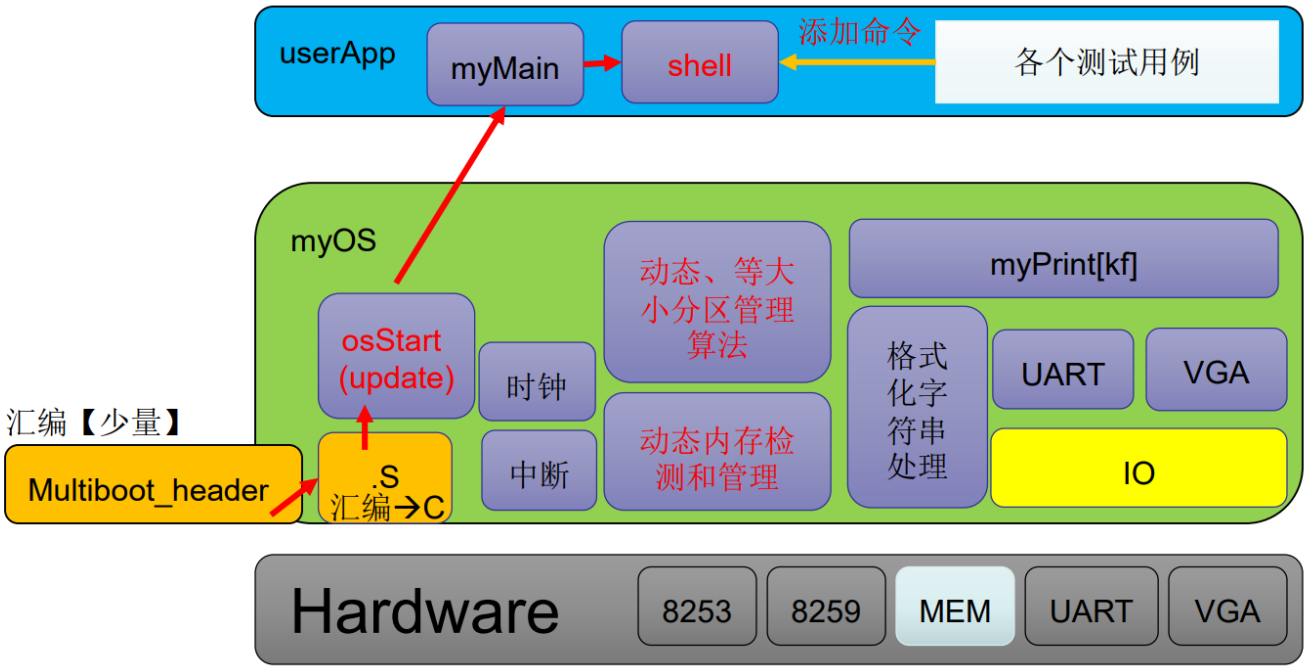
第四次实验报告

学号：PB20000024

姓名：陈奕衡

一、软件框图



相比上一次实验，本次实验新增了内存管理模块，主要包括内存分配算法以及检测。成功在用户模块实现了增添新指令的函数并且通过了内存分配的测试样例。

另外，本次实验使用了助教所给的框架而非自己的框架，因此重点叙述对于相应功能的实现。

二、主流程说明

主流程与实验三基本相同，首先是qemu虚拟机启动并运行Multiboot_header文件，通过启动头进入myOS操作系统（访问start.S），在操作系统中通过start32.S文件运行osStart.c文件，在文件中进行中断控制器的初始化以及允许时钟中断。最后访问Mymain函数进行内存分配测试以及添加新命令的测试。如下所示：

```
void osStart(void){
    pressAnyKeyToStart(); // prepare for uart device
    init8259A();
    init8253();
```

```
    enable_interrupt();

    clear_screen();

    pMemInit();  //after this, we can use kmalloc/kfree and malloc/free

    {
        unsigned long tmp = dPartitionAlloc(pMemHandler,100);
        dPartitionWalkByAddr(pMemHandler);
        dPartitionFree(pMemHandler,tmp);
        dPartitionWalkByAddr(pMemHandler);
    }

    // finished kernel init
    // NOW, run userApp
    myPrintk(0x2,"START RUNNING......\n");
    myMain();
    myPrintk(0x2, "STOP RUNNING......ShutDown\n");
    while(1);
}
```

