实验报告

【实验题目】

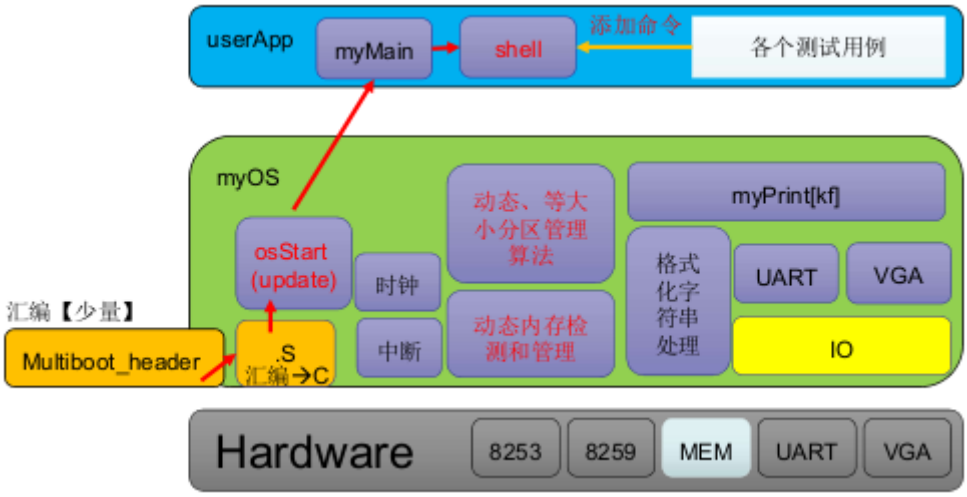
Memory Management

【实验要求】

1. 内存检测，确定动态内存的范围
2. 提供动态分区管理机制 dPartition
3. 提供等大小固定分区管理机制 ePartition
4. 使用动态分区管理机制来管理所有动态内存
5. 提供 kmalloc/kfree 和 malloc/free 两套接口，分别提供给内核和用户
6. 通过自测程序

【实验原理】

- 软件架构和功能



软件整个生命周期涉及三个层次：硬件层，操作系统层，用户应用层，其功能分别如下：

1. Hardware层：
① 基于UART和VGA的IO功能
② 基于PIC i8259可中断控制器的中断处理功能
③ 基于PIT i8253可编程间隔定时器的定时功能
④ 内存空间
2. myOS层：
① 系统初始化并为C语言准备环境：栈的建立以及BSS段的0初始化，为C程序的执行提供运行环境
② C语言函数封装库：UART和VGA的IO接口函数库，开关中断的接口函数
③ IDT的初始化及中断处理程序
④ 基于定时器的时钟显示模块
⑤ 基于硬件内存空间的管理与分配
3. UserApp层：
① 实现shell终端：接受相关指令输入并执行
② 内存管理与分配测试程序

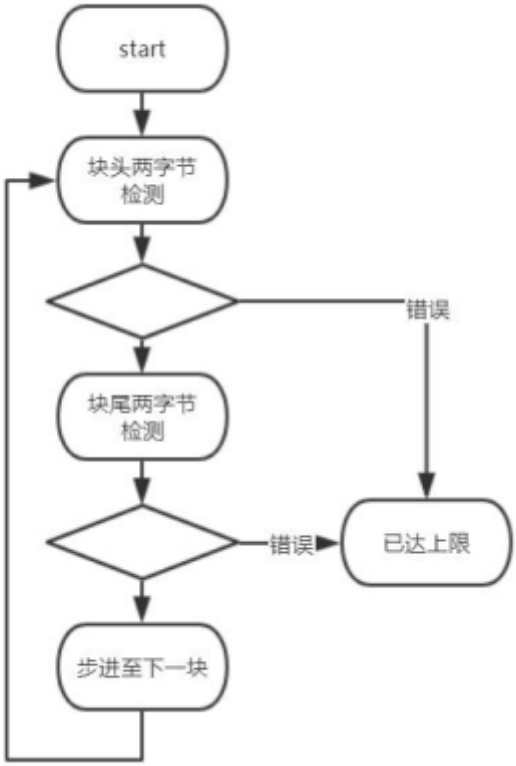
- 软件执行流程

Multiboot header(启动) ==> start32.S(准备上下文) ==> osStart.c(初始化操作系统) ==> myMain(用户程序入口) ==> shell(终端)

本实验的主流程如上：
1. 在 multiboot_header 中完成系统的启动
2. 通过start32.S完成系统初始化并为C语言准备环境，并调用osStart.c进入C程序
3. 在 osStart.c中完成初始化8259A，初始化8253，清屏及内存初始化等操作，调用myMain，进入userApp
4. 运行 myMain 中的代码，进行时钟设置， shell 初始化，内存测试初始化等操作，启动shell
5. 进入 shell 程序，等待与用户交互

- 新增功能模块及其实现

1. 内存检测模块



该模块用于检测给定内存起始位置后的可用内存大小，主要思路是，按给定的步长grainize，逐块读写(为加速操作分别每块读写头尾各两个字节)，校对，如果出错，返回已经通过检测的内存大小

```
void memTest(unsigned long start, unsigned long grainSize){
    /*本函数需要实现！！！*/
    /*功能：检测算法
        这一个函数对应实验讲解ppt中的第一大功能-内存检测。
        本函数的功能是检测从某一个地址开始的内存是否可用，具体算法就可以用ppt上说的写了读看是否一致。
    注意点两个：
    1、开始的地址要大于1M，需要做一个if判断。
    2、grainsize不能太小，也要做一个if判断
    */
    const int min_grain_size = 0x1000;
```

```
    //内存起始地址要大于1M
    if(start < 0x100000){
        myPrintk(0x07,"The start address require >= 1M!\n");
    }
    else {
        grainSize = grainSize > min_grain_size ? grainSize : min_grain_size;

        char find_start = 0;
        unsigned char old[2];
        unsigned long i = start;

        while(1){
            // 一个内存块的头两个字节,两次读写校对
            old[0] = *((unsigned char*)i);
            old[1] = *((unsigned char*)(i+1));
            *((unsigned char*)i) = 0x55;
            *((unsigned char*)(i+1)) = 0xAA;
            if(*((unsigned char*)i) != 0x55 || *((unsigned char*)(i+1)) !=
0xAA){
                break;
            }
            *((unsigned char*)i) = 0xAA;
            *((unsigned char*)(i+1)) = 0x55;
            if(*((unsigned char*)i) != 0xAA || *((unsigned char*)(i+1)) !=
0x55){
                break;
            }
            *((unsigned char*)i) = old[0];
            *((unsigned char*)(i+1)) = old[1];

            // 一个内存块的尾两个字节,两次读写校对
            old[0] = *((unsigned char*)(i+grainSize-2));
            old[1] = *((unsigned char*)(i+1+grainSize-2));
            *((unsigned char*)(i+grainSize-2)) = 0x55;
            *((unsigned char*)(i+1+grainSize-2)) = 0xAA;
            if(*((unsigned char*)(i+grainSize-2)) != 0x55 || *((unsigned
char*)(i+1+grainSize-2)) != 0xAA){
                break;
            }
            *((unsigned char*)(i+grainSize-2)) = 0xAA;
            *((unsigned char*)(i+1+grainSize-2)) = 0x55;
            if(*((unsigned char*)(i+grainSize-2)) != 0xAA || *((unsigned
char*)(i+1+grainSize-2)) != 0x55){
                break;
            }
            *((unsigned char*)(i+grainSize-2)) = old[0];
            *((unsigned char*)(i+1+grainSize-2)) = old[1];
            //找到有效内存起始地址
            if(find_start == 0){
                pMemStart = i;
                find_start = 1;
            }
            i += grainSize;
        }
        pMemSize = i-pMemStart;
        myPrintk(0x7,"MemStart: 0x%x  \n",pMemStart);
        myPrintk(0x7,"MemSize:  0x%x  \n",pMemSize);
    }
```

```
}
```

