != EOF)
{
    int page = get_page(address);
    int frame = table(page, mod, &page_fault, &tlb_hit,
        backing_store, address_count);
    int offset = get_offset(address);
    address_count++;
    int physical_address = (frame<<8)+offset;
    printf("Virtual Address: %d Physical Address: %d Value: %d\n",
        address, physical_address, get_content(physical_address));
}

fclose(address_file);
fclose(backing_store);

```

- Then we use 'fscanf' to read address from 'addresses.txt'.
- 'address_count' is to record how many addresses are read.

```

nt table(int page, int mod, int *page_fault, int *tlb_hit,
FILE *backing_store, int time) {
    int frame = search_tlb(page);
    if (frame < 0) {
        if (valid[page] == 1) {
            frame = search_page_table(page);
            used_freq[page]++;
        }
        else {
            (*page_fault)++;
            if (free_mem < FRAME_NUM) {
                // enough space in memory
                this page
                frame = free_mem;
                page_table[page] = free_mem++;
                swap_frame_in_mem(backing_store, page, frame);
            }
            else {
                // not enough space in memory
                int victim_page = select_victim(mod);
                frame = page_table[victim_page];
            }
        }
    }
}

```

```

        valid[victim_page] = 0;
        swap_frame_in_mem(backing_store, page, frame);
        used_freq[victim_page] = 0;
        load_time[victim_page] = -1;
        used_freq[page] = 1;
        load_time[page] = time;
    }
    valid[page] = 1;
}
update_TLB(page, frame, mod, time);
}
else {
    tlb_freq[page]++;
    (*tlb_hit)++;
}
return frame;

```

- We choose a victim page to be replaced if memory is full.
- 'mod' is a control variable, 0 is 'LRU' and '1' is FIFO.

```

void swap_frame_in_mem(FILE *backing_store, int page, int frame)
{
    fseek(backing_store, page*FRAME_SIZE, SEEK_SET);
    fread(memory+frame*FRAME_SIZE, FRAME_SIZE, 1, backing_store);
}

```

- I use 'fseek' and 'fread' to read and change item between 'backing_store' and physical memory.

```

void update_TLB(int page, int frame, int mod, int time) {
    int pos = 0;
    switch (mod)
    {
        case 0:
            for (int i = 1; i < TLB_SIZE; ++i)
                if (tlb_freq[i] < tlb_freq[pos] && page_table[i] != -1)
                    pos = i;
            break;

        case 1:
            for (int i = 1; i < TLB_SIZE; ++i)
                if (tlb_instime[i] < tlb_instime[pos] && page_table[i] != -1)
                    pos = i;

            default:
                break;
    }
    tlb[pos].page = page;
    tlb[pos].frame = frame;
    tlb_freq[pos] = 1;
}