三、主要功能模块及其相应源代码

通过检测了解系统中的内存配置情况

- OS通过检测了解系统中的内存配置情况

- 本实验采用简化版检测
  - 假设从1M开始（1M以内的不考虑【可选】）
  - 假设只有1块连续的内存空间（多个非连续的【可选】）
  - 通过检测得到这个内存块的大小
  - 检测算法 void memTest(unsigned long start, unsigned long grainSize)
    - 从start开始，以grainSize为步长，进行内存检测
    - 检测方法：
      1）读出grain的头2个字节
      2）覆盖写入0xAA55，再读出并检查是否是0xAA55，若不是则检测结束；
      3）覆盖写入0x55AA，再读出并检查是否是0x55AA，若不是则检测结束；
      4）写回原来的值
      5）对grain的尾2个字节，重复2-4
      6）步进到下一个grain，重复1-5，直到检测结束

这里根据PPT中的检测算法，得出相应的检测函数：

```
void memTest(unsigned long start, unsigned long grainSize){
    if(start < 0x100000 || grainSize < 0x10){
        pMemSize = 0;
        pMemStart = 0;
        return;
    }
    unsigned short *p = (unsigned short *)start;
    pMemStart = start;
    while(1){
        unsigned short *q = p + grainSize - 1;
        unsigned short temp_p = *p, temp_q = *q;
        *p = 0xAA55;
        if (*p != 0xAA55) break;
        *p = 0x55AA;
        if (*p != 0x55AA) break;
        *q = 0xAA55;
        if (*q != 0xAA55) break;
        *q = 0x55AA;
        if (*q != 0x55AA) break;
        *p = temp_p; *q = temp_q;
        pMemSize += grainSize;
        p = q + 1;
    }
    myPrintk(0x7,"MemStart: %x  \n", pMemStart);
    myPrintk(0x7,"MemSize:  %x  \n", pMemSize);
}
```

检测完内存之后，需要进行分配，这里在分配时将内存均分，分别给系统和用户进行使用，并且分配了不同的句柄以进行访问

```
extern unsigned long _end;
void pMemInit(void){
    unsigned long _end_addr = (unsigned long) &_end;
    memTest(0x100000,0x1000);
    myPrintk(0x7,"_end:  %x  \n", _end_addr);
    if (pMemStart <= _end_addr) {
        pMemSize -= _end_addr - pMemStart;
        pMemStart = _end_addr;
    }

    kMemHandler = dPartitionInit(pMemStart, pMemSize / 2);
    uMemHandler = dPartitionInit(pMemStart + pMemSize / 2, pMemSize -
pMemSize / 2);
}
```

动态内存分配

首先是每个内存块的数据结构定义：

```
//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

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

使用简单的单向链表进行整个空闲链表的构建

之后是对于内存块的初始化以及输出检测：

```
unsigned long dPartitionInit(unsigned long start, unsigned long totalSize){
    //本函数需要实现！！！

    dPartition *handler = (dPartition *)start;
    EMB *emb = (EMB *)(start + dPartition_size);

    handler->size = totalSize - sizeof(dPartition);
    handler->firstFreeStart = (unsigned long)emb;

    emb->size = handler->size - sizeof(EMB);
    emb->nextStart = 0;

    return start;
}

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

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

void dPartitionWalkByAddr(unsigned long dp){
    showdPartition((dPartition *)dp);
    dPartition *handler = (dPartition *)dp;
    EMB *emb = (EMB *)(handler->firstFreeStart);
    while (emb != 0){
        showEMB(emb);
        emb = (EMB *)emb->nextStart;
    }
```

```
    return;
}
```

此处操作就是以handler为头指针建立一个空闲链表，并且维护相应的空间大小。至于输出，就是简单的遍历整个链表。

之后是空间分配函数，使用的是firstfit算法分配空间：