2. 动态内存管理机制

动态内存管理的模型如上，该模块对内存管理的具体流程如下：

1. 对一块内存初始化，在内存的首部划分出一块空间放入dPartition结构体，具体作为整个内存的管理数据结构，并将余下的内存划分为一个EMB块(头部的EMB管理信息与余下的空闲空间)，通过dPartition与EMB的联结构成初始的空闲内存静态链表
2. 对申请分配内存的请求，在空闲内存链表里查找第一块满足用户需求的内存块，将内存地址返回给用户，同时维护内部的空闲链表结构
3. 对释放内存的请求，根据释放块的地址找到其在空闲链表中的相邻块，依据是否相接进行合并，同时维护内部的空闲链表结构

```
//dPartition 是整个动态分区内存的数据结构
typedef struct dPartition{
    unsigned long size;
    unsigned long firstFreeStart;
} dPartition;   //共占8个字节

#define dPartition_size ((unsigned long)0x8)

void showdPartition(struct dPartition *dp){
    myPrintk(0x5,"dPartition(start=0x%x, size=0x%x,
firstFreeStart=0x%x)\n", dp, dp->size,dp->firstFreeStart);
}


// EMB每一个block的数据结构，userdata可以暂时不用管。
typedef struct EMB{
    unsigned long size;
    union {
        unsigned long nextStart;    // if free: pointer to next block
        unsigned long userData;     // if allocated, belongs to user
    };
} EMB;  //共占8个字节

#define EMB_size ((unsigned long)0x8)

void showEMB(struct EMB * emb){
    myPrintk(0x3,"EMB(start=0x%x, size=0x%x, nextStart=0x%x)\n", emb,
emb->size, emb->nextStart);
```

```
}

unsigned long dPartitionInit(unsigned long start, unsigned long
totalSize){
    //本函数需要实现！！！
    /*功能：初始化内存。
    1.在地址start处，首先是要有dPartition结构体表示整个数据结构(也即句柄)。
    2.然后，一整块的EMB被分配（以后使用内存会逐渐拆分），在内存中紧紧跟在dP后面，然后
dP的firstFreeStart指向EMB。
    3.返回start首地址(也即句柄)。
    注意有两个地方的大小问题：
        第一个是由于内存肯定要有一个EMB和一个dPartition，totalSize肯定要比这两个
加起来大。
        第二个注意EMB的size属性不是totalsize，因为dPartition和EMB自身都需要要占
空间。

    */
    // 确保待初始化的块满足大小要求
    if(totalSize <= EMB_size+dPartition_size){
        myPrintk(0x07,"Fail! The Init Block Size Must Be Larger than %d
Bytes\n",EMB_size+dPartition_size);
        return 0;
    }
    // 为待管理内存建立dPartition与EMB结构，同时建立初始的空闲静态链表结构
    dPartition *dp = (dPartition *)start;
    dp->size = totalSize - dPartition_size-EMB_size;
    EMB        *emb = (EMB*)(start+dPartition_size);
    dp->firstFreeStart = (unsigned long)emb;
    emb->size = totalSize - dPartition_size-EMB_size;
    emb->nextStart = 0;

    return start;
}

void dPartitionWalkByAddr(unsigned long dp){
    //本函数需要实现！！！
    /*功能：本函数遍历输出EMB 方便调试
    1.先打印dP的信息，可调用上面的showdPartition。
    2.然后按地址的大小遍历EMB，对于每一个EMB，可以调用上面的showEMB输出其信息

    */
    dPartition* dp_ptr = (dPartition*)dp;

    showdPartition(dp_ptr);

    //依据内存块自身的空闲链表便利各个空闲块
    unsigned long emb = dp_ptr->firstFreeStart;
    while(1){
        showEMB((EMB*)emb);
        emb = ((EMB*)emb)->nextStart;
        if(emb==0)
            break;
    }
}

//=================firstfit, order: address, low--
>high====================
/**
```

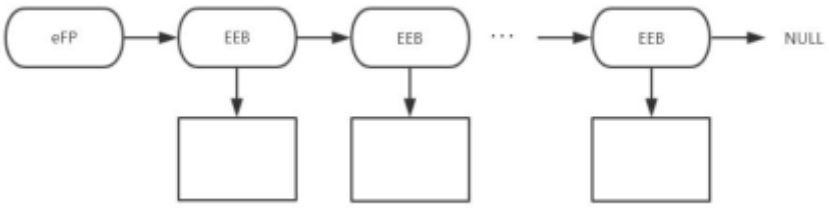
```
 * return value: addr (without overhead, can directly used by user)
**/

unsigned long dPartitionAllocFirstFit(unsigned long dp, unsigned long
size){
    //本函数需要实现！！！
    /*功能：分配一个空间
    1.使用firstfit的算法分配空间，当然也可以使用其他fit，不限制。
    2.成功分配返回首地址，不成功返回0
    3.从空闲内存块组成的链表中拿出一块供我们来分配空间(如果提供给分配空间的内存块空间
大于size，我们还将把剩余部分放回链表中)，并维护相应的空闲链表以及句柄
    注意的地方：
        1.EMB类型的数据的存在本身就占用了一定的空间。

    */

    dPartition* dp_ptr = (dPartition*)dp;

    unsigned long emb_p = dp_ptr->firstFreeStart;
    unsigned long emb_q = dp_ptr->firstFreeStart;

    /**--------------------------------通过前后双指针法找到第一个符合用户使用大
小要求的空闲块，
                                                    并维护分配块与前后块的空闲链表关
系-----------------------------------**/

    while(emb_q){
        if(((EMB*)emb_q)->size+4 > size){
            //分配后的剩余空间足够建立EMB结构
            if(((EMB*)emb_q)->size +4- size >= EMB_size ){
                EMB* new_emb = (EMB*) (emb_q+4 + size);
                new_emb->nextStart = ((EMB*)emb_q)->nextStart;
                new_emb->size   = ((EMB*)emb_q)->size +4- size -
EMB_size;
                //如果分配的是第一个块
                if(emb_p == dp_ptr->firstFreeStart){
                    dp_ptr->firstFreeStart = (unsigned long)new_emb;
                }
                else {
                    ((EMB*)emb_p)->nextStart = new_emb;
                }
                dp_ptr->size = dp_ptr->size -size - EMB_size+4;
                ((EMB*)emb_q)->size = size;
            }
            //分配后的剩余空间不够建立EMB结构，直接全部分配给用户
            else {
                if(emb_p == dp_ptr->firstFreeStart){
                    dp_ptr->firstFreeStart = ((EMB*)emb_q)->nextStart;
                }
                else {
                    ((EMB*)emb_p)->nextStart = ((EMB*)emb_q)->nextStart;
                }
                dp_ptr->size = dp_ptr->size -((EMB*)emb_q)->size;
            }
            // 返回有效空闲空间的地址
            return emb_q+4;
        }
        else {        //寻找下一块
```

```
            emb_p = emb_q;
            emb_q = ((EMB*)emb_q)->nextStart;
        }
    }
    // 没有空闲空间或没有符合大小要求的空闲空间可以分配
    return 0;
}

/*
 *r
 */
unsigned long dPartitionFreeFirstFit(unsigned long dp, unsigned long
start){
    //本函数需要实现！！！
    /*功能：释放一个空间
    1.按照对应的fit的算法释放空间
    2.注意检查要释放的start~end这个范围是否在dp有效分配范围内
        返回1 没问题
        返回0 error
    3.需要考虑两个空闲且相邻的内存块的合并
    */
    dPartition* dp_ptr = (dPartition*)dp;
    unsigned long emb_p = dp_ptr->firstFreeStart;
    unsigned long emb_q = dp_ptr->firstFreeStart;
    unsigned long size = ((EMB*)(start-4))->size;

    /** ------------------------------使用双指针法根据待释放块地址找到其在空闲
链表相邻块，
                                        并维护待释放块与前后块的空闲链表关系 -----
--------------------------**/

    // 释放位置非法(不符合内存分配后的地址要求)
    if(start < dp + dPartition_size + 4){
        return 0;
    }

    // 释放的位置在所有空闲块之前，要维护dPartition信息
    if(start < emb_p){
        // 释放位置非法(不符合内存分配后的地址要求)
        if(size +start > emb_p){
            return 0;
        }
        else {
            dp_ptr->firstFreeStart = start -4;
            if(size +start == emb_p){
                // 与空闲链表中其后空间块相接，可以合并
                dp_ptr->size +=  ((EMB*)(start-4))->size-4 + EMB_size;
                ((EMB*)(start-4))->nextStart = ((EMB*)emb_p)->nextStart;
                ((EMB*)(start-4))->size = ((EMB*)(start-4))->size +
((EMB*)emb_p)->size + EMB_size-4;
            }
            else {
                ((EMB*)(start-4))->nextStart = emb_p;
                dp_ptr->size +=  ((EMB*)(start-4))->size -4 ;
            }
        }
    }
    // 找到待释放块在空闲链表中的前后相邻块(可能不存在)
```

```
    else {
        while(emb_q){
            if(emb_q < start){
                emb_p = emb_q;
                emb_q = ((EMB*)emb_q)->nextStart;
            }
            //找到待释放块的在空闲链表中的后块
            else {
                // 释放位置非法(不符合内存分配后的地址要求)
                if(size+ 4 > emb_q -emb_p-EMB_size-((EMB*)emb_p)->size){
                return 0;
                }
                else {
                    // 待释放块与前后块地址都相接
                    if(emb_p+EMB_size+((EMB*)emb_p)->size + 4 == start
&& start + size == emb_q){
                        ((EMB*)emb_p)->nextStart = ((EMB*)emb_q)-
>nextStart;
                        ((EMB*)emb_p)->size = ((EMB*)emb_p)->size + 4+
size + EMB_size + ((EMB*)emb_q)->size;
                        dp_ptr->size +=  ((EMB*)(start-4))->size
+4+EMB_size;
                    }
                    // 待释放块只与前块地址相接
                    else if(emb_p+EMB_size+((EMB*)emb_p)->size +
EMB_size == start ){
                        ((EMB*)emb_p)->size = ((EMB*)emb_p)->size + 4+
size ;
                        dp_ptr->size +=  ((EMB*)(start-4))->size +4;
                    }
                    // 待释放块只与后块地址相接
                    else if(start + size == emb_q){
                        ((EMB*)(start-4))->size = size + EMB_size +
((EMB*)emb_q)->size-4;
                        ((EMB*)(start-4))->nextStart = ((EMB*)emb_q)-
>nextStart;
                        ((EMB*)emb_p)->nextStart = (EMB*)(start-4);
                        dp_ptr->size +=  ((EMB*)(start-4))->size +4;
                    }
                     // 待释放块与前后块地址都不相接
                    else {
                        ((EMB*)(start-4))->size = size;
                        ((EMB*)(start-4))->nextStart = emb_q;
                        ((EMB*)emb_p)->nextStart = (EMB*)(start-4);
                        dp_ptr->size +=  ((EMB*)(start-4))->size -4;
                    }
                    break;
                }
            }
        }
        // 待释放块地址在所有空闲块之后或此时空闲链表为空
        if(emb_q==0){
            // 待释放块地址与前块地址相接
            if(emb_p+EMB_size+((EMB*)emb_p)->size + 4 == start){
                ((EMB*)emb_p)->size = ((EMB*)emb_p)->size + 4+ size;
                dp_ptr->size +=  ((EMB*)(start-4))->size +4;
            }
            else {
```

```
                ((EMB*)emb_p)->nextStart = start -4;
                ((EMB*)(start-4))->nextStart = 0;
                dp_ptr->size +=  ((EMB*)(start-4))->size -4;
            }
        }
    }
    return 1;
}
```

