Proj7 Contiguous Memory Allocation

by 郑航 520021911347

1 Abstract

实现了一个连续内存分配的模拟程序

功能：管理内存，进行内存的连续分配和回收，碎片合并，打印当前内存分配状态等

2 Requirements

This project will involve managing a contiguous region of memory of size MAX where addresses may range from 0 ... MAX - 1. The 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

3 Implementation

本次project主要分以下六部分进行实现：

- 数据结构和全局变量的设计
- main函数的设计与实现
- request的实现
- release的实现
- compact的实现

- print_state的实现

其中，main函数是整个程序的框架，其他的部分是对各个命令的分开实现：①request部分实现了命令“RQ”，②release部分实现了命令“RL”，③compact部分实现了命令“C”，④print_state部分实现了命令“STAT”，⑤命令“X” 直接在main中实现即可

3.1 数据结构和全局变量的设计

在本project中，考虑到连续内存分配的进程数不确定，以及需要经常进行合并等操作，利用数组来实现会显得不太灵活，因此决定采用链表的数据结构来表示内存，其中链表的结点定义如下：

```
typedef struct MEM_NODE
{
    int type;    // 1 for allocated memory, 0 for non_allocated memory
    char *name; // process name (if allocated(type==1))
    int begin;  // starting address
    int end;    // ending address
    struct MEM_NODE *next;  //next node
} mem_node;
```

- 由于内存片段存在unused和allocated两种状态，在这里用一个数据成员type来表示，其中type==1代表该段内存已经被分配，type==0表示该段内存还未使用
- 内存分配时需要记录每段内存分配给了哪个进程，这里采用一个数据成员name来表示进程名，若该段内存未被分配，则不会为name分配空间
- begin和end分别表示该段内存的起始和终止位置
- next指向下一段内存

在本程序中，我们不专门实现链表的特定操作如insert等，而是将其与具体的功能结合起来，在各个功能函数中加以间接实现

程序的全局变量设计如下：

- char args[3][100]：用于RQ和RL命令中存储字符串形式的参数
- char cmd[100]：用于读入命令时暂存命令，为了避免每次进行动态内存分配和回收，选择设计为全局变量
- mem_node *head：内存链表的起始结点

3.2 main函数的设计与实现

main函数主要分为两个部分：

- 初始化部分，依据命令行参数，初始化一段mem_size大小的unused内存
- 循环命令执行部分，主体是一个while循环，每个循环读入一行命令并执行

3.2.1 初始化部分

主要分为两个部分：

- 参数正确性检查，我们检查参数数目是否为2，第二个表示内存大小的参数是否超过MAX（我们设为1GB）或是否溢出等
- 动态内存分配，创建head结点作为链表的头结点，此时head指向代表整块未使用的内存的结点

该部分代码如下：

```c
// initialization
init(argc, argv);
```

init()代码如下：

```c
void init(int argc, char *argv[])
{
    if (argc <= 1 || argc >= 3)
    {
        printf("Error: invalid arguments!\n\n");
        exit(1);
    }
    int mem_size = atoi(argv[1]);
    if (mem_size > MAX || mem_size < 0)
    {
        printf("Error: invalid memory size!\n\n");
        exit(1);
    }

    head = (mem_node *)malloc(sizeof(mem_node *)*10);
    if (!head)
    {
        printf("Error: memory initialization failed!\n\n");
        exit(1);
    }

    head->begin = 0;
    head->end = mem_size - 1;
    head->type = 0;
    head->next = NULL;
}
```

3.2.2 循环命令执行部分

一个条件恒为true的while循环中，首先利用函数read_cmd()读入新的一行命令并判断命令类型，根据不同的返回值，分别执行switch语句中的一个case，对指令进行执行

- 其中，read_cmd()的返回值存入一个int类型的变量mode中，mode值和命令类型有如下对应关系：
    - mode 0 represents "RQ"
    - mode 1 represents "RL"
    - mode 2 represents "C"
    - mode 3 represents "STAT"
    - mode 3 represents "X"
    - mode -1 represents invalid command

  read_cmd()的实现较为简单，主要就是读入输入的命令并利用一系列条件判断，判断命令类型；若是RQ或RL命令还需要将后续的参数暂时存入args中。

- read_cmd()后可以利用如下语句，对缓冲区中多余的字符读出删去

  ```c
  scanf("%*[^\n]%*c");
  ```

- case mode=0：

  RQ命令，调用**request()** 函数

request()的具体实现在 3.3 request()的实现 中详细说明

- case mode=1:

  RL命令，调用**release()** 函数

  release()的具体实现在 3.4 release()的实现 中详细说明

- case mode=2:

