Project 7: Contiguous Memory Allocation

白骐硕 518030910102

任务概述

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:

- Request for a contiguous block of memory
- Release of a contiguous block of memory
- Compact unused holes of memory into one single block
- Report the regions of free and allocated memory

具体实现

内存的数据结构

为了实现本实验的要求，我使用双链表的方式来维护整个Memory。该双链表的每一个node，即内存中的每个block 的属性有：

- name，进程名字（如果为hole，则为一个空指针）
- begin，该block的起始地址
- end，该block的结束地址
- used，指示该block是否被使用，0则表示Hole
- 两个指针，*prev，*next

结构体和相关参数定义如下：

```
typedef struct block{
    char *name;
    int begin;
    int end;
    int used;
    struct block *prev,*next;
}Block;

Block *memoryHead;
Block *memoryTail;
```

Main函数

在main函数需要执行的内容有如下两个部分：

- 获取命令行参数，并根据该参数对内存（双链表）进行初始化
- 获取用户输入的命令，调用相应函数实现命令。（不断重复该过程直到得到退出命令）

```
int main(int argc, char *argv[]){
    //get the commandLine
    if(argc != 2){
        fprintf(stderr, "Usage: ./main X | Example: ./main 1048576 \n");
```

```
        return 0;
    }
    maxMem = atoi(argv[1]);
    //initialize the memory
    initMemory();

    char op[10];
    printf("allocator>");
    while(scanf("%s", op) == 1){
        if(strcmp(op,"RQ")==0){
            //RQ
            request();
        }else if(strcmp(op, "RL")==0){
            //RL
            release();
        }else if(strcmp(op, "C")==0){
            //C
            compact();
        }else if(strcmp(op, "STAT") == 0){
            //STAT
            displayState();
        }else if(strcmp(op, "X") == 0){
            //X
            shutdownMemory(memoryHead);
            break;
        }else{
            printf("Illegal Operation! \n");
        }
        printf("allocator>");
    }

    return 0;
}
```

在这里首先介绍，初始化函数initMemory() 和 销毁函数shutdownMemory(memoryHead)（用于释放内存防止泄露）.

在初始化的结果为，双链表中只有一个block。该block的used=0，是一个Hole，且begin=0，end为最大内存地址-1。

```
void initMemory(){
    memoryHead = malloc(sizeof(Block));
    memoryHead->begin = 0;
    memoryHead->end = maxMem - 1;
    memoryHead->used = 0;
    memoryHead->prev = NULL;
    memoryHead->next = NULL;
    memoryTail = memoryHead;
}
```

销毁函数（释放内存）：

```
void shutdownMemory(Block *head){
    Block *p = head;
    Block *tmp = NULL;
    while(p!=NULL){
        tmp = p;
        p = p->next;
        if(tmp->used == 1){
            free(tmp->name);
        }
        free(tmp);
    }
}
```

RQ指令

main函数，读取到RQ指令后，调用request()函数，该函数的执行逻辑如下：

- 首先读入用户输入参数，进程名，请求内存大小，策略（F|B|W）
- 判断输入是否合法
- 根据用户输入的策略，遍历双链表，寻找目标Hole
    - 如果找到了，则将该进程的地址空间插入到该Hole中，有两种情况：
        - 如果请求空间大小刚好等于该Hole，则将该block的名字改为进程名，used改为1
        - 如果请求空间大小小于该Hole，则将该block拆分为两个block，前一个block分配给该进程，后一个block为剩余的Hole大小
    - 如果没找到目标Hole，则说明没有足够的内存Hole来分配给该进程，向用户输出请求失败。

