上海交通大學

《操作系统》课程

学生实验报告

实验名称:	Project7:Contiguous Memory Allocation
姓　　名:	洪瑄锐
学　　号:	517030910227
班　　级:	F1703302
手　　机:	15248246044
任课老师:	吴晨涛

2019 年 6 　月　 8 　 日

目录

一、实验要求

该项目涉及管理大小为 MAX 的连续内存区域，其中地址范围为 0…MAX-1,程序必须响应 4 个不同的请求：

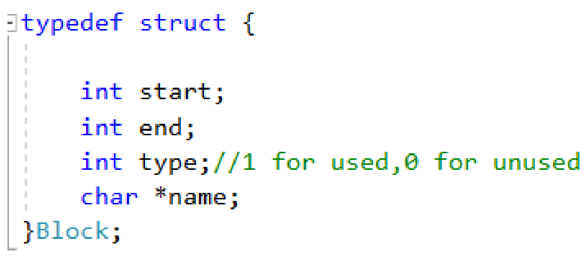
1. 请求连续的内存块
2. 释放连续的内存
3. 将未使用的内存块压缩到一个块中
4. 报告空闲和已分配内存的区域

二、程序设计思想及代码解释

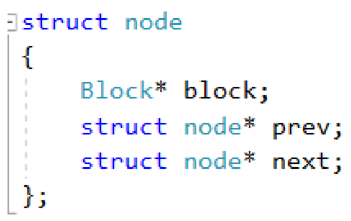
1. 基本数据结构

• 内存块 Block，包括 start，end，name，type（1 表示该块被使用，0 表示该块未使用）

• 链表结点 node，用于构建内存块列表



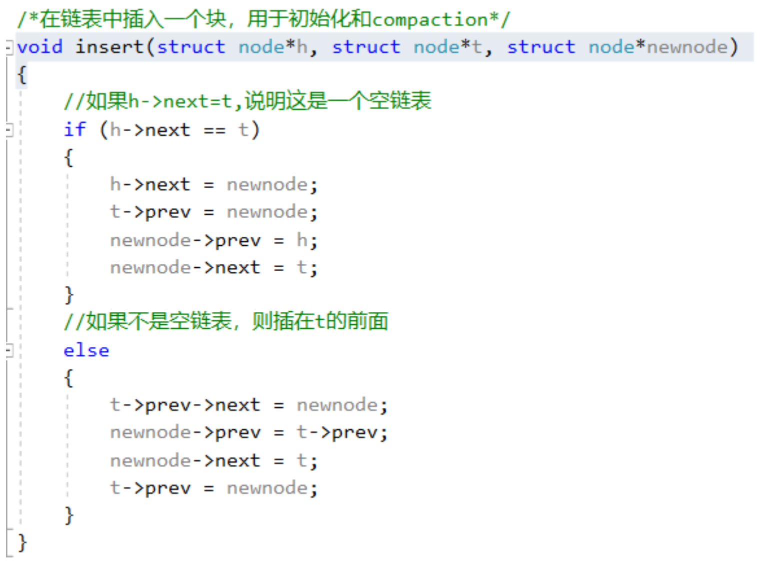
```
typedef struct {

    int start;
    int end;
    int type;//1 for used,0 for unused
    char *name;
}Block;
```



```
struct node
{
    Block* block;
    struct node* prev;
    struct node* next;
};
```

2. void insert(struct node*h, struct node*t, struct node*newnode)