3. 等大小分区管理机制



等大小内存管理的模型如上，该模块对内存管理的具体流程如下：

1．对一块内存初始化，在内存的首部划分出一块空间放入eFPartition结构体，具体作为整个内存的管理数据结构，并将余下的内存划分为n个等大小的EEB块(头部的管理信息与余下的有效空闲空间)，需要注意的是初始化的内存大小要符合内存对齐要求与大小要求(具体实现通过eFPartitionTotalSize函数提前计算)，然后将eFPartition与EEB间联结形成初始的静态空闲链表

2．对申请分配内存的请求，将空闲内存链表中的第一块空闲EEB分配给用户(可能为空)，同时维护内部的空闲链表结构

3．对释放内存的请求，根据释放块的地址找到其在空闲链表中的相邻块，插入其间即可

```
// 一个EEB表示一个空闲可用的Block
typedef struct EEB {
    unsigned long next_start;
}EEB;   //占4个字节

#define EEB_size 4

void showEEB(struct EEB *eeb){
    myPrintk(0x7,"EEB(start=0x%x, next=0x%x)\n", eeb, eeb->next_start);
}

//eFPartition是表示整个内存的数据结构
typedef struct eFPartition{
    unsigned long totalN;
    unsigned long perSize;  //unit: byte
    unsigned long firstFree;
}eFPartition;   //占12个字节

#define eFPartition_size 12

void showeFPartition(struct eFPartition *efp){
    myPrintk(0x5,"eFPartition(start=0x%x, totalN=0x%x, perSize=0x%x,
firstFree=0x%x)\n", efp, efp->totalN, efp->perSize, efp->firstFree);
}

void eFPartitionWalkByAddr(unsigned long efpHandler){
    //本函数需要实现！！！
    /*功能：本函数是为了方便查看和调试的。
```

```
    1、打印eFPartiiton结构体的信息，可以调用上面的showeFPartition函数。
    2、遍历每一个EEB，打印出他们的地址以及下一个EEB的地址（可以调用上面的函数showEEB）

    */
    eFPartition* eFP_ptr = (eFPartition*)efpHandler;
    showeFPartition(eFP_ptr);
    unsigned long  eeb = eFP_ptr->firstFree;
    while(eeb!=0){
        showEEB((EEB*)eeb);
        eeb = ((EEB*)eeb)->next_start;
    }

}


unsigned long eFPartitionTotalSize(unsigned long perSize, unsigned long n){
    //本函数需要实现！！！
    /*功能：计算占用空间的实际大小，并将这个结果返回
    1.根据参数persize（每个大小）和n个数计算总大小，注意persize的对齐。
        例如persize是31字节，你想8字节对齐，那么计算大小实际代入的一个块的大小就是32字节。
    2.同时还需要注意“隔离带”EEB的存在也会占用4字节的空间。
        typedef struct EEB {
            unsigned long next_start;
        }EEB;
    3.最后别忘记加上eFPartition这个数据结构的大小，因为它也占一定的空间。
    */

    //4字节对齐
    perSize = (perSize&0x11 !=0)? ((perSize>>2)+1)<<2 : perSize;

    return n*(perSize+4)+eFPartition_size;


}

unsigned long eFPartitionInit(unsigned long start, unsigned long perSize,
unsigned long n){
    //本函数需要实现！！！
    /*功能：初始化内存
    1.需要创建一个eFPartition结构体，需要注意的是结构体的perSize不是直接传入的参数perSize，需要对齐。结构体的next_start也需要考虑一下其本身的大小。
    2.就是先把首地址start开始的一部分空间作为存储eFPartition类型的空间
    3.然后再对除去eFPartition存储空间后的剩余空间开辟若干连续的空闲内存块，将他们连起来构成一个链。注意最后一块的EEB的nextstart应该是0
    4.需要返回一个句柄，也即返回eFPartition *类型的数据
    注意的地方：
        1.EEB类型的数据的存在本身就占用了一定的空间。
    */
    //4字节对齐
    unsigned long i=0;

    perSize = (perSize&0x3 !=0)? ((perSize>>2)+1)<<2 : perSize;

    eFPartition* eFP_ptr = (eFPartition*)start;
    eFP_ptr->perSize = perSize;
    eFP_ptr->totalN = n;
    while(i<n-1){
```

```
        ((EEB*)(start+eFPartition_size+i*(EEB_size+perSize)))->next_start =
 start+eFPartition_size+(i+1)*(EEB_size+perSize);
        i++;
    }
    ((EEB*)(start+eFPartition_size+i*(EEB_size+perSize)))->next_start = 0;

    eFP_ptr->firstFree = start + eFPartition_size;
    return  start;

}


unsigned long eFPartitionAlloc(unsigned long EFPHandler){
    //本函数需要实现！！！
    /*功能：分配一个空间
    1.本函数分配一个空闲块的内存并返回相应的地址，EFPHandler表示整个内存的首地址
    2.事实上EFPHandler就是我们的句柄，EFPHandler作为eFPartition *类型的数据，其存放
了我们需要的firstFree数据信息
    3.从空闲内存块组成的链表中拿出一块供我们来分配空间，并维护相应的空闲链表以及句柄
    注意的地方：
        1.EEB类型的数据的存在本身就占用了一定的空间。

    */
    eFPartition* eFP_ptr = (eFPartition*)EFPHandler;
    unsigned long eeb = eFP_ptr->firstFree;

    // eeb非空
    if(eeb){
        eFP_ptr->firstFree = ((EEB*)eeb)->next_start;
    }
    return eeb;
}


unsigned long eFPartitionFree(unsigned long EFPHandler,unsigned long mbStart)
{
    //本函数需要实现！！！
    /*功能：释放一个空间
    1.mbstart将成为第一个空闲块，EFPHandler的firstFree属性也需要相应大的更新。
    2.同时我们也需要更新维护空闲内存块组成的链表。

    */
    eFPartition* eFP_ptr = (eFPartition*)EFPHandler;
    // 释放位置非法(不符合内存分配后的地址要求)
    if(mbStart < EFPHandler+eFPartition_size){
        return 0;
    }

    // 空闲链表为空
    if(eFP_ptr->firstFree ==0){
        eFP_ptr->firstFree = mbStart;
        ((EEB*)mbStart)->next_start = 0;
    }
    // 待释放块在所有空闲块之前
    else if(mbStart < eFP_ptr->firstFree){
        ((EEB*)mbStart)->next_start = eFP_ptr->firstFree;
        eFP_ptr->firstFree = mbStart;
    }
```

```
    // 通过双指针找到待释放块在空闲链表中相邻的前块
    else {
        unsigned long eeb_p,eeb_q;
        eeb_p = eFP_ptr->firstFree;
        eeb_q = eFP_ptr->firstFree;
        while(eeb_q < mbStart){
            eeb_p = eeb_q;
            eeb_q = ((EEB*)eeb_q)->next_start;
            if(eeb_q ==0){
                break;
            }
        }
        ((EEB*)eeb_p)->next_start = mbStart;
        ((EEB*)mbStart)->next_start = eeb_q;
    }
    return 0;

}
```