```
unsigned long dPartitionAllocFirstFit(unsigned long dp, unsigned long size)
{
    dPartition *handler = (dPartition *)dp;
    if (handler->firstFreeStart == 0) {//no free space
        return 0;
    }

    EMB *emb = (EMB *)(handler->firstFreeStart);
    EMB *prev = 0;

    while (emb != 0){//find the first free space
        if (emb->size >= size) break;
        prev = emb;
        emb = (EMB *)(emb->nextStart);
    }
    if(emb == 0) return 0;

    if (emb->size < (size + EMB_size)){//the block cannot be splited again
        if(emb == (EMB *)(handler->firstFreeStart)){
            handler->firstFreeStart = emb->nextStart;
        } else{
            prev->nextStart = emb->nextStart;
        }
        emb->nextStart = 0;
    } else{ //the block has enough space to be splited
        unsigned long new_emb = (unsigned long)(emb + 1) + size;
        EMB *new = (EMB *)new_emb;
        new->nextStart = emb->nextStart;
        new->size = emb->size - size - EMB_size;
        emb->size = size;
        if (emb == (EMB *)(handler->firstFreeStart)){//head of the linklist
            handler->firstFreeStart = new_emb;
        } else{
            prev->nextStart = new_emb;
        }
    }

    return (unsigned long)(emb + 1);
}
```

- 首先遍历链表找到满足容量的空闲块
- 之后分以下两种情况分配块内空间：

    - 如果块内空间已经不足以申请一个新的链表头EMB，就直接将块分配出去
    - 相反的话就需要进行对于内存块的分割，分配出去所需内存后将剩余内存构建成一个新的空闲块new_emb
- 构建过程是从低地址到高地址，即先将低地址的内存分配出去

分配结束后还需要相应的回收函数，如下：

```
unsigned long dPartitionFreeFirstFit(unsigned long dp, unsigned long start)
{
    if (start < dp + sizeof(dPartition) ||
        start >= dp + sizeof(dPartition) + ((dPartition *)dp)->size)
{//invalid value of start
        return 0;
    }

    dPartition *handler = (dPartition *)dp;
    if (handler->firstFreeStart == 0){//the linklist is empty
        handler->firstFreeStart = (unsigned long)((EMB *)start - 1);
        ((EMB *)start - 1)->nextStart = 0;
        return 1;
    }

    EMB *emb = (EMB *)(handler->firstFreeStart);
    EMB *prev = 0;
    EMB *freeSpace = (EMB *)start - 1;

    while (emb != 0){
        if (emb > freeSpace){//the next block of freespace
            break;
        }
        if (emb == freeSpace){//wrong value of start
            return 1;
        }
        prev = emb;
        emb = (EMB *)emb->nextStart;
    }

    if (emb == 0){//tail of the linklist
        prev->nextStart = (unsigned long)freeSpace;
        freeSpace->nextStart = 0;
    } else{
        freeSpace->nextStart = (unsigned long)emb;
        if (emb == (EMB *)handler->firstFreeStart){//head of the linklist
            handler->firstFreeStart = (unsigned long)freeSpace;
        } else{
            prev->nextStart = (unsigned long)freeSpace;
        }
    }

    emb = (EMB *)(handler->firstFreeStart);
    while (emb != 0){//detect and combine two blocks
        if ((unsigned long)(emb + 1) + emb->size == emb->nextStart){
```

```
                emb->size += ((EMB *)emb->nextStart)->size + EMB_size;
                emb->nextStart = ((EMB *)emb->nextStart)->nextStart;
                ((EMB *)emb->nextStart)->nextStart = 0;
                continue;
            }
            emb = (EMB *)emb->nextStart;
        }

        return 1;
    }
```

- 回收空间时可能遇到以下三种情况：
  - 空闲链表为空，将回收块当成第一个空闲块加入
  - 回收块在空闲链表尾端，直接加入队尾
  - 回收块在中间位置，这时需要插入链表中
- 在将回收块插入链表后再次遍历链表，检测是否存在可合并块并进行合并操作

至此整个动态分配内存模块完成，即`malloc`和`free`函数已经实现。