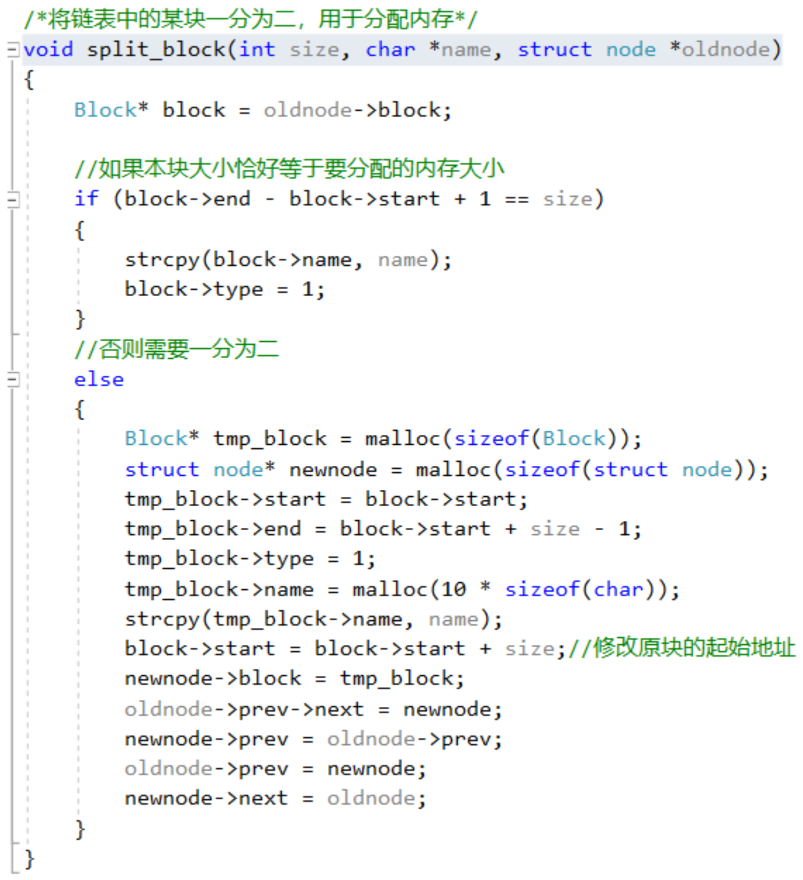
向内存块链表中插入一个结点（内存块），注意插入在链表的后面。



```
/*在链表中插入一个块，用于初始化和compaction*/
void insert(struct node*h, struct node*t, struct node*newnode)
{
    //如果h->next=t,说明这是一个空链表
    if (h->next == t)
    {
        h->next = newnode;
        t->prev = newnode;
        newnode->prev = h;
        newnode->next = t;
    }
    //如果不是空链表，则插在t的前面
    else
    {
        t->prev->next = newnode;
        newnode->prev = t->prev;
        newnode->next = t;
        t->prev = newnode;
    }
}
```

3. void split_block(int size, char *name, struct node *oldnode)

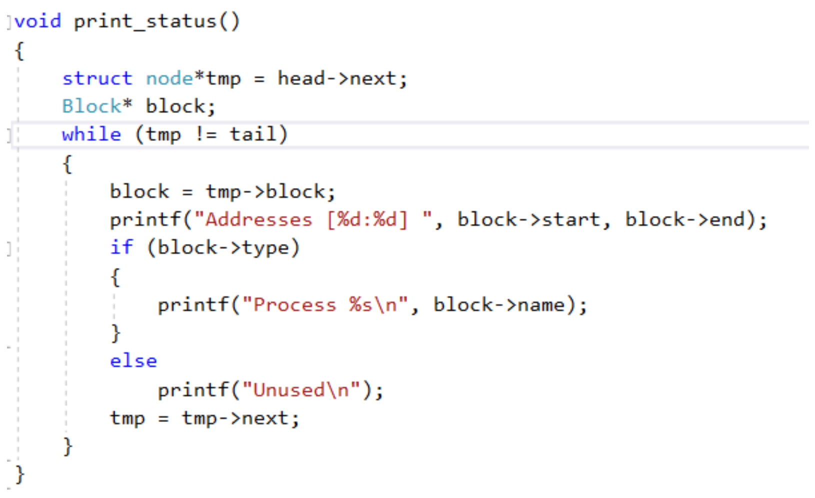
当有内存块需要被分配时，如果原内存块和所需内存大小一致，则修改 name 和 type 即可，否则原内存块 oldnode 被一分为二，前一部分继承原内存块的 start，并赋值给 end 和 name，type 修改为 1 即被使用，后一部分继承原内存块的 end，type，其 start 根据前一部分的 end 修改。



```c
/*将链表中的某块一分为二，用于分配内存*/
void split_block(int size, char *name, struct node *oldnode)
{
    Block* block = oldnode->block;

    //如果本块大小恰好等于要分配的内存大小
    if (block->end - block->start + 1 == size)
    {
        strcpy(block->name, name);
        block->type = 1;
    }
    //否则需要一分为二
    else
    {
        Block* tmp_block = malloc(sizeof(Block));
        struct node* newnode = malloc(sizeof(struct node));
        tmp_block->start = block->start;
        tmp_block->end = block->start + size - 1;
        tmp_block->type = 1;
        tmp_block->name = malloc(10 * sizeof(char));
        strcpy(tmp_block->name, name);
        block->start = block->start + size;//修改原块的起始地址
        newnode->block = tmp_block;
        oldnode->prev->next = newnode;
        newnode->prev = oldnode->prev;
        oldnode->prev = newnode;
        newnode->next = oldnode;
    }
}
```