4. 可扩展的命令添加实现

> 通过定义命令的结构体，并在内存中维护一个命令的动态链表，并借用动态内存分配机制实现动态添加命令的功能

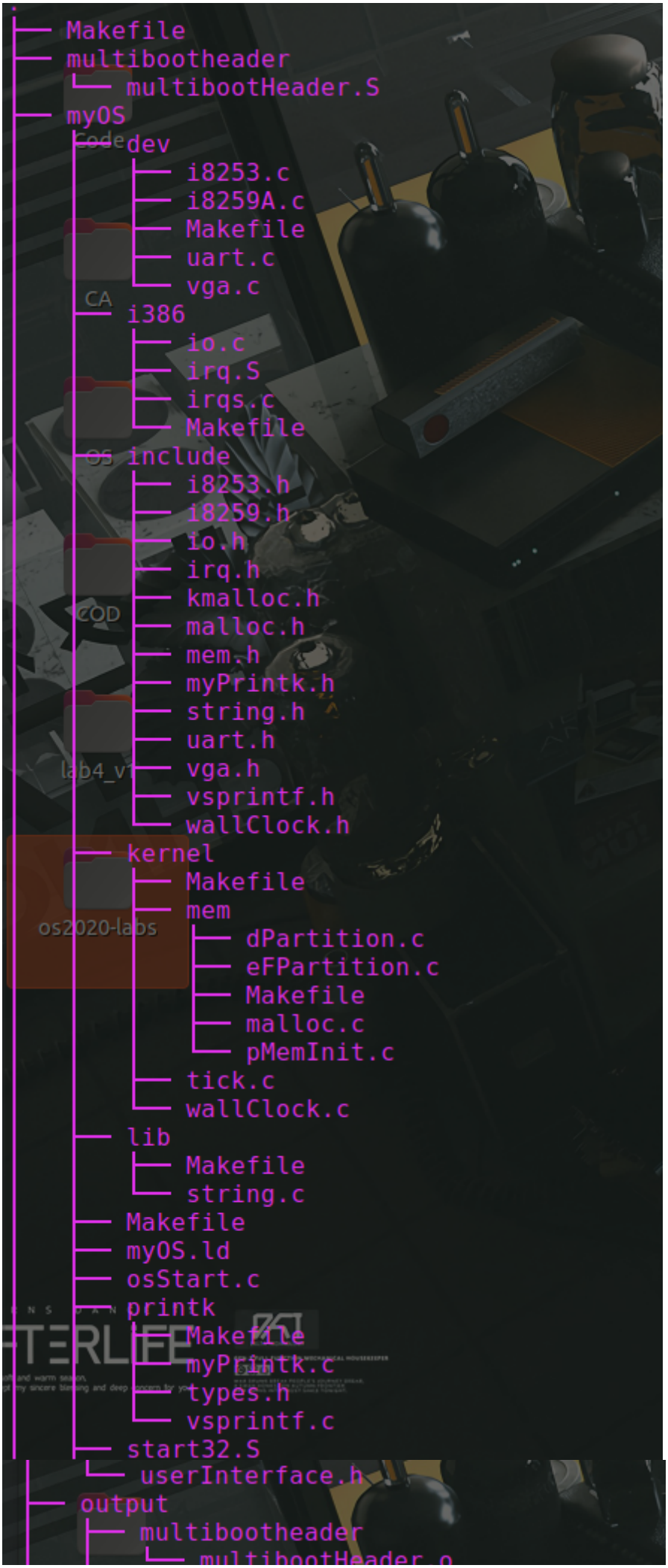
```
//shell.c --- malloc version
#include "../myOS/userInterface.h"
#define NULL (void*)0


//获取终端用户输入并回显
int getCmdline(unsigned char *buf, int limit){
    unsigned char *ptr = buf;
    int n = 0;
    while (n < limit) {
        *ptr = uart_get_char();
        if (*ptr == 0xd) {
            *ptr++ = '\n';
            *ptr = '\0';
            uart_put_char('\r');
            uart_put_char('\n');
            return n+2;
        }
        uart_put_char(*ptr);
        ptr++;
        n++;
    }
    return n;
}
//计算字符串的长度
int strlen(unsigned char* str){
    int i=0;
    while(str[i]){
        i++;
    }
    return i;
```