  C命令，调用**compact()** 函数

  compact()的具体实现在 3.5 compact()的实现 中详细说明

- case mode=3:

  STAT命令，调用**print_state()** 函数打印当前状态

  print_state()的具体实现在 3.6 print_state()的实现 中详细说明

- case mode=4:

  X命令，调用**clean()** 函数完成资源释放等工作后，打印退出成功的信息，并直接return 0表示程序正常退出即可

- default 情况:

  表示程序接受到了无效的命令，会打印Error信息后break，进入下一次循环重新等待新的命令

read_cmd()具体代码如下:

```
/*
mode 0 represents "RQ"
mode 1 represents "RL"
mode 2 represents "C"
mode 3 represents "STAT"
mode 4 represents "quit"
mode -1 represents invalid command
*/
int read_cmd(void)
{
    printf("allocator>  ");
    scanf("%s", cmd);

    if (!strcmp(cmd, "RQ"))
    {
        for (int i = 0; i < 3; i++)
            scanf(" %s", args[i]);
        return 0;
    }
    else if (!strcmp(cmd, "RL"))
    {
        scanf(" %s", args[0]);
        return 1;
    }
    else if (!strcmp(cmd, "C"))
    {
        return 2;
    }
    else if (!strcmp(cmd, "STAT"))
    {
        return 3;
    }
    else if (!strcmp(cmd, "X"))
```

```
    {
        return 4;
    }
    else
        return -1;
}
```

clean()具体代码如下：

```
void clean(void)
{
    while (head)
    {
        mem_node *tmp = head;
        head = head->next;
        free(tmp);
    }
}
```

main()具体代码如下：

```
int main(int argc, char *argv[])
{
    // initialization
    init(argc, argv);
    while (TRUE)
    {
        int mode = read_cmd();
        scanf("%*[^\n]%*c");

        switch (mode)
        {
        // RQ
        case 0:
        {
            request();
            break;
        }
        // RL
        case 1:
        {
            release();
            break;
        }
        // C
        case 2:
        {
            compact();
            break;
        }
        // STAT
        case 3:
        {
            print_state();
            break;
        }
```

```
        // x
        case 4:
        {
            printf("Exit successfully!\n\n");
            clean();
            return 0;
        }
        // error
        default:
        {
            printf("Error:Invalid command!\n\n");
            break;
        }
        }
    }
    return 0;
}
```

3.3 request的实现

主要分为以下三个部分：

- 判断参数是否合法
- 依据参数中的allocation strategy，寻找一块对应的未分配内存
- 在该块内存上进行为该进程进行分配

其中，第二和第三部分我们分别封装了一个工具函数search_available() 和allocate()，使得代码框架更清晰

- mem_node *search_available(int size, char strategy)

  接受目标内存大小和对应的内存分配策略，在当前可用内存中寻找一个可用结点，并返回指向该结点的指针；若无可用内存，则返回NULL

  函数主体是一个switch语句，根据不同的strategy作不同的操作，其中

  - “F”：采用first-fit的策略，只需从head结点往下遍历，每个结点处判断该内存是否已被分配，如仍处于unused状态且大小大于待分配大小，则返回该结点指针；否则打印错误信息，表示无可用内存，并返回NULL
  - “B”：采用best-fit的策略，采用两个额外的数据current_best_target和current_best_size来记录当前已遍历到的最佳选择的结点及其对应大小，从head遍历直到链表结尾，每次如有更小的可用内存则更新current_best_target和current_best_size，遍历结束后若current_best_target不为NULL则返回该指针；否则打印错误信息，表示无可用内存，并返回NULL
  - “W”：采用worst-fit的策略，采用两个额外的数据current_worst_target和current_worst_size来记录当前已遍历到的最差选择的结点及其对应大小，从head遍历直到链表结尾，每次如有更大的可用内存则更新current_worst_target和current_worst_size，遍历结束后若current_worst_target不为NULL则返回该指针；否则打印错误信息，表示无可用内存，并返回NULL
- void allocate(mem_node *target, char *proc_name, int size)