4. 打印内存状态

遍历整个内存链表，打印信息。

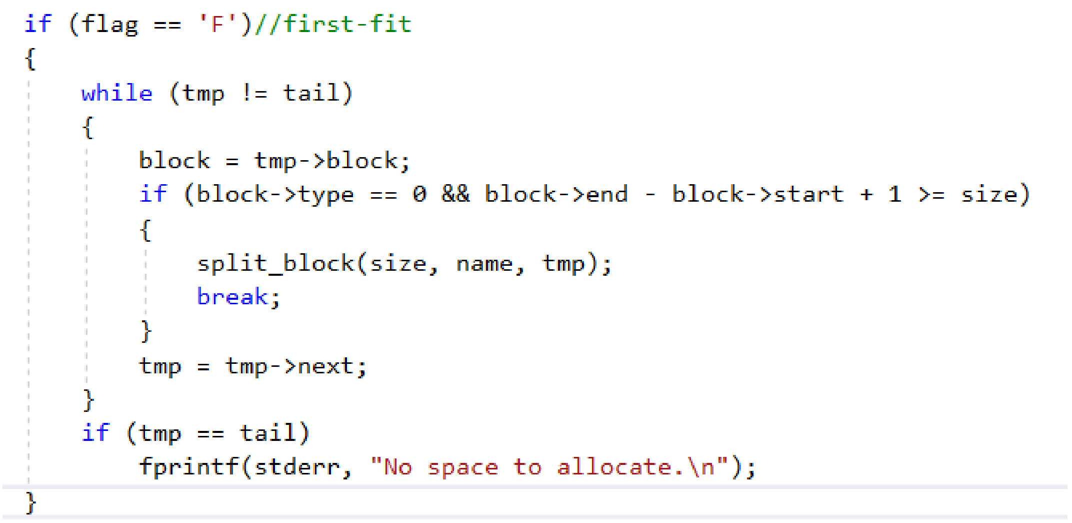


```c
void print_status()
{
    struct node*tmp = head->next;
    Block* block;
    while (tmp != tail)
    {
        block = tmp->block;
        printf("Addresses [%d:%d] ", block->start, block->end);
        if (block->type)
        {
            printf("Process %s\n", block->name);
        }
        else
            printf("Unused\n");
        tmp = tmp->next;
    }
}
```

5. void command_rq(char *command)

根据 command 识别 name， size 以及分配内存方式。

• 如果分配方式为 First fit，那么遍历整个内存链表至找到第一个 size 足够且未被使用的内存块，如果遍历到链表尾部 tail 都没有找到，说明没有可分配的内存。如果找到则调用 split_block 函数。

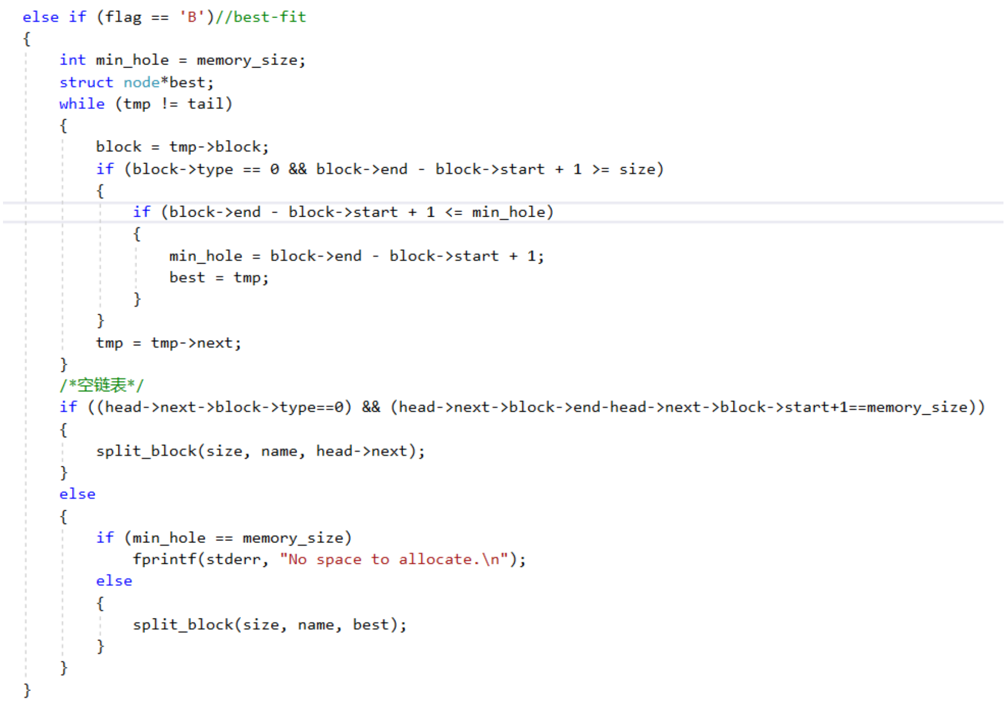


```
if (flag == 'F')//first-fit
{
    while (tmp != tail)
    {
        block = tmp->block;
        if (block->type == 0 && block->end - block->start + 1 >= size)
        {
            split_block(size, name, tmp);
            break;
        }
        tmp = tmp->next;
    }
    if (tmp == tail)
        fprintf(stderr, "No space to allocate.\n");
}
```

