操作系统 project-7 实验报告

∗ 姓名: 管仁阳　学号:519021911058　邮箱: guanrenyang@sjtu.edu.cn

1　实验名称

Contiguous Memory Allocation

2　实验任务

编写 C 程序来实现连续内存分配，需要支持以下方式：

1. First-Fit
2. Worst-Fit
3. Best-Fit

且支持以下功能

1. 用户请求分配连续的内存
2. 用户释放连续的内存块
3. 将为使用的内存块压缩成一整个
4. 显示使用的和未使用的的内存块

3　预备知识

3.1　连续内存分配：

主内存必须同时兼顾操作系统和各种用户进程。因此我们需要使用连续内存分配。内存通常分为两个分区：一个用于操作系统和一个用于用户进程的系统。每个进程都包含在单个内存单元中且与包含下一个进程的部分相邻。

3.2　First-Fit

分配足够大的第一个 hole。搜索可以在第一个 hole 或者它以前开始。一旦发现空的 hole 就可以立即停止搜索因为它已经足够大了。

3.3　Worst-Fit

分配最大的一个 hole。搜索必须遍历所有 hole 以找到最大的 hole。若未找到则拒绝请求。

3.4　Best-Fit

分配大小最合适的一个 hole。搜索必须遍历所有 hole 以找到最合适的 hole。若未找到则拒绝请求。

4 实验内容

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

#define MAX_INPUT 20

#define RQ 1
#define RL 2
#define C 3
#define STAT 4

#define Fail -1

short *memory;//0 empty, 1 full

int *memory_size;
struct processor
{
    char * name;
    int starting_address;
    int ending_address;
    struct processor * next;
};

struct processor *head;
// list function
void insert(char* name, int starting_address , int ending_address);
struct processor * search_by_address(int i);
struct processor * search_by_name(char * name);
struct processor * delete_by_name(char * name);

int Decode_Input(char * input);

int request(char * input);
int release(char * input);
void compact();
void status();

int first_fit(int size);
int worst_fit(int size);
int best_fit(int size);

int main(int argc,char * argv[])
{
    memory_size=(int*)malloc(sizeof(int));
    (*memory_size)=atoi(argv[1]);
    memory=(short*) malloc(sizeof(short)*(*memory_size));

    fprintf(stdout,"allocator>");

    char * input=malloc(MAX_INPUT*sizeof(char));
    while (fgets(input,MAX_INPUT-1,stdin))
    {

        int command=Decode_Input(input);
        switch (command)
        {
        case RQ:
            request(input);
            break;
        case RL:
            release(input);
            break;
        case C:
            compact();
            break;
        case STAT:
            status();
        default:
            break;
        }
        fprintf(stdout,"allocator>");
    }

}
int Decode_Input(char * input)
{
    if(input[0]=='C')
        return C;
    else if(input[0]=='S')
        return STAT;
    else if(input[1]=='Q')
        return RQ;
    else if(input[1]=='L')
        return RL;
}
int request(char *input)
{
    char * command=strsep(&input," ");
    char * name=strsep(&input," ");
    int size=atoi(strsep(&input," "));
    char * strategy=strsep(&input," ");
    //ok

    int start;
    if(strategy[0]=='F')
        start=first_fit(size);
    else if(strategy[0]=='W')
```

```
        start=worst_fit(size);
    else if(strategy[0]=='B')
        start=best_fit(size);

    for(int i=0;i<size;++i)
    {
        memory[start+i]=1;
    }
    insert(name,start ,start+size-1);

}
int release(char * input)
{
    char * command=strsep(&input," ");
    char * name=strsep(&input,"\n");

    struct processor * temp=delete_by_name(name);
    for(int i=temp->starting_address;i<=temp->ending_address;++i)
        memory[i]=0;
    free(temp);
}

void compact()
{
    struct processor *new_head=NULL;
    for(int i=0;i<(*memory_size);++i)
    {
        while (head!=NULL)
        {
            int size=head->ending_address-head->starting_address+1;

            for(int j=0;j<size;j++)
                memory[i+j]=1;

            head->starting_address=i;
            head->ending_address=i+size-1;
            struct processor * temp=head;
            head=head->next;