下面，以first fit 为例，展示部分源代码：

```
switch(strategy){
    case 'F':
        {
            //first fit
            Block *p = memoryHead;
            while(p!=NULL){
                if(p->used == 1){
                    p = p->next;
                    continue;
                }
                if(p->end - p->begin +1 >= requestSize){
                    targetHole = p;
                    break;
                }
                p = p->next;
            }
            break;
        }
     //TODO
}
if(targetHole == NULL){
    printf("No enough memory hole to satisfy the request! \nRequest Denied!
\n");
}else{
    insert2Hole(requestName, requestSize, targetHole);
    printf("Request Success! \n");
}
```

```
void insert2Hole(char *requestName, int requestSize, Block* targetHole){
    int targetHoleSize = targetHole->end - targetHole->begin + 1;
    if(targetHoleSize == requestSize){
        targetHole->name = malloc(sizeof(char) * (strlen(requestName)+1));
        strcpy(targetHole->name, requestName);
        targetHole->used = 1;
    }else{
        Block *tmp = malloc(sizeof(Block));
        tmp->name = malloc(sizeof(char) * (strlen(requestName)+1));
        strcpy(tmp->name, requestName);
        tmp->begin = targetHole->begin;
        tmp->end = tmp->begin + requestSize -1;
        tmp->used = 1;
        targetHole->begin = tmp->end +1;
        tmp->next = targetHole;
        tmp->prev = targetHole->prev;
        if(targetHole->prev == NULL){
            memoryHead = tmp;
            targetHole->prev = tmp;
        }else{
            tmp->prev->next = tmp;
            targetHole->prev = tmp;
        }
    }
}
```

RL指令

main函数读入RL指令后，调用release()函数，该函数的执行逻辑如下：

- 读入参数，需释放的进程名
- 遍历双链表，寻找该进程名对应的block：
    - 如果找到了则将block的name释放，且将used = 0。之后判断该block相邻的两个block是否也为Hole，如果有相邻的Hole，则进行合并。
    - 如果没找到，则报告给用户，没有找到该进程名对应的block，release失败。

具体代码如下：

```
void release(){
    char releaseName[10];
    scanf("%s", releaseName);
    Block *p = memoryHead;
    int success = 0;
    while(p!=NULL){
        if(p->used == 0){
            p = p->next;
            continue;
        }
        if(strcmp(p->name, releaseName) == 0){
            success = 1;
            free(p->name);
            p->used = 0;
            //combine the adjacent hole
            mergeHole(p);
            break;
        }
```

```
        p = p->next;
    }
    if(success == 1){
        printf("Release Success! \n");
    }else{
        printf("No process named %s \nRelease Failed! \n", releaseName);
    }
}
```

其中，mergeHole(p)函数的内部逻辑已经在上述描述中解释清楚，具体代码主要为链表操作，源代码在此不再展示。

C指令

main函数读入C指令后，调用compact()函数，该函数的执行逻辑如下：

- 创建一个新的空双链表
- 遍历之前memory 的双链表，将被占用的内存空间（used=1的block），从地址0开始，连续的插入到新的双链表中。
- 遍历结束之后，向新的双链表的最后插入一个used=0的block（Hole），其大小为内存的全部剩余大小。
- 最后将原有的双链表的内存释放，memoryHead，memoryTail指针重新指向新的双链表

```
void compact(){
    int usedEnd = -1;
    Block *newMemoryHead = NULL;
    Block *p = memoryHead;
    Block *last = NULL;
    while(p!=NULL){
        if(p->used == 0){
            p = p->next;
            continue;
        }
        Block *tmp = malloc(sizeof(Block));
        tmp->name = malloc(sizeof(char) * (strlen(p->name)+1));
        strcpy(tmp->name,p->name);
        tmp->begin = usedEnd+1;
        tmp->end = p->end - p->begin + tmp->begin;
        tmp->used = 1;
        usedEnd = tmp->end;
        //insert to new memory list
        if(last==NULL){
            newMemoryHead = tmp;
            tmp->next = NULL;
            tmp->prev = NULL;
            last = tmp;
        }else{
            last->next = tmp;
            tmp->prev = last;
            tmp->next = NULL;
            last = tmp;
        }
        p = p->next;
    }
    //insert the hole to the tail
    if(usedEnd+1 != maxMem){
        Block *tmp = malloc(sizeof(Block));
```

```
            tmp->begin = usedEnd+1;
            tmp->end = maxMem-1;
            tmp->used = 0;
            if(last==NULL){
                newMemoryHead = tmp;
                tmp->next = NULL;
                tmp->prev = NULL;
                last = tmp;
            }else{
                last->next = tmp;
                tmp->prev = last;
                tmp->next = NULL;
                last = tmp;
            }
        }
        shutdownMemory(memoryHead);
        memoryHead = newMemoryHead;
        memoryTail = last;
        printf("Compact Success! \n");
    }
```

