Project 7--Contiguous Memory Allocation

陈思远 518021910567

1. 实验内容和目标

1. 使用不同的算法(`first-fit`,`best-fit`,`worst-fit`)实现连续内存空间的分配。
2. 释放连续的内存块。
3. 将内存中未使用的hole压缩成一个block。

2. 实验过程

2.1 要求与实现

这次实验需要模拟内存的连续分配。要实现的功能有：为某个进程申请一个连续的内存块、释放一个进程的连续内存块、将内存中的空位集中为一整块、显示当前内存的分配情况。内存范围由命令行输入。

2.1.1 内存块数据结构

把每个内存块作为一个结构体，储存这个块的起始地址、终止地址和所属的进程。如果这个块是空的（不属于任何进程），那么它的`process=-1`。

```
typedef struct Block{
    int begin;
    int end;
    int process;
}block;
```

2.1.2 main函数

我采用了自顶向下的设计方法。在main函数中，首先从命令行读入初始的内存大小，并将整个内存空间作为一个未被分配的内存块进行初始化。接着，用户可以输入不同的指令，进行申请空间、释放空间、合并块等操作，这些操作由不同的函数分别实现。

```
int main(int argc,char* argv[]) {
    //get command line arguments
    if(argc!=2){
```

```
        printf("error: wrong argument!\n");
        return 1;
    }
    init_memory=atoi(argv[1]);

    //initialize
    blocks[0].begin=0;
    blocks[0].end=init_memory-1;
    blocks[0].process=-1;
    block_num=1;

    while(1){
        printf("allocator>");
        char command[5];
        char space;
        scanf("%s",command);
        scanf("%c",&space);
        if(strcmp(command,"RQ")==0){        //申请空间
            int process;
            int req_mem;
            char type;
            scanf("P%d %d %c",&process,&req_mem,&type);
            scanf("%c",&space);
            request_mem(process,req_mem,type);
        }else{
            if(strcmp(command,"RL")==0){        //释放空间
                int process;
                scanf("P%d",&process);
                scanf("%c",&space);
                release_mem(process);
            }else{
                if(strcmp(command,"C")==0){     //合并内存块
                    compact_mem();
                }else{
                    if(strcmp(command,"STAT")==0){      //打印内存状态
                        report();
                    }else{
                        if(strcmp(command,"X")==0){     //退出
                            break;
                        }else {                         //指令错误
                            printf("ERROR! invalid command!\n");
                        }
                    }
                }
            }
        }
    }
    return 0;
}
```

2.1.3 申请内存空间

调用`request_mem()`函数申请内存空间。分配内存的方式有三种，`first-fit`分配首个足够大的孔；`best-fit`分配最小的足够大的孔；`worst-fit`分配最大的孔。每种分配方式都需要先找到符合条件的分配位置，之后再调用`allocate_mem()`函数分配内存空间。

```
int request_mem(int proc,int request,char type){
```

```
    if(type=='F'){      //first-fit
        for(int i=0;i<block_num;i++){
            if(blocks[i].process==-1 && (blocks[i].end-
blocks[i].begin)>=request){
                allocate_mem(i,request,proc);
                return 0;
            }
        }
        return -1;
    }else{
        if(type=='B'){      //best-fit
            int minhole=init_memory;
            int min_id=-1;
            for(int i=0;i<block_num;i++){
                int size=blocks[i].end-blocks[i].begin+1;
                if(blocks[i].process==-1 && size<minhole && size>=request){
                    minhole=blocks[i].end-blocks[i].begin;
                    min_id=i;
                }
            }
            if(min_id==-1){
                return -1;
            }else{
                allocate_mem(min_id,request,proc);
                return 0;
            }
        }else{
            if(type=='W'){      //worst-fit
                int maxhole=0;
                int max_id=-1;
                for(int i=0;i<block_num;i++){
                    int size=blocks[i].end-blocks[i].begin+1;
                    if(blocks[i].process==-1 && size>=request && size>maxhole){
                        maxhole=size;
                        max_id=i;
                    }
                }
                if(max_id==-1){
                    return -1;
                }else {
                    allocate_mem(max_id, request, proc);
                    return 0;
                }
            }else{
                printf("ERROR! invalid type\n");
                return -1;
            }
        }
    }
}
```

`allocate_mem()` 函数会在选定的位置分配一个连续的内存块，若整个hole都被填满，则只需修改该hole所属的进程信息；若没有填满，原来的hole会分成一个已被分配的内存块和一个稍小的hole。

```
void allocate_mem(int block_id,int request,int proc){
    if((blocks[block_id].end-blocks[block_id].begin+1)>request){    //未填满
        int end=blocks[block_id].end;
```

```
        blocks[block_id].end=blocks[block_id].begin+request-1;
        blocks[block_id].process=proc;
        for(int i=block_num;i>block_id+1;i--){
            blocks[i]=blocks[i-1];
        }
        blocks[block_id+1].process=-1;
        blocks[block_id+1].begin=blocks[block_id].end+1;
        blocks[block_id+1].end=end;
        block_num++;
    }else{                  //整个hole被填满
        blocks[block_id].process=proc;
    }
}
```