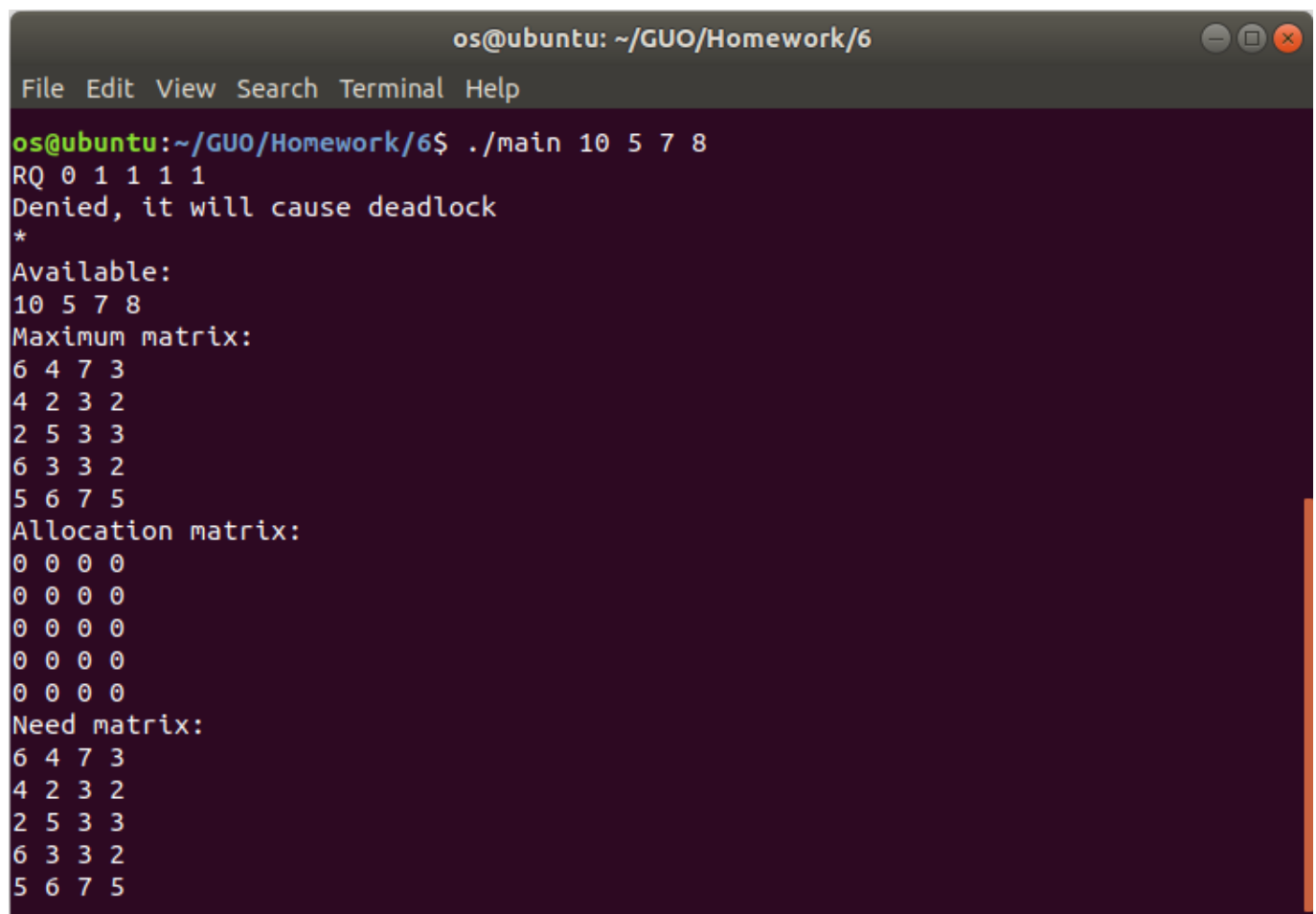
```

```
    tlb_instime[pos] = time;
}
```

- 'tlb_freq' is used to record frequency for LRU.
- 'tlb_instime' is used to record instime for FIFO.
- Whole source code is in the package, other part are relatively easier to implement.

Experiment Result

Chapter 6:



```
os@ubuntu: ~/GUO/Homework/6
File Edit View Search Terminal Help
os@ubuntu:~/GUO/Homework/6$ ./main 10 5 7 8
RQ 0 1 1 1 1
Denied, it will cause deadlock
*
Available:
10 5 7 8
Maximum matrix:
6 4 7 3
4 2 3 2
2 5 3 3
6 3 3 2
5 6 7 5
Allocation matrix:
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
Need matrix:
6 4 7 3
4 2 3 2
2 5 3 3
6 3 3 2
5 6 7 5
```

- RQ 0 1 1 1 1 is denied because it will cause deadlock.
- Maximum matrix, allocation matrix and need matrix are showed successfully with command *.

```
os@ubuntu: ~/GUO/Homework/6
File Edit View Search Terminal Help
os@ubuntu:~/GUO/Homework/6$ ./main 10 10 7 7
RQ 2 5 5 5 5
Denied, request exceeds need
RQ 0 1 1 1 1
Request satisfied
RL 0 1 1 1 1
Resource released
*
Available:
10 10 7 7
Maximum matrix:
6 4 7 3
4 2 3 2
2 5 3 3
6 3 3 2
5 6 7 5
Allocation matrix:
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
Need matrix:
6 4 7 3
4 2 3 2
2 5 3 3
6 3 3 2
5 6 7 5
```

- We now give available to 10 10 7 7.
- RQ 2 5 5 5 5 is denied because it exceeds need.
- RQ 0 1 1 1 1 is satisfied.
- RL 0 1 1 1 1 releases allocation successfully.
- The Banker's Algorithm is implement perfectly.

Project 7:


```

os@ubuntu:~/GU0/Homework/7$ ./main 100
allocator > RQ P1 5 W
allocator > STAT
Addresses [0:4] Process P1
Addresses [5] . . .
allocator > RQ P2 10 F
allocator > RQ P3 15 B
allocator > RQ P4 15 D
Error! No such flag, please enter W or B or F!
allocator > STAT
Addresses [0:4] Process P1
Addresses [5:14] Process P2
Addresses [15:29] Process P3
Addresses [30] . . .
allocator > RL P2
allocator > STAT
Addresses [0:4] Process P1
Addresses [5:14] Unused
Addresses [15:29] Process P3
Addresses [30] . . .
allocator >

```

- RQ P4 15 D is denied.
- STAT outputs the allocation successfully.
- RQ and RL work well.

```

allocator > RQ P5 5 W
allocator > STAT
Addresses [0:4] Process P1
Addresses [5:14] Unused
Addresses [15:29] Process P3
Addresses [30:34] Process P5
Addresses [35] . . .
allocator > RQ P6 5 B
allocator > STAT
Addresses [0:4] Process P1
Addresses [5:9] Process P6
Addresses [10:14] Unused
Addresses [15:29] Process P3
Addresses [30:34] Process P5
Addresses [35] . . .

```

- worst-fit, best-fit and first-fit work all well.