STAT指令

main函数读入到用户的STAT指令后，调用displayState()函数，在该函数中，遍历双链表，打印每个block的内容。

```
void displayState(){
    printf("================================================\n");
    Block *p = memoryHead;
    while(p!=NULL){
        if(p->used == 1){
            //Process
            printf("Addresses [%d:%d] Process %s \n", p->begin, p->end, p-
>name);
        }else{
            ///Unused
            printf("Addresses [%d:%d] Unused \n", p->begin, p->end);
        }
        p = p->next;
    }
    printf("================================================\n");
}
```

实验结果

测试所用的内存大小为200。

首先，向内存中请求4块连续的内存(P0,P1,P2,P3)，大小均为20：

```
qsbai@qsbai-virtual-machine:~/os-prj/7$ ./main 200
allocator>RQ P0 20 F
Request Success!
allocator>RQ P1 20 F
Request Success!
allocator>RQ P2 20 F
Request Success!
allocator>RQ P3 20 F
Request Success!
allocator>STAT
==================================================
Addresses [0:19] Process P0
Addresses [20:39] Process P1
Addresses [40:59] Process P2
Addresses [60:79] Process P3
Addresses [80:199] Unused
==================================================
allocator>
```

然后测试RL指令，依次释放P1,P3，能够成功释放且合并相邻Hole：

```
==================================================
Addresses [0:19] Process P0
Addresses [20:39] Process P1
Addresses [40:59] Process P2
Addresses [60:79] Process P3
Addresses [80:199] Unused
==================================================
allocator>RL P1
Release Success!
allocator>RL P3
Release Success!
allocator>STAT
==================================================
Addresses [0:19] Process P0
Addresses [20:39] Unused
Addresses [40:59] Process P2
Addresses [60:199] Unused
==================================================
```

测试RQ指令的不同strategy，依次使用三种不同的策略，测试RQ：

```
==================================================
Addresses [0:19] Process P0
Addresses [20:39] Unused
Addresses [40:59] Process P2
Addresses [60:199] Unused
==================================================
allocator>RQ P4 5 F
Request Success!
allocator>RQ P5 5 B
Request Success!
allocator>RQ P6 5 W
Request Success!
allocator>STAT
==================================================
Addresses [0:19] Process P0
Addresses [20:24] Process P4
Addresses [25:29] Process P5
Addresses [30:39] Unused
Addresses [40:59] Process P2
Addresses [60:64] Process P6
Addresses [65:199] Unused
==================================================
```

测试C指令，之后退出：

```
==================================================
Addresses [0:19] Process P0
Addresses [20:24] Process P4
Addresses [25:29] Process P5
Addresses [30:39] Unused
Addresses [40:59] Process P2
Addresses [60:64] Process P6
Addresses [65:199] Unused
==================================================
allocator>C
Compact Success!
allocator>STAT
==================================================
Addresses [0:19] Process P0
Addresses [20:24] Process P4
Addresses [25:29] Process P5
Addresses [30:49] Process P2
Addresses [50:54] Process P6
Addresses [55:199] Unused
==================================================
allocator>X
qsbai@qsbai-virtual-machine:~/os-prj/7$
```