```
unsigned long malloc(unsigned long size) {
    return dPartitionAlloc(uMemHandler, size);
}

unsigned long free(unsigned long start) {
    return dPartitionFree(uMemHandler, start);
}
```

静态内存分配

相比于动态内存分配，静态就要相对简单一些，只需维护定长定数量的内存块即可

首先是每个内存块的数据结构定义：

```
// 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

//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
```

相比于动态内存分配，静态分配只需将内存大小存在句柄头的位置，句柄头还需记录一共分配的内存块的数量

之后是对于内存块的初始化以及输出检测：

```
unsigned long eFPartitionTotalSize(unsigned long perSize, unsigned long n){
    unsigned long actual_size = (perSize / 8 + 1) * 8 + EEB_size;
    return actual_size * n + eFPartition_size;
}

unsigned long eFPartitionInit(unsigned long start, unsigned long perSize,
unsigned long n){
    eFPartition *handler = (eFPartition *)start;

    unsigned long actual_size = (perSize / 8 + 1) * 8;

    handler->totalN = n;
    handler->perSize = actual_size;
    handler->firstFree = start + eFPartition_size;

    unsigned long eeb = handler->firstFree;

    for (int i = 0; i < n; i++) {
        ((EEB *)eeb)->next_start = eeb + actual_size + EEB_size;
        eeb += actual_size + EEB_size;
    }
    ((EEB *)(eeb - actual_size - EEB_size))->next_start = 0;

    return start;
}

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

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){
    eFPartition *efp = (eFPartition *)efpHandler;
    showeFPartition(efp);
    EEB *eeb = (EEB *)efp->firstFree;
    while(eeb != 0) {
        showEEB(eeb);
        eeb = (EEB *)eeb->next_start;
    }
}
```

- 与动态分配不同的两点在于：
  - 静态分配时需要计算每块内存以及考虑对齐
  - 静态分配初始化的时候有`totalN`个等大小内存块，而动态分配在初始化完成后只有一个大内存块

之后同样是空间分配函数，实现如下：

```
unsigned long eFPartitionAlloc(unsigned long EFPHandler){
    eFPartition *efp = (eFPartition *)EFPHandler;
    if (efp->firstFree == 0){
        return 0;
    }

    EEB *eeb = (EEB *)efp->firstFree;
    efp->firstFree = eeb->next_start;

    return (unsigned long)eeb + EEB_size;
}
```

直接将链表中第一个空闲块分配出去即可

之后是回收函数，实现如下：

```
unsigned long eFPartitionFree(unsigned long EFPHandler,unsigned long
mbStart){
    eFPartition *efp = (eFPartition *)EFPHandler;
    EEB *eeb = (EEB *)(mbStart) - 1;

    eeb->next_start = efp->firstFree;
    efp->firstFree = (unsigned long)eeb;

    return 0;
}
```

回收时直接将相应块添加到链表头即可

####Shell中增加新指令的实现

同样首先展示每个指令结构体中所含元素：

```
typedef struct cmd {
    unsigned char *name; //TODO: dynamic
    int (*func)(int argc, unsigned char **argv);
    void (*help_func)(void);
    unsigned char *description; //TODO: dynamic?
    struct cmd *nextCmd;
}cmd;

struct cmd *ourCmds = NULL;

void listInit(void){
    ourCmds = (cmd *)malloc(sizeof(cmd));
    ourCmds->nextCmd = NULL;
}
```

增添新指令同样是维护一个指令链表，实现如下：

```
#include "../myOS/include/string.h"

void addNewCmd( unsigned char *name,
        int (*func)(int argc, unsigned char **argv),
        void (*help_func)(void),
        unsigned char* description){
    cmd* new_cmd = (cmd*) malloc(sizeof(cmd));
    new_cmd->name = (char*) malloc((strlen(name) + 1) * sizeof(char));
    strcpy(name, new_cmd->name);
    new_cmd->func = func;
    new_cmd->help_func = help_func;
    new_cmd->description = (char*) malloc((strlen(description) + 1) *
sizeof(char));
    strcpy(description, new_cmd->description);

    new_cmd->nextCmd = ourCmds->nextCmd;
    ourCmds->nextCmd = new_cmd;