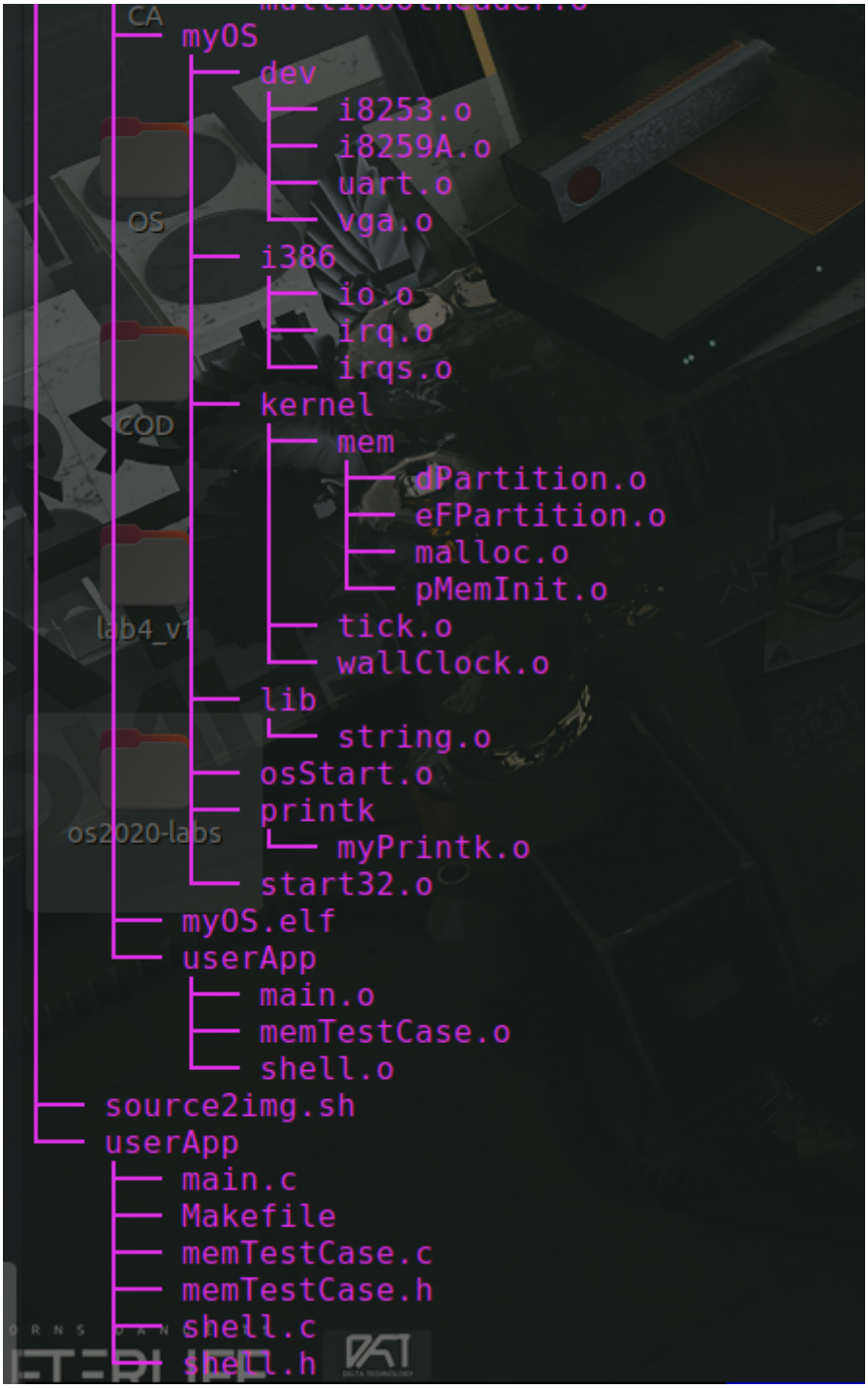
```
}
// 定义命令的结构体
typedef struct cmd {
    unsigned char cmd[20+1]; //TODO: dynamic
    int (*func)(int argc, unsigned char **argv);
    void (*help_func)(void);
    unsigned char description[100+1]; //TODO: dynamic?
    struct cmd * nextCmd;
} cmd;

#define cmd_size sizeof(struct cmd)

//在内存里维护命令的动态链表
struct cmd *ourCmds = NULL;


//依据命令的动态链表打印当前所有命令
int listCmds(int argc, unsigned char **argv){
    struct cmd *tmpCmd = ourCmds;
    myPrintf(0x7, "list all registered commands:\n");
    myPrintf(0x7, "command name: description\n");

    while (tmpCmd != NULL) {
        myPrintf(0x7,"% 12s: %s\n",tmpCmd->cmd, tmpCmd->description);
        tmpCmd = tmpCmd->nextCmd;
    }
    return 0;
}

// 向已有命令链表中动态添加一个命令
void addNewCmd( unsigned char *cmd,
        int (*func)(int argc, unsigned char **argv),
        void (*help_func)(void),
        unsigned char* description){
    //本函数需要实现！！！
    /*功能：增加命令
    1.使用malloc创建一个cmd的结构体，新增命令。
    2.同时还需要维护一个表头为ourCmds的链表。
    */

   struct cmd* new_cmd = (struct cmd*) malloc (cmd_size);
   new_cmd->func = func;
   new_cmd->help_func = help_func;

   strncpy(cmd,new_cmd->cmd,strlen(cmd));
   new_cmd->cmd[strlen(cmd)] = 0;
   strncpy(description,new_cmd->description,strlen(description));
   new_cmd->description[strlen(description)] = 0;
   new_cmd->nextCmd = 0;

   if(help_func){
       new_cmd->help_func = help_func;
   }

   //维护命令的动态链表
   if(ourCmds == NULL){
       ourCmds = new_cmd;
   }
```

```
   else {
       new_cmd->nextCmd = ourCmds->nextCmd;
       ourCmds->nextCmd = new_cmd;
   }
}

void help_help(void){
    myPrintf(0x7,"USAGE: help [cmd]\n\n");
}

int help(int argc, unsigned char **argv){
    int i;
    struct cmd *tmpCmd = ourCmds;
    if (argc==1) {
        help_help();
        return 0;
    }
    if (argc>2) return 1;

    while (tmpCmd != NULL) {
        if (strcmp(argv[1],tmpCmd->cmd)==0) {
            if (tmpCmd->help_func!=NULL)
                tmpCmd->help_func();
            else myPrintf(0x7,"%s\n",tmpCmd->description);
            break;
        }
        tmpCmd = tmpCmd->nextCmd;
    }
    return 0;
}

struct cmd *findCmd(unsigned char *cmd){
        struct cmd * tmpCmd = ourCmds;
    int found = 0;
        while (tmpCmd != NULL) {  //at lease 2 cmds
            if (strcmp(cmd,tmpCmd->cmd)==0){
            found=1;
            break;
        }
            tmpCmd = tmpCmd->nextCmd;
        }
    return found?tmpCmd:NULL;
}

int split2Words(unsigned char *cmdline, unsigned char **argv, int limit){
    unsigned char c, *ptr = cmdline;
    int argc=0;
    int inAWord=0;

    while ( c = *ptr ) {  // not '\0'
        if (argc >= limit) {
            myPrintf(0x7,"cmdline is tooooo long\n");
            break;
        }
        switch (c) {
            case ' ':  *ptr = '\0'; inAWord = 0; break; //skip white space
            case '\n': *ptr = '\0'; inAWord = 0; break; //end of cmdline?
            default:  //a word
```

```
            if (!inAWord) *(argv + argc++) = ptr;
            inAWord = 1;
            break;
        }
        ptr++;
    }
    return argc;
}


void initShell(void){
    addNewCmd("cmd\0",listCmds,NULL,"list all registered commands\0");
    addNewCmd("help\0",help,help_help,"help [cmd]\0");
    //TODO: may be we can add a new command exit or quit
    addNewCmd("exit\0",NULL,NULL,"Exit shell\0");
}

unsigned char cmdline[100];
void startShell(void){
    unsigned char *argv[10];  //max 10
    int argc;
    struct cmd *tmpCmd;
    dPartitionWalkByAddr(pMemHandler);
    char exit_or_not = 0;
    //实现有条件循环以提供退出功能
    while(!exit_or_not) {
        myPrintf(0x3,"Student >:");
        getCmdline(&cmdline[0],100);
        myPrintf(0x7,cmdline);
        argc = split2Words(cmdline,&argv[0],10);
        if (argc == 0) continue;
        tmpCmd = findCmd(argv[0]);
        if (tmpCmd) {
            if(! tmpCmd->func){
                exit_or_not = 1;
            }
            else {
                tmpCmd->func(argc, argv);
            }
        }
        else
            myPrintf(0x7,"UNKOWN command: %s\n",argv[0]);
    }
}
```