```

allocator > C
Addresses [0:4] Process P1
Addresses [5:9] Process P6
Addresses [10:24] Process P3
Addresses [25:29] Process P5
Addresses [30] . . .
allocator > X
os@ubuntu:~/GU0/Homework/7$

```

- Compaction is also effective.
- 'Contiguous Memory Allocation' is implemented successfully.

Project 8:

```
os@ubuntu: ~/GUO/Homework/8
File Edit View Search Terminal Help
Virtual Address: 23195 Physical Address: 35995 Value: -90
Virtual Address: 27227 Physical Address: 28763 Value: -106
Virtual Address: 42816 Physical Address: 19520 Value: 0
Virtual Address: 58219 Physical Address: 34155 Value: -38
Virtual Address: 37606 Physical Address: 21478 Value: 36
Virtual Address: 18426 Physical Address: 2554 Value: 17
Virtual Address: 21238 Physical Address: 37878 Value: 20
Virtual Address: 11983 Physical Address: 59855 Value: -77
Virtual Address: 48394 Physical Address: 1802 Value: 47
Virtual Address: 11036 Physical Address: 39964 Value: 0
Virtual Address: 30557 Physical Address: 16221 Value: 0
Virtual Address: 23453 Physical Address: 20637 Value: 0
Virtual Address: 49847 Physical Address: 31671 Value: -83
Virtual Address: 30032 Physical Address: 592 Value: 0
Virtual Address: 48065 Physical Address: 25793 Value: 0
Virtual Address: 6957 Physical Address: 26413 Value: 0
Virtual Address: 2301 Physical Address: 35325 Value: 0
Virtual Address: 7736 Physical Address: 57912 Value: 0
Virtual Address: 31260 Physical Address: 23324 Value: 0
Virtual Address: 17071 Physical Address: 175 Value: -85
Virtual Address: 8940 Physical Address: 46572 Value: 0
Virtual Address: 9929 Physical Address: 44745 Value: 0
Virtual Address: 45563 Physical Address: 46075 Value: 126
Virtual Address: 12107 Physical Address: 2635 Value: -46
TLB hit rate: 0.042000%
Page fault rate: 0.244000%
os@ubuntu:~/GUO/Homework/8$
```

- Virtual address, physical address , TLB hit rate and page fault rate are showed successfully.
- This is using FIFO plan.

```
os@ubuntu: ~/GUO/Homework/8
File Edit View Search Terminal Help
Virtual Address: 23195 Physical Address: 35995 Value: -90
Virtual Address: 27227 Physical Address: 28763 Value: -106
Virtual Address: 42816 Physical Address: 19520 Value: 0
Virtual Address: 58219 Physical Address: 34155 Value: -38
Virtual Address: 37606 Physical Address: 21478 Value: 36
Virtual Address: 18426 Physical Address: 2554 Value: 17
Virtual Address: 21238 Physical Address: 37878 Value: 20
Virtual Address: 11983 Physical Address: 59855 Value: -77
Virtual Address: 48394 Physical Address: 1802 Value: 47
Virtual Address: 11036 Physical Address: 39964 Value: 0
Virtual Address: 30557 Physical Address: 16221 Value: 0
Virtual Address: 23453 Physical Address: 20637 Value: 0
Virtual Address: 49847 Physical Address: 31671 Value: -83
Virtual Address: 30032 Physical Address: 592 Value: 0
Virtual Address: 48065 Physical Address: 25793 Value: 0
Virtual Address: 6957 Physical Address: 26413 Value: 0
Virtual Address: 2301 Physical Address: 35325 Value: 0
Virtual Address: 7736 Physical Address: 57912 Value: 0
Virtual Address: 31260 Physical Address: 23324 Value: 0
Virtual Address: 17071 Physical Address: 175 Value: -85
Virtual Address: 8940 Physical Address: 46572 Value: 0
Virtual Address: 9929 Physical Address: 44745 Value: 0
Virtual Address: 45563 Physical Address: 46075 Value: 126
Virtual Address: 12107 Physical Address: 2635 Value: -46
TLB hit rate: 0.045000%
Page fault rate: 0.244000%
os@ubuntu:~/GUO/Homework/8$
```

- Virtual address, physical address , TLB hit rate and page fault rate are showed successfully.
- This is using LRU plan.
- 'Virtual Memory Manager' is implemented successfully.

Logs

These 3 projects are not easy to finish. I took more than 2 weeks to implement these algorithm and debugged for a very long time. But after finishing these work, I felt satisfied and I hope I can learn more in further study. Finally, thanks teacher and TA for reading my report and code.

517030910374 郭嘉宋