    return;
}
```

这里的name和description实现了动态分配，并且使用了string.c中的相应函数进行字符串操作。

最后是对exit指令的添加：

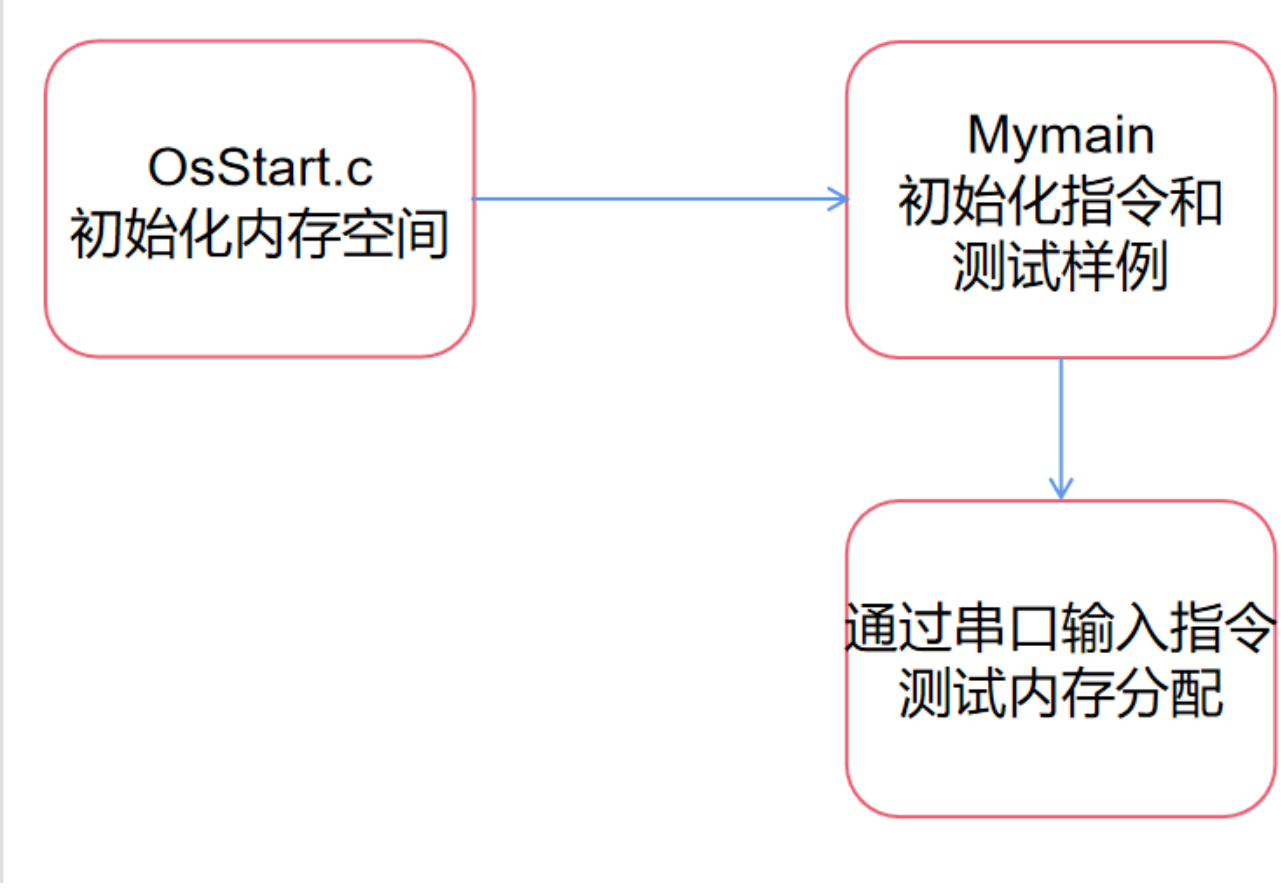
```
void startShell(void){
    unsigned char *argv[10];  //max 10
    int argc;
    struct cmd *tmpCmd;

    while(1) {
        myPrintf(0x3,"Student >:");
        getCmdline(&cmdline[0],100);
        myPrintf(0x7,cmdline);

        argc = split2Words(cmdline,&argv[0],10);
        if (argc == 0) continue;
        if (strcmp(argv[0], "exit")){
            return;
        }

        tmpCmd = findCmd(argv[0]);
        if (tmpCmd)
            tmpCmd->func(argc, argv);
        else
            myPrintf(0x7,"UNKOWN command: %s\n",argv[0]);
```