            // all temp to the new head
            if(new_head==NULL)
            {
                new_head=temp;
                new_head->next=NULL;
            }
            else
            {
                temp->next=new_head;
                new_head=temp;
            }
            i=i+size;
        }
        memory[i]=0;
    }
    head=new_head;
}
void status()
{

    int start=0;
    int end=start;
    int prev=0;
    for(int i=0;i<(*memory_size);++i)
    {
        if(prev==0&&memory[i]==1&&i!=0)
        {
            fprintf(stdout,"Addresses [%d:%d] Unuesd\n",start,end);
        }
        else if(prev==1&&memory[i]==0)
        {
            start=end=i;
            prev=0;
            if(i==(*memory_size)-1)
                fprintf(stdout,"Addresses [%d:%d] Unuesd\n",start,end);
        }
        else if(prev==0&&memory[i]==0)
        {
            end=i;
            if(i==(*memory_size)-1)
                fprintf(stdout,"Addresses [%d:%d] Unuesd\n",start,end);
        }
        if(memory[i]==1)
        {
            struct processor* proc=search_by_address(i);
            fprintf(stdout,"Addresses [%d:%d] Process %s\n",proc->starting_ad
            i=proc->ending_address;
            prev=1;
        }
    }

}

int first_fit(int size)
{
    int start=0;
    int end=start;
    int prev=1;
    int sum=0;

    for(int i=0;i<(*memory_size);i++)
    {
        if(prev==1&&memory[i]==0)
        {
            start=i;
            end=start;
            prev=0;
            sum++;
        }
        else if(prev==0&&memory[i]==0)
        {
            sum++;
            end=i;
        }
        else if(prev==0&&memory[i]==1)
        {
            prev=1;
            sum=0;
        }

        if(sum>=size)
        {
            return start;
        }
    }
    return -1;
}
int worst_fit(int size)
{
    int start=0;
    int end=start;
    int prev=1;
    int sum=0;

    int max=-1;
    int max_start=-1;
    for(int i=0;i<(*memory_size);i++)
    {
        if(prev==1&&memory[i]==0)//start of contious empty memory
        {
            start=i;
            prev=0;
            sum++;
        }
        else if(prev==0&&memory[i]==0&&i!=((*memory_size)-1))
        {
            sum++;
        }
        else if(prev==0&&memory[i]==1)//end of contious empty memory
        {
            prev=1;
            sum=0;
        }

        if((i==((*memory_size)-1)&&memory[i]==0)||(memory[i]==0&&memory[i+1]=
        {
            if(sum>max)
            {
                max=sum;
                max_start=start;
            }
        }

    }
    if(max_start!=-1)
        return max_start;
    else
        return -1;
}
int best_fit(int size)
{
    int start=0;
    int end=start;
    int prev=1;
    int sum=0;

    int min=(*memory_size)+1;
    int min_start=-1;
    for(int i=0;i<(*memory_size);i++)
    {
        if(prev==1&&memory[i]==0)//start of contious empty memory
        {
            start=i;
            prev=0;
            sum++;
        }
        else if(prev==0&&memory[i]==0&&i!=((*memory_size)-1))
        {
            sum++;
        }
        else if(prev==0&&memory[i]==1)//end of contious empty memory
        {
            prev=1;
            sum=0;
        }

        if((i==((*memory_size)-1)&&memory[i]==0)||(memory[i]==0&&memory[i+1]=
        {
            if(sum<min&&sum>=size)
            {
                min=sum;
                min_start=start;
            }
        }
    }
    if(min_start!=-1)
        return min_start;
    else
        return -1;
}

void insert(char* name, int starting_address, int ending_address)
{
    if(head==NULL)
    {
        head=(struct processor*) malloc(sizeof(struct processor));
        head->name=strdup(name);
        head->starting_address=starting_address;
        head->ending_address=ending_address;
        head->next=NULL;
    }
    else
    {
        struct processor * temp=(struct processor*) malloc(sizeof(struct proc
        temp->name=strdup(name);
        temp->starting_address=starting_address;
        temp->ending_address=ending_address;

        temp->next=head->next;
        head->next=temp;
    }
}
struct processor * search_by_address(int i)
{
    if(head==NULL)
        return NULL;

    struct processor *temp=head;
    struct processor *prev;
    while (temp!=NULL)
    {
        if(temp->starting_address==i)
            return temp;
        prev=temp;
        temp=temp->next;
    }
    return NULL;
}
struct processor * search_by_name(char * name)
{
    if(head==NULL)
```