• 如果分配方式为 Best fit，那么定义一个变量 min_size 用于找到足够规模且最小的内存块,初始化为整个内存的大小。遍历整个链表，

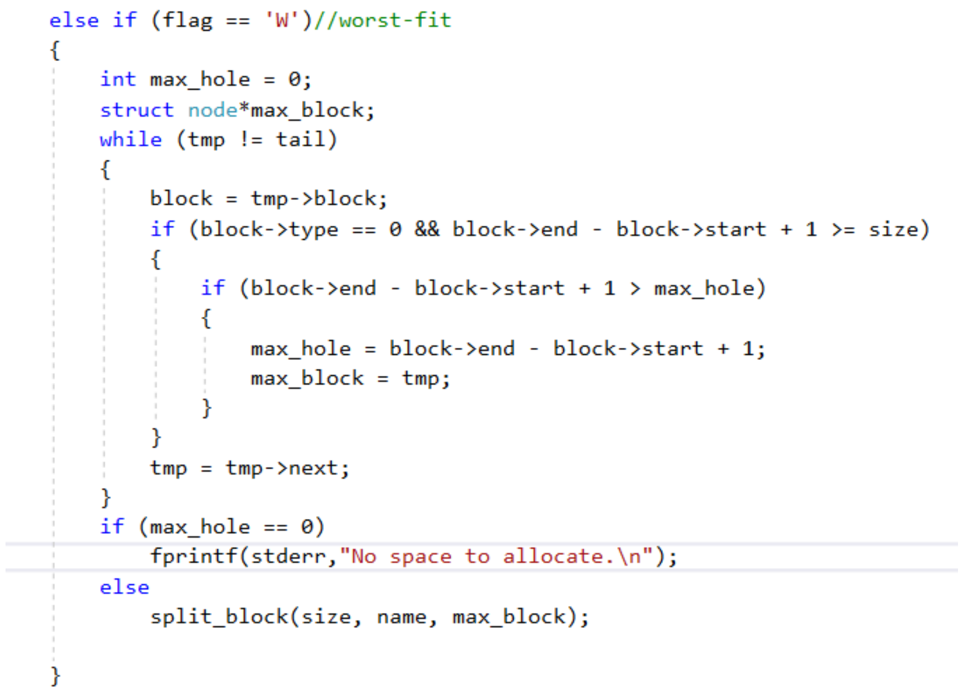
如果遍历结束后 min_size 仍然等于 memory_size,有两种情况(1)该内存还未被分配，整个内存都处于未使用状态，则调用 split_block 函数即可(2)该内存找不到 size 足够的内存，则输出错误信息。