```
    }
}

int func_exit(int argc, unsigned char **argv){
    return 0;
}

void initShell(void){
    listInit();
    addNewCmd("cmd\0",listCmds,NULL,"list all registered commands\0");
    addNewCmd("help\0",help,help_help,"help [cmd]\0");
    addNewCmd("exit\0",func_exit,NULL,"exit the shell\0");
}
```

直接在读取完成后进行检测即可。

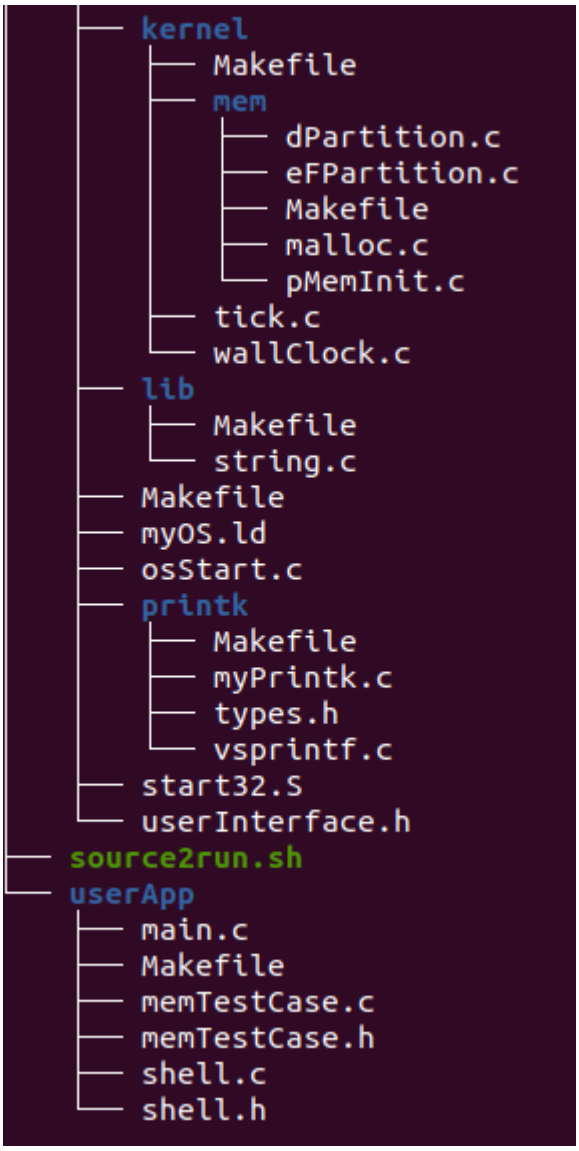
至此，主模块描述完毕，下面为主模块流程图：

- 内存管理模块



四、组织说明

本实验组织架构如下(发生改动的主要架构)：



Makefile组织架构如下：

```
MULTI_BOOT_HEADER = output/multibootheader/multibootHeader.o
MYOS_OBJS = output/myOS/start32.o output/myOS/osStart.o\
        I386_OBJS = output/myOS/i386/io.o \
                    output/myOS/i386/irq.o \
                    output/myOS/i386/irqs.o
        DEV_OBJS = output/myOS/dev/uart.o \
                    output/myOS/dev/vga.o \
                    output/myOS/dev/i8253.o \
                    output/myOS/dev/i8259A.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
        LIB_OBJS = output/myOS/lib/string.o
        PRINTK_OBJS = output/myOS/printk/myPrintk.o
USER_APP_OBJS =  output/userApp/main.o \
                 output/userApp/shell.o \
                 output/userApp/memTestCase.o
```

五、代码布局说明

代码空间分布如下（与上次试验相同）：

```
SECTIONS {
    . = 1M;
    .text : {
        *(.multiboot_header)
        . = ALIGN(8);
        *(.text)
    }