```
        return NULL;

    struct processor *temp=head;
    struct processor *prev;
    while (temp!=NULL)
    {
        if(strcmp(name,temp->name)==0)
            return temp;
        prev=temp;
        temp=temp->next;
    }
    return NULL;
}
struct processor * delete_by_name(char * name)
{
    if(head==NULL)
        return NULL;

    struct processor *temp=head;
    struct processor *prev;
    while (temp!=NULL)
    {
        if(strcmp(name,temp->name)==0)
        {
            if(temp==head)
                head=temp->next;
            else
                prev->next=temp->next;
            return temp;
        }

        prev=temp;
        temp=temp->next;
    }
    return NULL;
}
```

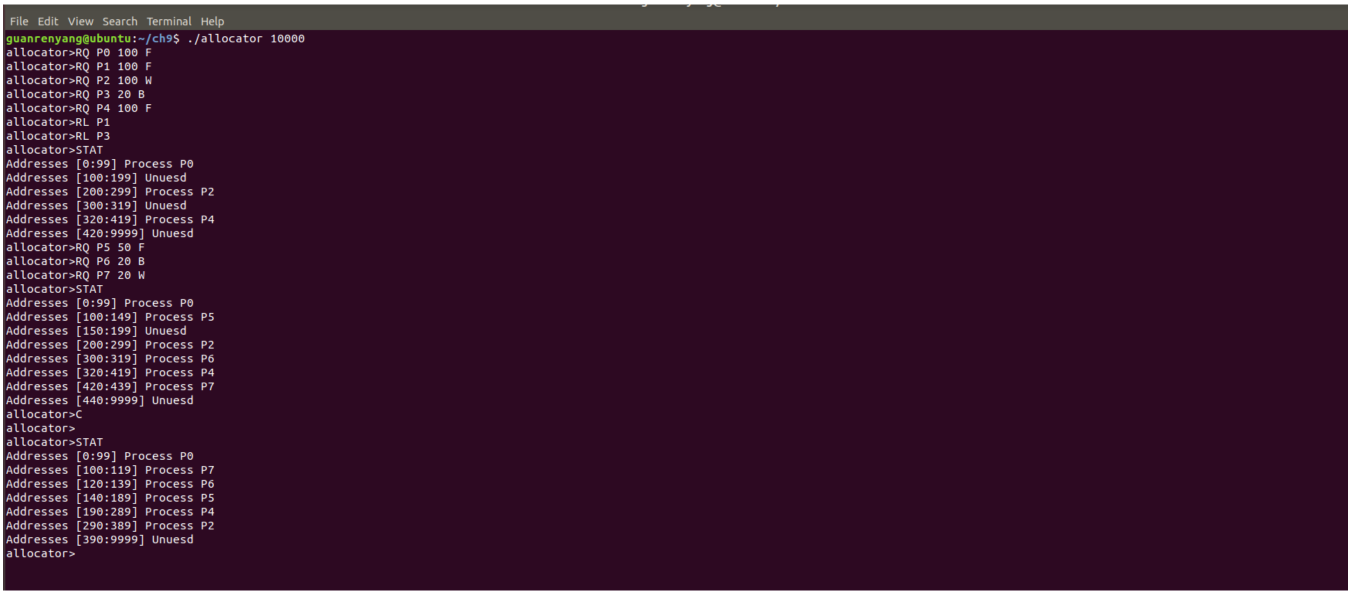
5 实验结果

实验结果如图 1。可见 RQ 中 F、W、B 三个功能，RL、STAT 与 C 都很好地完成。

6 总结与思考

通过此次实验我实现了 First-Fit、Best-Fit 与 Worst-Fit 三种内存分配算法。且实现了内存碎片压缩。在此过程中我对硬件的理解与调度算法理解地更加深入了，并且我更好地理解了连续内存分配的效率高的原因。

同时我也知道了这种直接内存分配的限制所在，促使我更好地理解了为什么要使用虚拟内存与物理内存分离的做法。



```
guanrenyang@ubuntu:~/ch9$ ./allocator 10000
allocator>RQ P0 100 F
allocator>RQ P1 100 F
allocator>RQ P2 100 W
allocator>RQ P3 20 B
allocator>RQ P4 100 F
allocator>RL P1
allocator>RL P3
allocator>STAT
Addresses [0:99] Process P0
Addresses [100:199] Unuesd
Addresses [200:299] Process P2
Addresses [300:319] Unuesd
Addresses [320:419] Process P4
Addresses [420:9999] Unuesd
allocator>RQ P5 50 F
allocator>RQ P6 20 B
allocator>RQ P7 20 W
allocator>STAT
Addresses [0:99] Process P0
Addresses [100:149] Process P5
Addresses [150:199] Unuesd
Addresses [200:299] Process P2
Addresses [300:319] Process P6
Addresses [320:419] Process P4
Addresses [420:439] Process P7
Addresses [440:9999] Unuesd
allocator>C
allocator>
allocator>STAT
Addresses [0:99] Process P0
Addresses [100:119] Process P7
Addresses [120:139] Process P6
Addresses [140:189] Process P5
Addresses [190:289] Process P4
Addresses [290:389] Process P2
Addresses [390:9999] Unuesd
allocator>
```

图 1: Result