- 代码组织及其实现
  - 目录组织


.
├── Makefile
├── multibootheader
│   └── multibootHeader.S
├── myOS
│   ├── dev
│   │   ├── i8253.c
│   │   ├── i8259A.c
│   │   ├── Makefile
│   │   ├── uart.c
│   │   └── vga.c
│   ├── i386
│   │   ├── io.c
│   │   ├── irq.S
│   │   ├── irqs.c
│   │   └── Makefile
│   ├── include
│   │   ├── i8253.h
│   │   ├── i8259.h
│   │   ├── io.h
│   │   ├── irq.h
│   │   ├── kmalloc.h
│   │   ├── malloc.h
│   │   ├── mem.h
│   │   ├── myPrintk.h
│   │   ├── string.h
│   │   ├── uart.h
│   │   ├── vga.h
│   │   ├── vsprintf.h
│   │   └── wallClock.h
│   ├── kernel
│   │   ├── Makefile
│   │   ├── mem
│   │   │   ├── dPartition.c
│   │   │   ├── eFPartition.c
│   │   │   ├── Makefile
│   │   │   ├── malloc.c
│   │   │   └── pMemInit.c
│   │   ├── tick.c
│   │   └── wallClock.c
│   ├── lib
│   │   ├── Makefile
│   │   └── string.c
│   ├── Makefile
│   ├── myOS.ld
│   ├── osStart.c
│   ├── printk
│   │   ├── Makefile
│   │   ├── myPrintk.c
│   │   ├── types.h
│   │   └── vsprintf.c
│   ├── start32.S
│   └── userInterface.h
└── output
    └── multibootheader
        └── multibootHeader.o



代码按模块主要分为四部分：

1. multibootheader

> 此模块提供`multibootHeader`段的代码，使得`bootloader`成功将操作系统载入内存

2. myOS

> 此模块提供操作系统的代码，包括
> `start32.S` 初始化C程序的运行环境，设定`IDT`的汇编级程序
> `dev` 提供硬件层`UART`,`VGA`的使用封装接口,`i8253`,`i8259A`的初始化接口
> `i386` 提供基于`i386`架构的底层硬件`IO`的接口,中断处理程序
> `include` C程序的头文件
> `lib` C程序的库
> `printk` `myPrintk`,`vsprintf`函数的实现
> `kernel` 时钟显示模块的实现以及内存管理模块的实现
> `osStart.c` 初始化系统并进入用户程序

3. userApp

> 此模块存放用户程序，本次实验包含`main.c`,`startShell.c`,`memTestCase.c`，提供`shell`终端服务,以及内存管理测试

4. output

> 此模块存放含操作系统及用户程序在内的所有程序根据`Makefile`,`ld`文件编译链接后的生成的可执行文件，结构与`myOS`类似，不再赘述

- Makefile组织

```
.
├── MULTI_BOOT_HEADER
|   └── output/multibootheader/multibootHeader.o
└── OS_OBJS
    ├── MYOS_OBJS
    |   ├── output/myOS/start32.o
    |   ├── output/myOS/osStart.o
    |   ├── DEV_OBJS
    |   |   ├── output/myOS/dev/uart.o
    |   |   ├── output/myOS/dev/vga.o
    |   |   ├── output/myOS/dev/i8259A.o
    |   |   └── output/myOS/dev/i8253.o
    |   ├── I386_OBJS
    |   |   ├── output/myOS/i386/io.o
    |   |   ├── output/myOS/i386/irqs.o
    |   |   └── output/myOS/i386/irq.o
    |   ├── PRINTK_OBJS
    |   |   ├── output/myOS/printk/myPrintk.o
    |   |   └── output/myOS/printk/vsprintf.o
    |   ├── LIB_OBJS
    |   |   └── output/myOS/lib/string.o
    |   └── KERNEL_OBJS
    |       ├── output/myOS/kernel/tick.o
    |       ├── output/myOS/kernel/wallClock.o
    |       └── MEM_OBJS
    |           ├── output/myOS/kernel/mem/pMemInit.o
    |           ├── output/myOS/kernel/mem/dPartition.o
    |           ├── output/myOS/kernel/mem/eFPartition.o
    |           └── output/myOS/kernel/mem/malloc.o
    └── USER_APP_OBJS
        ├── output/userApp/main.o
        ├── output/userApp/shell.o
        └── output/userApp/memTestCase.o
```

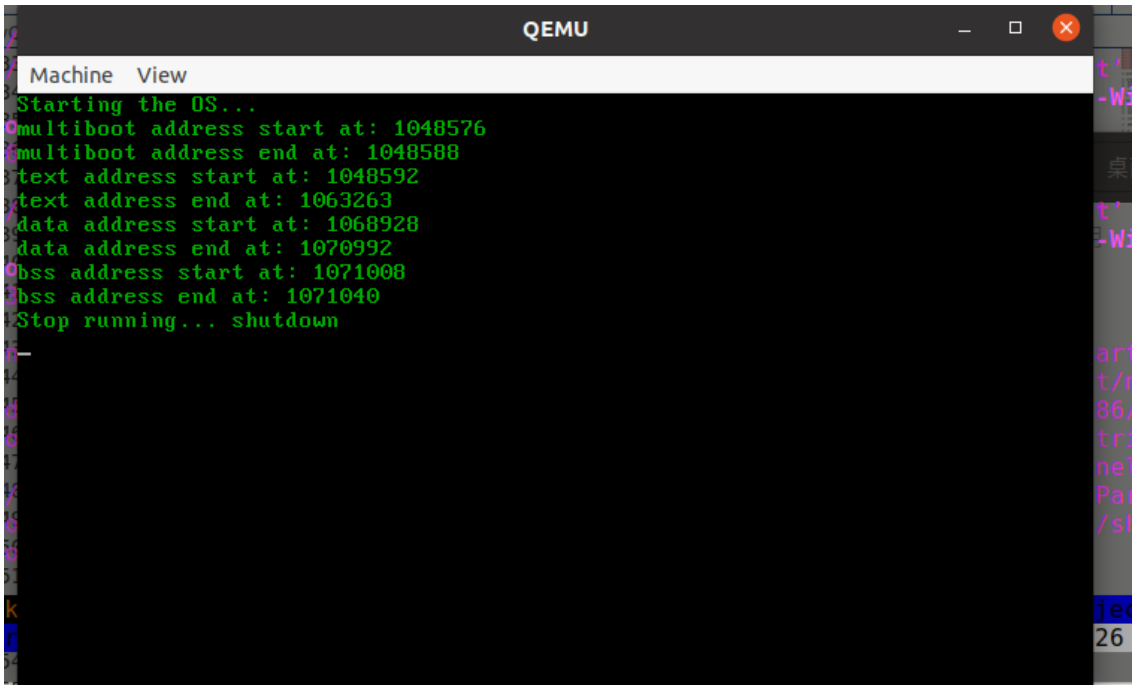
- 代码布局说明

  借助C程序可查看各段的起止位置

```
#include "vga.h"
#include "myPrintk.h"
extern unsigned long __multiboot_start;
extern unsigned long __multiboot_end;
extern unsigned long __text_start;
extern unsigned long __text_end;
extern unsigned long __data_start;
extern unsigned long __data_end;
extern unsigned long __bss_start;
extern unsigned long __bss_end;
/* 此文件无需修改 */

// 用户程序入口
void myMain(void);

void osStart(void) {
    clear_screen();
    myPrintk(0x2, "Starting the OS...\n");
    //myMain();
    myPrintk(0x2,"multiboot address start at: %d\n",(unsigned
long)&__multiboot_start);
    myPrintk(0x2,"multiboot address end at: %d\n",(unsigned
long)&__multiboot_end);
    myPrintk(0x2,"text address start at: %d\n",(unsigned long)&__text_start);
    myPrintk(0x2,"text address end at: %d\n",(unsigned long)&__text_end);
    myPrintk(0x2,"data address start at: %d\n",(unsigned long)&__data_start);
    myPrintk(0x2,"data address end at: %d\n",(unsigned long)&__data_end);

    myPrintk(0x2,"bss address start at: %d\n",(unsigned long)&__bss_start);
    myPrintk(0x2,"bss address end at: %d\n",(unsigned long)&__bss_end);

    myPrintk(0x2, "Stop running... shutdown\n");
    while(1);
}
```

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(start)

SECTIONS {
    . = 1M;
    .text : {
        __multiboot_start = .;
        *(.multiboot_header)
        __multiboot_end = .;
        . = ALIGN(8);
        __text_start = .;
        *(.text)
        __text_end = .;
    }

    . = ALIGN(16);
    __data_start = .;
    .data       : { *(.data*) }
    __data_end = .;