  在search_available返回的可用内存结点处，分配一块size大小的新内存，并将其分配给名为proc_name的进程，其中分配的过程就是新malloc一个结点的过程，并注意更新两个结点的begin和end

search_available() 具体代码如下：

```c
mem_node *search_available(int size, char strategy)
{
    mem_node *ptr = head;
    switch (strategy)
    {
    case 'F':
    {
        while (ptr)
        {
            if (ptr->type == 1)
            {
                ptr = ptr->next;
                continue; // already allocated
            }
            int current_size = ptr->end - ptr->begin + 1;
            if (current_size >= size)
                return ptr;
            else
                ptr = ptr->next;
        }
        if (ptr == NULL)
        {
            printf("Error: There's insufficient memory to be allocated, request rejected!\n");
            return NULL;
        }
        break;
    }
    case 'B':
    {
        mem_node *current_best_target = NULL;
        int current_best_size = INT_MAX;
        while (ptr)
        {
            if (ptr->type == 1)
            {
                ptr = ptr->next;
                continue; // already allocated
            }
            int current_size = ptr->end - ptr->begin + 1;
            if (current_size >= size)
            {
                if (current_best_size > current_size)
                {
                    current_best_target = ptr;
                    current_best_size = current_size;
                }
            }
            ptr = ptr->next;
        }
        if (current_best_target == NULL)
        {
            printf("Error: There's insufficient memory to be allocated, request rejected!\n");
            return NULL;
        }
        return current_best_target;
```

```
        break;
    }
    case 'W':
    {
        mem_node *current_worst_target = NULL;
        int current_worst_size = 0;
        while (ptr)
        {
            if (ptr->type == 1)
            {
                ptr = ptr->next;
                continue; // already allocated
            }
            int current_size = ptr->end - ptr->begin + 1;
            if (current_size >= size)
            {
                if (current_worst_size < current_size)
                {
                    current_worst_target = ptr;
                    current_worst_size = current_size;
                }
            }
            ptr = ptr->next;
        }
        if (current_worst_target == NULL)
        {
            printf("Error: There's insufficient memory to be allocated, request rejected!\n");
            return NULL;
        }
        return current_worst_target;
        break;
    }
    default:
    {
        printf("Error: invalid allocation strategy argument!\n");
        break;
    }
    }
    return NULL;
}
```

allocate() 具体代码如下：

```
void allocate(mem_node *target, char *proc_name, int size)
{
    mem_node *new_node = (mem_node *)malloc(sizeof(mem_node *));
    target->name = (char *)malloc(sizeof(char) * (strlen(proc_name) + 1));

    // new_node
    new_node->next = target->next;
    target->next = new_node;
    new_node->begin = target->begin + size;
    new_node->end = target->end;
    new_node->type = 0;

    // target
```

```
    target->end = target->begin + size - 1;
    target->type = 1;
    strcpy(target->name, proc_name);
}
```

有了这两个工具函数，则request()的实现就变得非常简单了，只需要调用两个函数并判断一下返回值即可

request() 具体代码如下：

```
void request(void)
{
    int alloc_size = atoi(args[1]);
    if (alloc_size > MAX || alloc_size < 0)
    {
        printf("Error: Invalid requested memory size!\n");
        return;
    }

    mem_node *target = search_available(alloc_size, args[2][0]);
    if (target == NULL)
        return;

    allocate(target, args[0], alloc_size);
    printf("Request for %s has been satisfied!\n\n", args[0]);
}
```

3.4 release的实现

相比request，release的实现要简单一些，只需要从head开始遍历，遇到名字与参数相同的已分配内存块则将其type改为0（表示未分配），如此即可表示该内存被回收了

- 需要注意release可能产生两个相邻的hole，此时需要对此两个块进行合并，我们封装一个新的名为merge() 的工具函数，检查整个内存中是否有相邻hole，有的话将其合并（如此设计的话，该函数也可用与compact功能中）

  merge() 的实现，采用双指针的方式，从head开始不断向下遍历，遇到两个相邻的type=0的结点，则将其合并为一个，并将另一个的空间进行释放

merge() 具体代码如下：

```
void merge(void)
{
    if (!head->next)
        return;
    mem_node *p1 = head, *p2 = head->next;

    while (p2)
    {
        if (p1->type == 0 && p2->type == 0)
        {
            p1->next = p2->next;
            p1->end = p2->end;
            free(p2);
            p2 = p1->next;
```

```
        }
        else
        {
            p1 = p1->next;
            p2 = p2->next;
        }
    }
}
```

release() 具体代码如下:

```
void release(void)
{
    char *name = args[0];
    mem_node *tmp = head;
    int flag = 0;   //flag=0 means there's no such a block of memory allocated for this process
    while (tmp)
    {
        if (tmp->type == 0)
        {
            tmp = tmp->next;
            continue;
        }
        else if (!strcmp(tmp->name, name))
        {
            tmp->type = 0;
            flag = 1;
        }
        tmp = tmp->next;
    }
    if (flag)   //release successfully
    {
        merge();
        printf("Memory allocated for %s has been released!\n\n", args[0]);
    }
    else
    {
        printf("Error: Release failed, no memory has been allocated for process named %s!\n\n", args[0]);
    }
}
```