如果遍历结束后 min_size 不等于 memory_size,则证明找到了 best， 调用 split_block 即可。



```
else if (flag == 'B')//best-fit
{
    int min_hole = memory_size;
    struct node*best;
    while (tmp != tail)
    {
        block = tmp->block;
        if (block->type == 0 && block->end - block->start + 1 >= size)
        {
            if (block->end - block->start + 1 <= min_hole)
            {
                min_hole = block->end - block->start + 1;
                best = tmp;
            }
        }
        tmp = tmp->next;
    }
    /*空链表*/
    if ((head->next->block->type==0) && (head->next->block->end-head->next->block->start+1==memory_size))
    {
        split_block(size, name, head->next);
    }
    else
    {
        if (min_hole == memory_size)
            fprintf(stderr, "No space to allocate.\n");
        else
        {
            split_block(size, name, best);
        }
    }
}
```

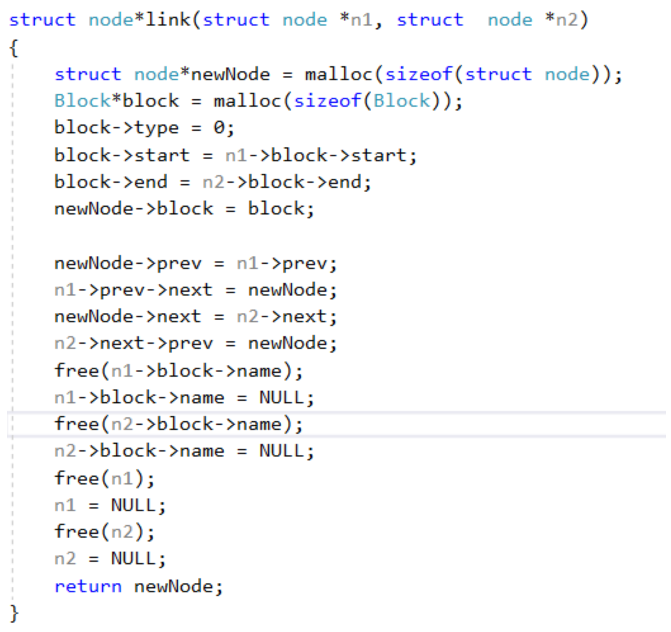
• 如果分配方式为 Worst fit，那么定义一个变量 max_size,初始化为 0，用于找到足够规模且最大的内存块。遍历整个链表，如果遍历结束后 max_size 仍为 0，说明无法分配，输出错误信息，否则调用 split_block 函数。



```
else if (flag == 'W')//worst-fit
{
    int max_hole = 0;
    struct node*max_block;
    while (tmp != tail)
    {
        block = tmp->block;
        if (block->type == 0 && block->end - block->start + 1 >= size)
        {
            if (block->end - block->start + 1 > max_hole)
            {
                max_hole = block->end - block->start + 1;
                max_block = tmp;
            }
        }
        tmp = tmp->next;
    }
    if (max_hole == 0)
        fprintf(stderr,"No space to allocate.\n");
    else
        split_block(size, name, max_block);

}
```

6. struct node*link(struct node *n1, struct node *n2)

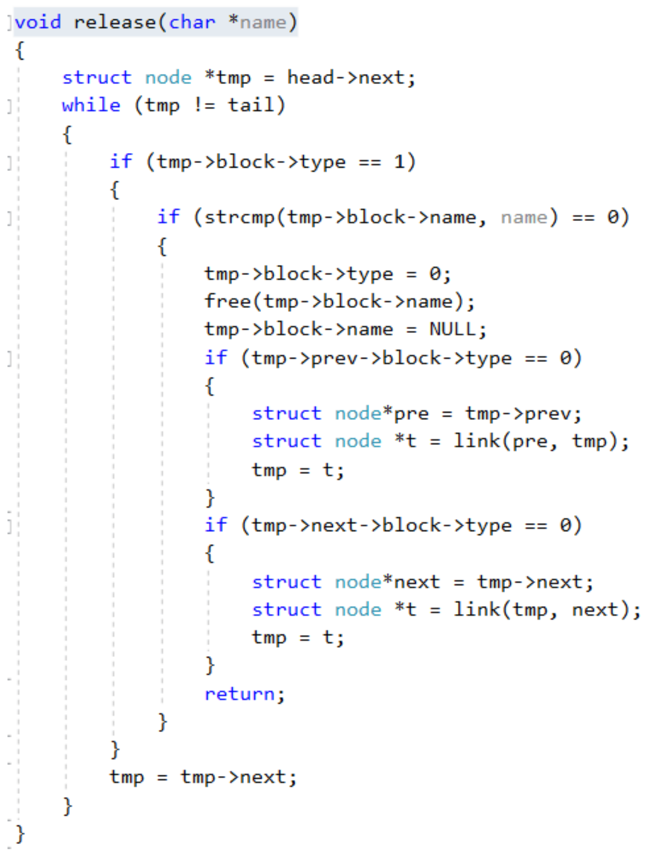
将内存链表中的两个 block 块连接为一个块。



```
struct node*link(struct node *n1, struct  node *n2)
{
    struct node*newNode = malloc(sizeof(struct node));
    Block*block = malloc(sizeof(Block));
    block->type = 0;
    block->start = n1->block->start;
    block->end = n2->block->end;
    newNode->block = block;

    newNode->prev = n1->prev;
    n1->prev->next = newNode;
    newNode->next = n2->next;
    n2->next->prev = newNode;
    free(n1->block->name);
    n1->block->name = NULL;
    free(n2->block->name);
    n2->block->name = NULL;
    free(n1);
    n1 = NULL;
    free(n2);
    n2 = NULL;
    return newNode;
}
```