    . = ALIGN(16);
```

```
    .bss        :
    {
        __bss_start = .;
        _bss_start = .;
        *(.bss)
        __bss_end = .;
    }
    . = ALIGN(16);
    _end = .;
}
```



Section	Offset(Base = 0)
.multiboot_header	0x00100000~0x0010000c
.text	0x00100010~0x0010395f
.data	0x00104f80~0x00105790
.bss	0x001057a0~0x001057c0

【实验过程】

- 编译过程及运行过程说明

> 直接执行脚本文件`source2img.h`实现编译链接及运行

脚本文件及外层Makefile如下：

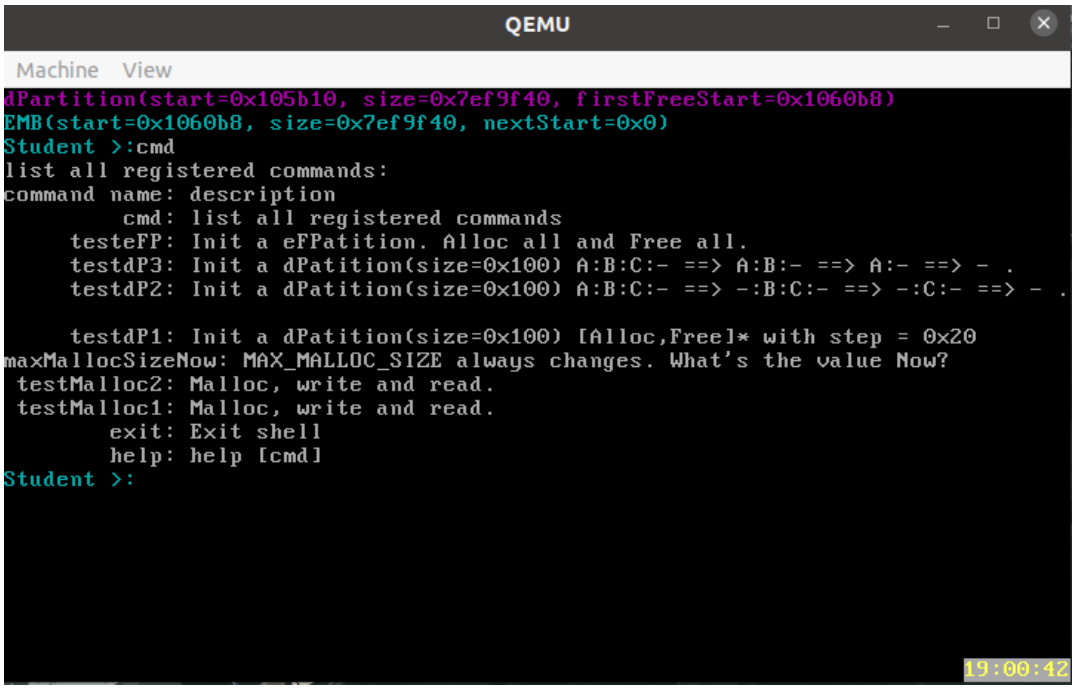
```
#!/bin/bash
make clean

make

if [ $? -ne 0 ]; then
    echo "make failed"
else
    echo "make succeed"
    qemu-system-i386 -kernel output/myOS.elf -serial stdio
fi
```

```
#SRC_RT=/home/xlanchen/workspace/OS_EX/multiboot_serials/3_shell_interrupt_ti
mer/
SRC_RT=$(shell pwd)

CROSS_COMPILE=
ASM_FLAGS= -m32 --pipe -Wall -fasm -g -O1 -fno-stack-protector
C_FLAGS =  -m32 -fno-stack-protector -fno-builtin-free -g

.PHONY: all
all: output/myOS.elf

MULTI_BOOT_HEADER=output/multibootheader/multibootHeader.o
include $(SRC_RT)/myOS/Makefile
include $(SRC_RT)/userApp/Makefile

OS_OBJS       = ${MYOS_OBJS} ${USER_APP_OBJS}

output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
    ${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${OS_OBJS} -o
output/myOS.elf

output/%.o : %.S
    @mkdir -p $(dir $@)
    @${CROSS_COMPILE}gcc ${ASM_FLAGS} -c -o $@ $<

output/%.o : %.c
    @mkdir -p $(dir $@)
    @${CROSS_COMPILE}gcc ${C_FLAGS} -c -o $@ $<