2.1.4 释放内存空间

调用`release_mem()`释放内存空间。若释放的内存空间前后均已被分配，那么只需将该内存块的process设为-1；若前后两块有未被分配的内存块，就需要调用`merge_blocks()`函数将它们合并。

```
void release_mem(int proc){
    for(int i=0;i<block_num;i++){
        if(blocks[i].process==proc){
            blocks[i].process=-1;
            if(i>0 && blocks[i-1].process==-1){
                merge_blocks(i-1,i);
                if(blocks[i].process==-1){
                    merge_blocks(i-1,i);
                }
            }else{
                if(blocks[i+1].process==-1){
                    merge_blocks(i,i+1);
                }
            }
            break;
        }
    }
}
```

```
void merge_blocks(int block1,int block2){
    if(block1<block2){
        blocks[block1].end=blocks[block2].end;
        for(int i=block1+1;i<block_num-1;i++){
            blocks[i]=blocks[i+1];
        }
        block_num--;
    }
}
```

2.1.5 合并内存块

调用`compact_mem()`合并内存块。采用一个临时数组按顺序储存所有已分配的内存块，最后一个块未分配。然后将临时数组赋给内存块`blocks`数组，并修改内存块数量。

```
int compact_mem(){
    block tmp[100];
    int num=0;
```

```
    for(int i=0;i<block_num;i++){
        if(blocks[i].process!=-1){
            if(num==0){
                tmp[num].begin=0;
                tmp[num].end=blocks[i].end-blocks[i].begin;
                tmp[num].process=blocks[i].process;
                num++;
            }else{
                tmp[num].begin=tmp[num-1].end+1;
                tmp[num].end=tmp[num].begin+blocks[i].end-blocks[i].begin;
                tmp[num].process=blocks[i].process;
                num++;
            }
        }
    }
    if(tmp[num-1].end<init_memory){
        tmp[num].begin=tmp[num-1].end+1;
        tmp[num].end=init_memory-1;
        tmp[num].process=-1;
        num++;
    }
    block_num=num;
    for(int i=0;i<num;i++){
        blocks[i]=tmp[i];
    }
}
```

2.1.6 输出当前内存状态

```
void report(){
    printf("-------------------\n");
    for(int i=0;i<block_num;i++){
        if(blocks[i].process==-1){
            printf("Address [%d:%d] Unused\n",blocks[i].begin,blocks[i].end);
        }else{
            printf("Address [%d:%d] Process
P%d\n",blocks[i].begin,blocks[i].end,blocks[i].process);
        }
    }
    printf("-------------------\n");
}
```

2.2 结果

编译并运行程序，通过命令行参数初始化内存为1MB (1048576Byte)。首先用first-fit算法，为四个进程申请四块大小不等的连续的内存空间。之后释放掉进程P1的空间，从而出现了碎片。接着为进程P4用best-fit算法申请300Byte的空间，为进程P5用worst-fit算法申请200Byte的空间。再释放掉进程P3的空间，由此出现了两个孔。最后合并内存块，将所有的hole合成一个内存块。从运行截图中可以看出，每一步的结果都是正确的。

```
chesiy@ubuntu:~/osProject/Proj7$ gcc memory.c
chesiy@ubuntu:~/osProject/Proj7$ ./a.out 1048576
allocator>RQ P0 1000 F
allocator>RQ P1 1500 F
allocator>RQ P2 2000 F
allocator>RQ P3 2500 F
allocator>STAT
------------------
Address [0:999] Process P0
Address [1000:2499] Process P1
Address [2500:4499] Process P2
Address [4500:6999] Process P3
Address [7000:1048575] Unused
------------------
allocator>RL P1
allocator>STAT
------------------
Address [0:999] Process P0
Address [1000:2499] Unused
Address [2500:4499] Process P2
Address [4500:6999] Process P3
Address [7000:1048575] Unused
------------------
allocator>RQ P4 300 B
allocator>RQ P5 200 W
allocator>STAT
------------------
Address [0:999] Process P0
Address [1000:1299] Process P4
Address [1300:2499] Unused
Address [2500:4499] Process P2
Address [4500:6999] Process P3
Address [7000:7199] Process P5
Address [7200:1048575] Unused
------------------
allocator>RL P3
allocator>STAT
------------------
Address [0:999] Process P0
Address [1000:1299] Process P4
Address [1300:2499] Unused
Address [2500:4499] Process P2
Address [4500:6999] Unused
Address [7000:7199] Process P5
Address [7200:1048575] Unused
------------------
allocator>C
allocator>STAT
------------------
Address [0:999] Process P0
Address [1000:1299] Process P4
Address [1300:3299] Process P2
Address [3300:3499] Process P5
Address [3500:1048575] Unused
------------------
allocator>X
chesiy@ubuntu:~/osProject/Proj7$
```

3. 总结与思考

在本次project中，我模拟了连续内存分配的过程，复习了`first-fit`，`best-fit`，`worst-fit`三种分配算法，这让我对课堂上学过的知识有了更深入的理解。这次project难度不大，但实现过程中需要重视细节。计算内存块大小、内存块终止地址的时候加一、减一容易弄混。此外，在写程序之前，需要仔细设计数据结构，考虑用链表实现和数组实现哪种更加合适，以及已分配的内存块和未分配的内存块是否需要用两个数组分开储存。不同的数据结构实现起来的繁简程度差异较大，因此需要事先认真思考。