7. void release(char *name)

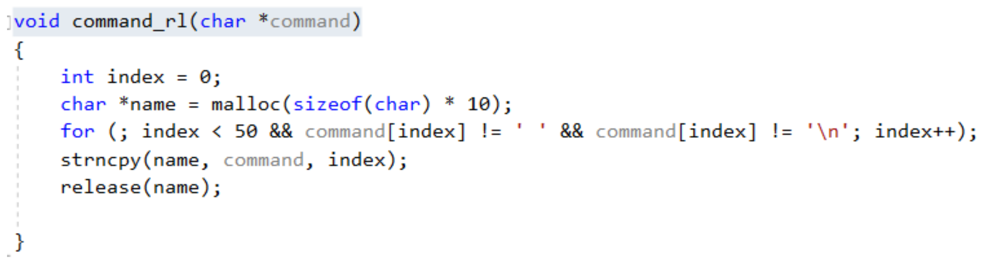
遍历链表找到名字为 name 的内存块，将内存块 type 置 0，释放 name 内存，并检查其前后相邻内存块是否为空，如果有相邻空内存块，则调用 link 函数连接。



```
void release(char *name)
{
    struct node *tmp = head->next;
    while (tmp != tail)
    {
        if (tmp->block->type == 1)
        {
            if (strcmp(tmp->block->name, name) == 0)
            {
                tmp->block->type = 0;
                free(tmp->block->name);
                tmp->block->name = NULL;
                if (tmp->prev->block->type == 0)
                {
                    struct node*pre = tmp->prev;
                    struct node *t = link(pre, tmp);
                    tmp = t;
                }
                if (tmp->next->block->type == 0)
                {
                    struct node*next = tmp->next;
                    struct node *t = link(tmp, next);
                    tmp = t;
                }
                return;
            }
        }
        tmp = tmp->next;
    }
}
```

8. void command_rl(char *command)

   在命令中识别出 name，调用 release 函数即可。



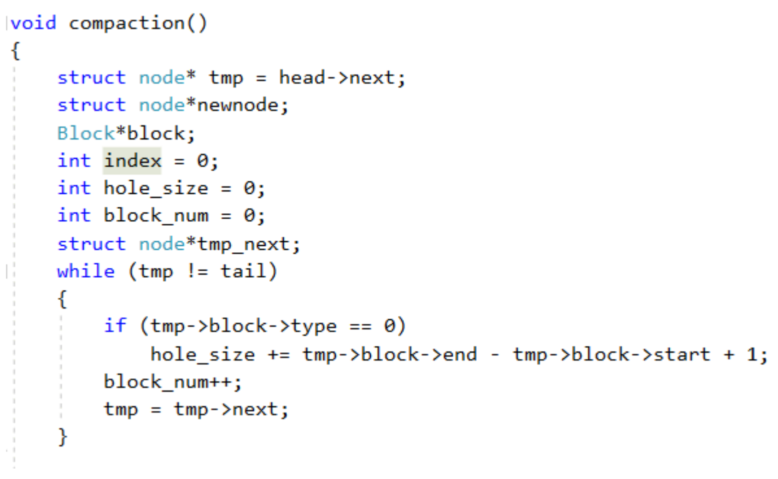
```
void command_rl(char *command)
{
    int index = 0;
    char *name = malloc(sizeof(char) * 10);
    for (; index < 50 && command[index] != ' ' && command[index] != '\n'; index++);
    strncpy(name, command, index);
    release(name);

}
```