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

    . = ALIGN(16);
    .bss        :
    {
        __bss_start = .;
        _bss_start = .;
        *(.bss)
        __bss_end = .;
    }
    . = ALIGN(16);
    _end = .;
    . = ALIGN(512);
}
```

其中.text代码段在1m位置进行存储，以8字节形式对齐；.data段、.bss段以16字节形式对齐；最后的_end以及后续代码以512字节形式对齐。

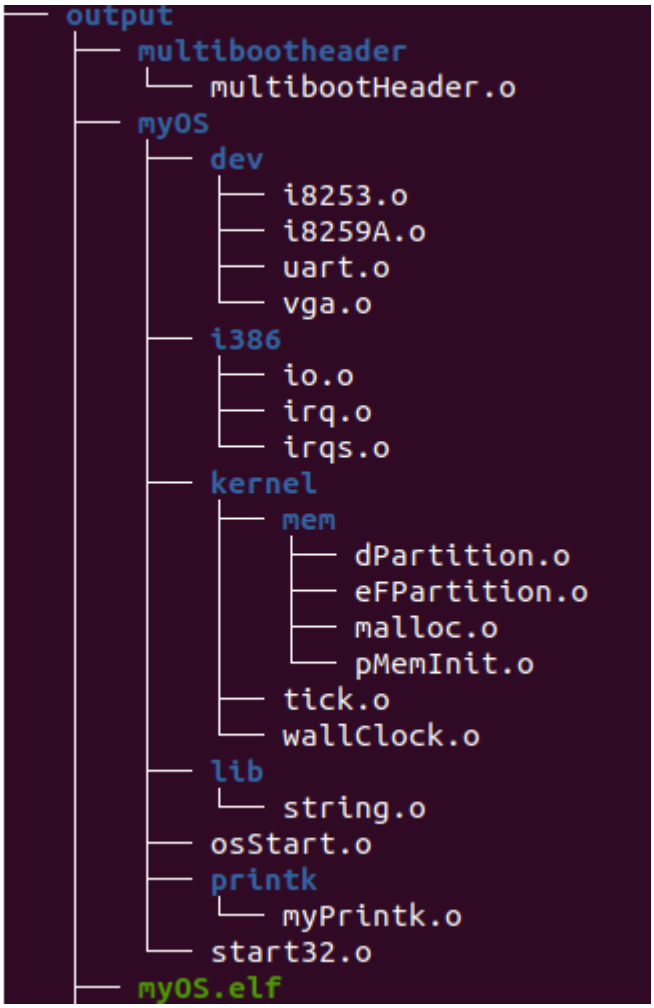
六、编译过程

编译所用脚本文件如下，在终端输入`./source2run.sh`便可一键编译加重定向串口（与上次试验相同）：

```
#!/bin/bash
make clean
make

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

```
cyh@cyh-ROG-Strix-G732LWS-G732LWS:~/os/lab4_v1——1$ ./source2run.sh
rm -rf output
ld -n -T myOS/myOS.ld output/multibootheader/multibootHeader.o output/myOS/start
32.o output/myOS/osStart.o output/myOS/dev/uart.o output/myOS/dev/vga.o output/m
yOS/dev/i8253.o output/myOS/dev/i8259A.o output/myOS/i386/io.o output/myOS/i386/
irq.o output/myOS/i386/irqs.o output/myOS/printk/myPrintk.o output/myOS/lib/stri
ng.o output/myOS/kernel/tick.o output/myOS/kernel/wallClock.o output/myOS/kernel
/mem/pMemInit.o output/myOS/kernel/mem/dPartition.o output/myOS/kernel/mem/eFPar
tition.o output/myOS/kernel/mem/malloc.o output/userApp/main.o output/userApp/sh
ell.o output/userApp/memTestCase.o -o output/myOS.elf
make succeed
cyh@cyh-ROG-Strix-G732LWS-G732LWS:~/os/lab4_v1——1$ char device redirected to /de
v/pts/1 (label serial0)
```

编译完成之后，还需使用screen命令重定向串口至伪终端才能进行相应shell的交互操作。编译过程中由专属的elf文件对相应的Makefile进行连接并最后统一输出至output文件夹中。