3.5 compact的实现

compact() 的实现我采用了置换的思路，即从head开始遍历，每次遇到一个unused的内存块，就在它的后面遍历直到找到一个已分配的内存块，并将两个内存块位置互换，重复以上过程直到遇到unused内存块后，其后无法找到一个已分配的内存块，即所有的unused内存块都已集中于内存的后半部分，所有的已分配内存块都在内存的前半部分，即实现了compact的功能，之后只需要调用3.4中的merge() 函数，将所有相邻的unused内存块合并为一整块即可

具体细节:

- 每次置换完两个结点，就需要从被置换点开始，更新其后的内存结点的begin和end值，表示该内存已经被移动

- 为了实现置换，我们需要记录待置换的unused和allocated结点的前一个结点，分别采用unused_pre和allocated_pre来表示，并在遍历过程中不断更新
- 为了处理head指向的首个结点即为unused，此时不存在unused_pre的情况，出于方便统一方式进行处理的目的，我们设计了一个虚拟的头结点，其next指向head结点，并在完成工作后free掉该虚拟结点
- 在置换全部完成后，需要调用merge() 合并所有的相邻unused结点
- 为了防止置换过程中，将head移到链表的中间位置，丢失了前面的结点，我们需要判断当前结点是否为head，是的话需要将head重新指向第一个结点

compact() 具体代码如下:

```
void compact(void)
{
    mem_node *unused = head, *unused_pre;
    mem_node *allocated, *allocated_pre;
    unused_pre = (mem_node *)malloc(sizeof(mem_node *));    //virtual head node
    unused_pre->next = head;
    unused_pre->end = -1;
    mem_node *to_be_free = unused_pre;

    while (TRUE)
    {
        while (unused && unused->type == 1)
        {
            unused_pre = unused;
            unused = unused->next;
        }
        if (unused == NULL) // all memory been allocated, no hole exists
            break;
        ;

        allocated = unused->next;
        allocated_pre = unused;
        while (allocated && allocated->type != 1)
        {
            allocated_pre = allocated;
            allocated = allocated->next;
        }
        if (allocated == NULL)
            break;

        // swap the unused node and the allocated node
        unused_pre->next = allocated;
        allocated_pre->next = unused;
        mem_node *tmp = allocated->next;
        allocated->next = unused->next;
        unused->next = tmp;

        if (unused == head)     //in case that head is swapped, update the head pointer
            head = unused_pre->next;

        tmp = unused_pre->next;
        int current_begin = unused_pre->end + 1;
        while (tmp)     //update the begin and end of the affected memory nodes
```

```
        {
            int size = tmp->end - tmp->begin + 1;
            tmp->begin = current_begin;
            tmp->end = tmp->begin + size - 1;
            current_begin = tmp->end + 1;
            tmp = tmp->next;
        }

        unused = unused_pre->next;
    }

    merge();
    free(to_be_free);
    printf("Compact unused holes successfully!\n\n");
}
```

3.6 print_state的实现

实现较为简单，只需要从head开始遍历链表，根据type输出各段内存的信息即可

```
void print_state(void)
{
    printf("The current state:\n");
    mem_node *tmp = head;
    while (tmp)
    {
        printf("Addresses [ %d : %d ]\t", tmp->begin, tmp->end);
        if (tmp->type)
        {
            printf("Process %s\n", tmp->name);
        }
        else
        {
            printf("Unused\n");
        }
        tmp = tmp->next;
    }
    printf("\n");
}
```

4 Test result

首先编写如下的Makefile对C文件进行编译

```
CC=gcc
CFLAGS=-Wall

all: allocator.o
	$(CC) $(CFLAGS) -o allocator allocator.o

allocator.o: allocator.c
	$(CC) $(CFLAGS) -c allocator.c

clean:
	rm -rf *.o
	rm -rf allocator
```