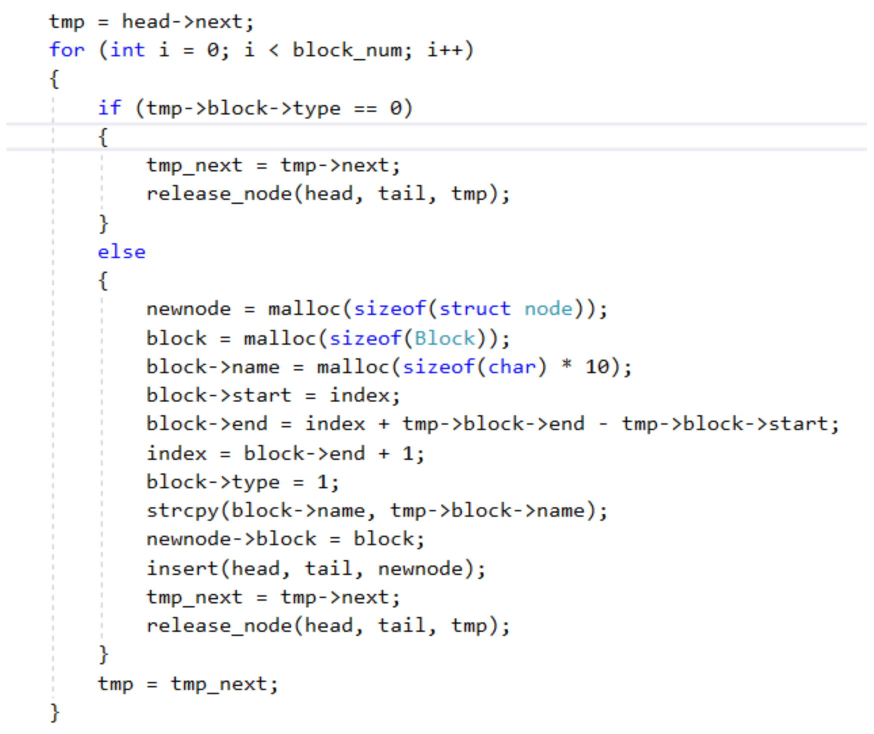
9. void compaction()

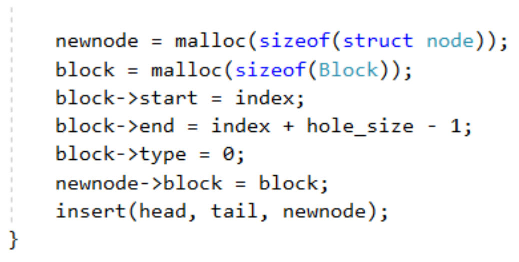
   (1)遍历整个链表，得到内存块数目 block_num 和空内存块的 size 加和

   (2)遍历整个链表，采用 for 循环 block_num 次，如果某块内存未使用，则删除，如果某块内存被使用，复制后调用 insert 函数插入链表尾部，删除原内存块。