clean:
    rm -rf output
```

- 运行测试
  - shell终端

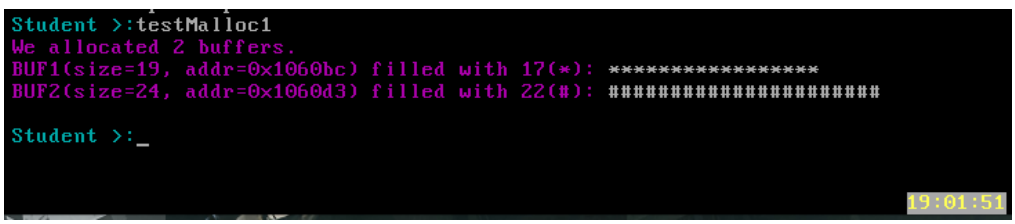
    初始界面

cmd打印当前所有命令



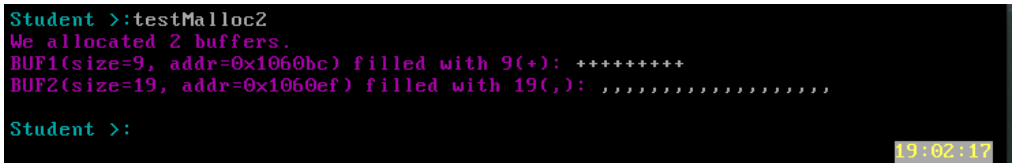
- 内存测试

1. testMalloc1



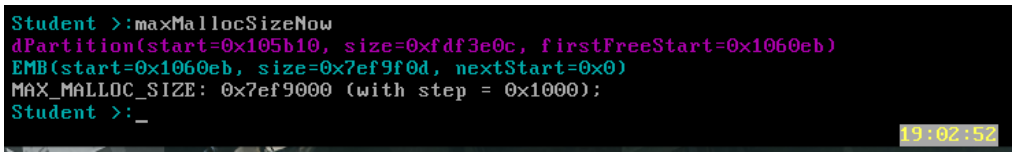
功能说明：测试动态分配
使用动态分配大小分别为19和24的空间，分别写入17个'*'和22个'#'并打印出字符串
由结果可知符合预期

2. testMalloc2



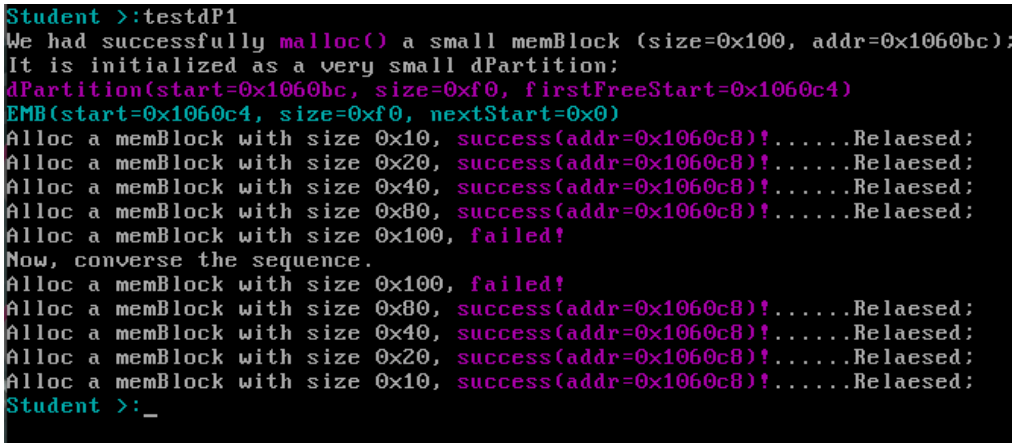
功能说明：测试动态分配
使用动态分配大小分别为11和21的空间，分别写入9个'+'和19个','并打印出字符串
由结果可知符合预期

3. maxMallocSizeNow



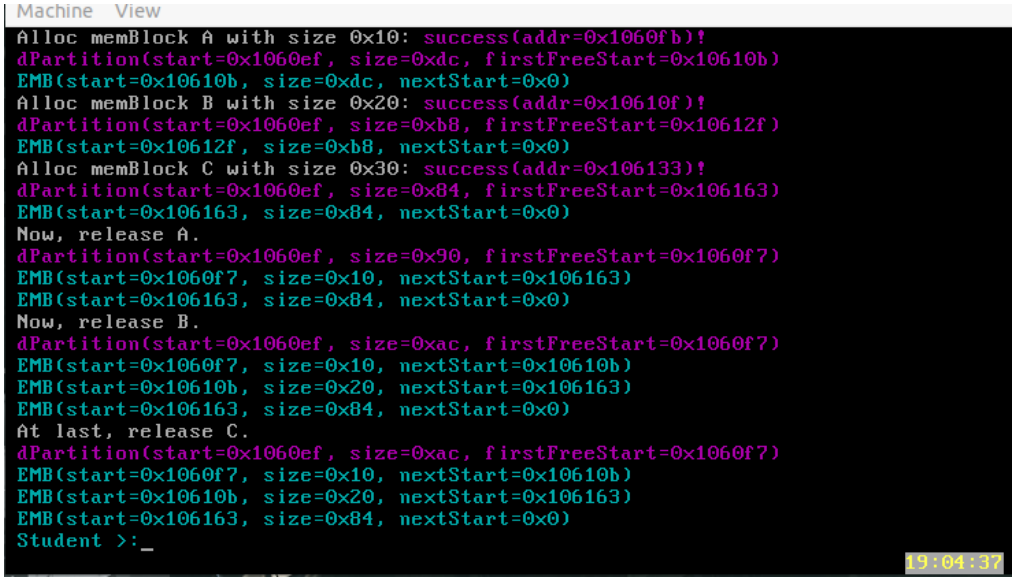
功能说明：测试动态分配
以0x1000为步长,从i=0x1000开始，逐步要求分配大小为i的空间直至失败，由shell初始界面可知内存最大为0x7ef9f40,故按步长0x1000最大分配空间为0x7ef9000
由结果可知符合预期

4. testdP1



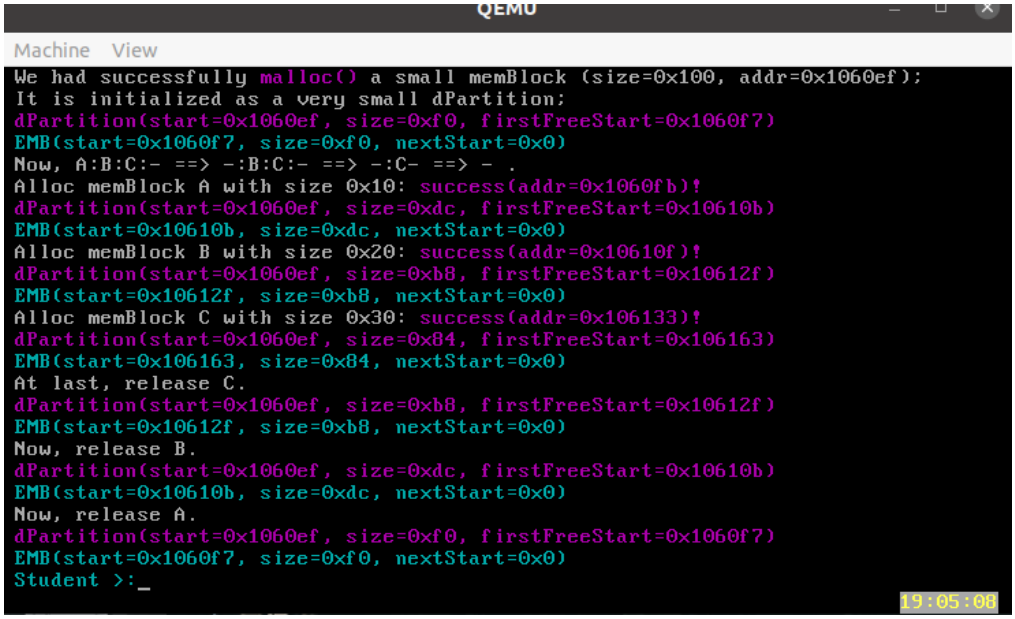
功能说明：测试动态分配
先分配一个大小为0x100的空间用作初始内存块并进行动态管理，之后分别对该内存进行分配并释放请求，大小依次为0x10,0x20,0x40,0x80,0x100，由于内存块需要维护一定大小的管理所需数据结构，在0x100大小的请求会失败，之后再对该内存进行分配并释放请求，大小依次为0x100,0x80,0x40,0x20,0x10
由结果可知符合预期

5. testdP2



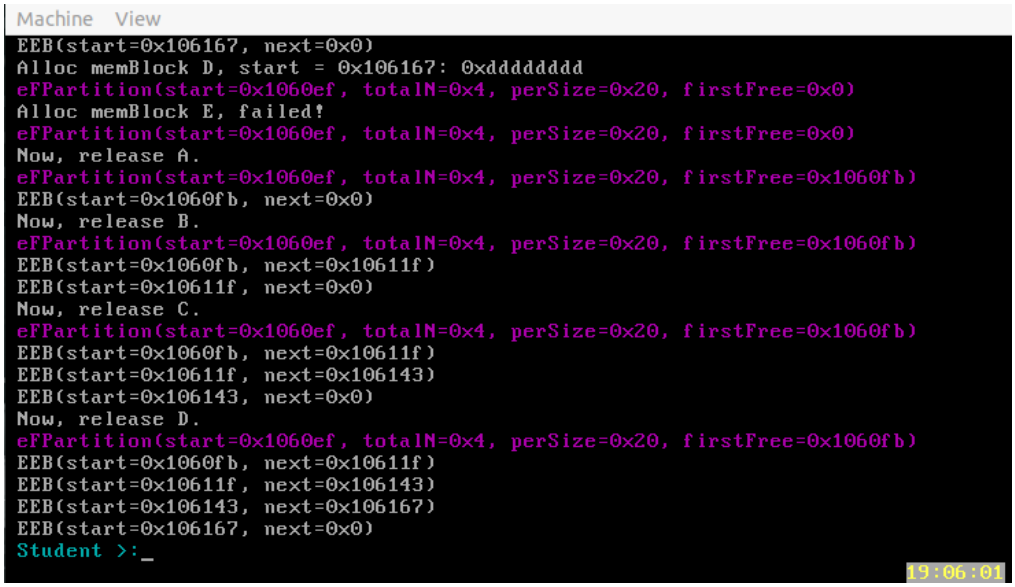
功能说明：测试动态分配
先分配一个大小为0x100的空间用作初始内存块并进行动态管理，之后分别对该内存进行连续分配请求，大小依次为0x10,0x20,0x30,最后发出连续释放请求，释放顺序与请求顺序一致，三次分配均成功，，分配前后dPartition结构应保持一致
由结果可知符合预期

6. testdP3



功能说明：测试动态分配
先分配一个大小为0x100的空间用作初始内存块并进行动态管理，之后分别对该内存进行连续分配请求，大小依次为0x10,0x20,0x30,最后发出连续释放请求，释放顺序与请求顺序相反，三次分配均成功，分配前后dPartition结构应保持一致
由结果可知符合预期

7. testeFP



> 功能说明：测试等大小分配
> 先申请一块空间(满足划分成4个32字节块外加eFPartition结构与EEB结构大小)用作初始内存块并进行等大小内存管理，之后分别对该内存进行连续分配请求，由于只有4块，在申请第五次时失败，最后连续4次释放
> 由结果可知符合预期

【问题与解决】

- 实现addNewCmd时，将命令的cmd域与description域拷贝后，命令链表里的相应域会有乱码

  > 通过`strncpy`指定复制的长度，最后手动在最后末尾置`0`

- 在动态分区管理释放块后对空闲链表的维护不当(没有考虑到所有可能的情况)

  > 通过测试发现错误完善了各种情况