设计如下的测试命令：

```
//异常处理
./allocator 100000000000
./allocator
./allocator 1048576 xxxx
```

```
zh@ubuntu:~/os-project/pro7$ ./allocator 100000000000
Error: invalid memory size!

zh@ubuntu:~/os-project/pro7$ ./allocator
Error: invalid arguments!

zh@ubuntu:~/os-project/pro7$ ./allocator 1048576 xxxx
Error: invalid arguments!
```

```
//基础测试
RQ P0 40000 W
RQ P1 20000 B
RQ P2 30000 F
RQ P3 10000 W
STAT
RL P1
RL P2
STAT
```

```
zh@ubuntu:~/os-project/pro7$ ./allocator 1048576
allocator>  RQ P0 40000 W
Request for P0 has been satisfied!

allocator>  RQ P1 20000 B
Request for P1 has been satisfied!

allocator>  RQ P2 30000 F
Request for P2 has been satisfied!

allocator>  RQ P3 10000 W
Request for P3 has been satisfied!

allocator>  STAT
The current state:
Addresses [ 0 : 39999 ] Process P0
Addresses [ 40000 : 59999 ]     Process P1
Addresses [ 60000 : 89999 ]     Process P2
Addresses [ 90000 : 99999 ]     Process P3
Addresses [ 100000 : 1048575 ]  Unused

allocator>  RL P1
Memory allocated for P1 has been released!

allocator>  RL P2
Memory allocated for P2 has been released!

allocator>  STAT
The current state:
Addresses [ 0 : 39999 ] Process P0
Addresses [ 40000 : 89999 ]     Unused
Addresses [ 90000 : 99999 ]     Process P3
Addresses [ 100000 : 1048575 ]  Unused
```

可以看到，RL后相邻的holes可以成功合并为一个，基础的RQ，RL和STAT命令都测试成功

```
// allocation strategy测试
STAT
RQ P4 1000 F
RQ P5 1000 W
RQ P6 1000 B
STAT
```

```
allocator>  STAT
The current state:
Addresses [ 0 : 39999 ] Process P0
Addresses [ 40000 : 89999 ]     Unused
Addresses [ 90000 : 99999 ]     Process P3
Addresses [ 100000 : 1048575 ]  Unused

allocator>  RQ P4 1000 F
Request for P4 has been satisfied!

allocator>  RQ P5 1000 W
Request for P5 has been satisfied!

allocator>  RQ P6 1000 B
Request for P6 has been satisfied!

allocator>  STAT
The current state:
Addresses [ 0 : 39999 ] Process P0
Addresses [ 40000 : 40999 ]     Process P4
Addresses [ 41000 : 41999 ]     Process P6
Addresses [ 42000 : 89999 ]     Unused
Addresses [ 90000 : 99999 ]     Process P3
Addresses [ 100000 : 100999 ]   Process P5
Addresses [ 101000 : 1048575 ]  Unused
```

可以看到，RQ指令的allocation strategy测试成功

```
//compact测试
RL P0
RL P4
STAT
C
STAT
```

```
allocator>  RL P0
Memory allocated for P0 has been released!

allocator>  RL P4
Memory allocated for P4 has been released!

allocator>  STAT
The current state:
Addresses [ 0 : 40999 ] Unused
Addresses [ 41000 : 41999 ]     Process P6
Addresses [ 42000 : 89999 ]     Unused
Addresses [ 90000 : 99999 ]     Process P3
Addresses [ 100000 : 100999 ]   Process P5
Addresses [ 101000 : 1048575 ]  Unused

allocator>  C
Compact unused holes successfully!

allocator>  STAT
The current state:
Addresses [ 0 : 999 ]   Process P6
Addresses [ 1000 : 10999 ]      Process P3
Addresses [ 11000 : 11999 ]     Process P5
Addresses [ 12000 : 1048575 ]   Unused
```

可以看到，compact功能测试成功

```
//异常处理 以及 quit测试
RQ P0 100000000000 W    //超过MAX
RQ P0 1000000 W
RQ P1 1000000 B         //剩余内存不足以分配
RQ P1 10000 A           //错误allocation strategy
RL P1                   //释放未分配的内存
q
```

```
allocator>  RQ P0 100000000000 W
Error: Invalid requested memory size!

allocator>  RQ P0 1000000 W
Request for P0 has been satisfied!

allocator>  RQ P1 1000000 B
Error: There's insufficient memory to be allocated, request rejected!

allocator>  RQ P1 10000 A
Error: invalid allocation strategy argument!

allocator>  RL P1
Error: Release failed, no memory has been allocated for process named P1!

allocator>  X
Exit successfully!
```

综上，本程序顺利通过了所有功能的测试

5 Summary

本次通过编写模拟连续内存分配的程序，对课内内容进行实践，使得我对该内存分配算法有了更为深入的理解。同时，本次project函数数目较多，需要好好组织代码结构并且合理添加注释，才可以使得代码可读性更强一些。此外，本次project思路不算太难，主要是更熟悉了一些链表的操作技巧，锻炼了编程能力。总之，本次project难度不算很大，而且完成过程也富有乐趣（比如多种可行思路权衡一个最高效的算法等），收获良多。