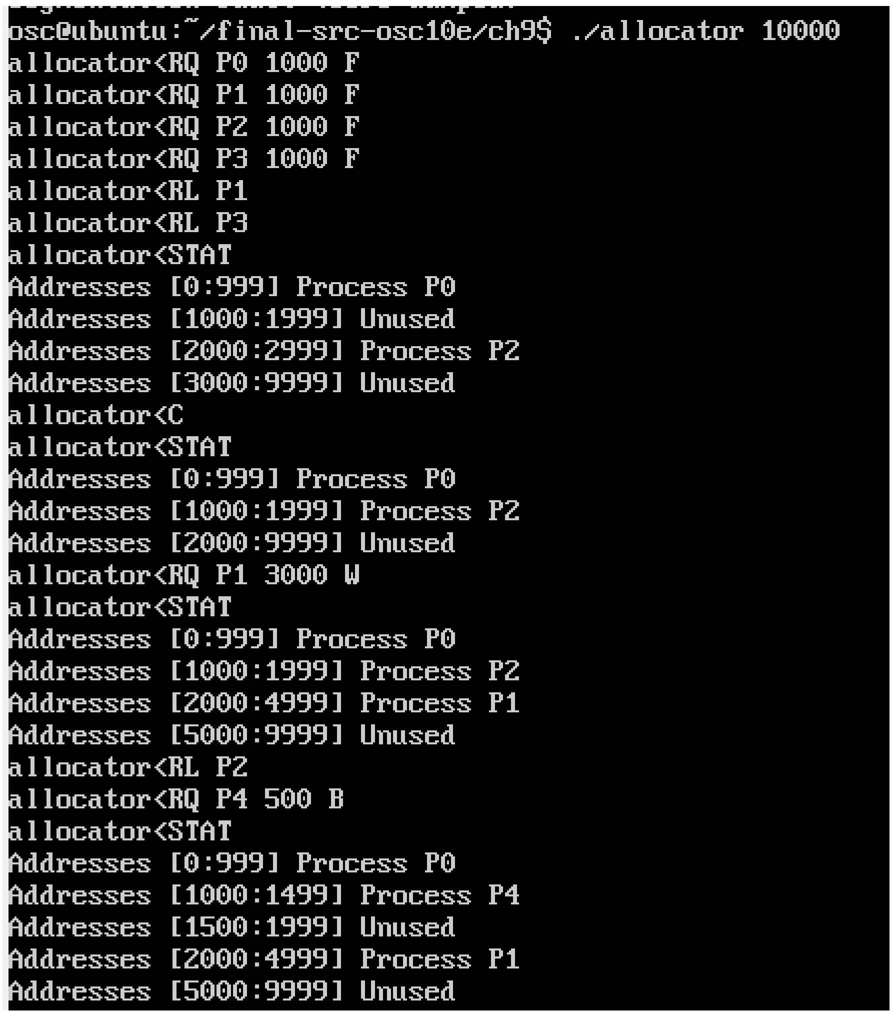
   (3)当 for 循环结束后，链表只剩下复制的被使用的内存块，新建一个内存块，size 为空内存块的 size 加和，插入到链表尾部。


```c
void compaction()
{
    struct node* tmp = head->next;
    struct node*newnode;
    Block*block;
    int index = 0;
    int hole_size = 0;
    int block_num = 0;
    struct node*tmp_next;
    while (tmp != tail)
    {
        if (tmp->block->type == 0)
            hole_size += tmp->block->end - tmp->block->start + 1;
        block_num++;
        tmp = tmp->next;
    }
```


```c
    tmp = head->next;
    for (int i = 0; i < block_num; i++)
    {
        if (tmp->block->type == 0)
        {
            tmp_next = tmp->next;
            release_node(head, tail, tmp);
        }
        else
        {
            newnode = malloc(sizeof(struct node));
            block = malloc(sizeof(Block));
            block->name = malloc(sizeof(char) * 10);
            block->start = index;
            block->end = index + tmp->block->end - tmp->block->start;
            index = block->end + 1;
            block->type = 1;
            strcpy(block->name, tmp->block->name);
            newnode->block = block;
            insert(head, tail, newnode);
            tmp_next = tmp->next;
            release_node(head, tail, tmp);
        }
        tmp = tmp_next;
    }
```


```c
    newnode = malloc(sizeof(struct node));
    block = malloc(sizeof(Block));
    block->start = index;
    block->end = index + hole_size - 1;
    block->type = 0;
    newnode->block = block;
    insert(head, tail, newnode);
}
```

三、运行结果截图



```
osc@ubuntu:~/final-src-osc10e/ch9$ ./allocator 10000
allocator<RQ P0 1000 F
allocator<RQ P1 1000 F
allocator<RQ P2 1000 F
allocator<RQ P3 1000 F
allocator<RL P1
allocator<RL P3
allocator<STAT
Addresses [0:999] Process P0
Addresses [1000:1999] Unused
Addresses [2000:2999] Process P2
Addresses [3000:9999] Unused
allocator<C
allocator<STAT
Addresses [0:999] Process P0
Addresses [1000:1999] Process P2
Addresses [2000:9999] Unused
allocator<RQ P1 3000 W
allocator<STAT
Addresses [0:999] Process P0
Addresses [1000:1999] Process P2
Addresses [2000:4999] Process P1
Addresses [5000:9999] Unused
allocator<RL P2
allocator<RQ P4 500 B
allocator<STAT
Addresses [0:999] Process P0
Addresses [1000:1499] Process P4
Addresses [1500:1999] Unused
Addresses [2000:4999] Process P1
Addresses [5000:9999] Unused
```