上图为编译过后的部分.o文件结构。

实验结果

运行后的实验结果如下：

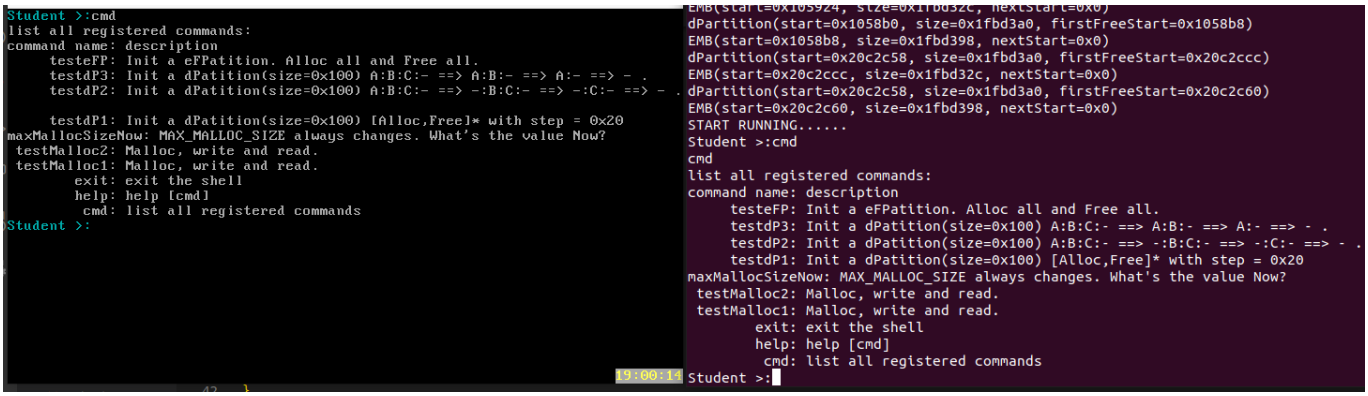
首先是对于初始分配的检查：

```
    {
        unsigned long tmp = dPartitionAlloc(kMemHandler,100);
        dPartitionWalkByAddr(kMemHandler);
        dPartitionFree(kMemHandler,tmp);
        dPartitionWalkByAddr(kMemHandler);
    }

    {
        unsigned long tmp = dPartitionAlloc(uMemHandler,100);
        dPartitionWalkByAddr(uMemHandler);
        dPartitionFree(uMemHandler,tmp);
        dPartitionWalkByAddr(uMemHandler);
    }
```

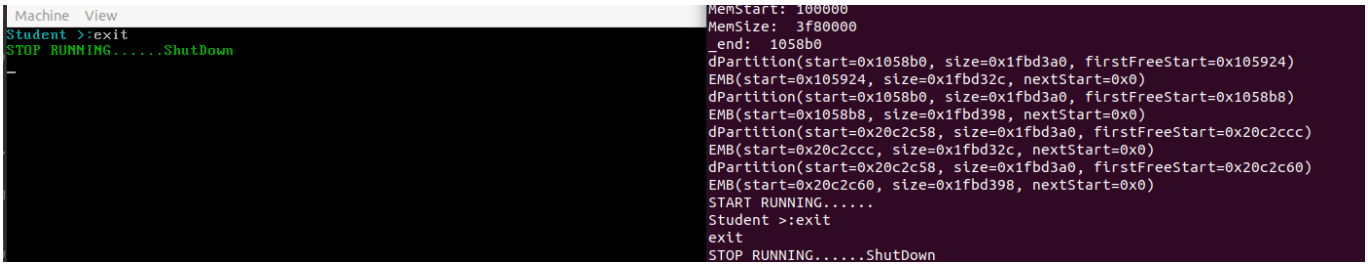
以上是初始化时对于`kmalloc`和`malloc`的测试样例

```
MemStart: 100000
MemSize:  3f80000
_end:  105810
dPartition(start=0x105810, size=0x1fbd3f0, firstFreeStart=0x105884)
EMB(start=0x105884, size=0x1fbd37c, nextStart=0x0)
dPartition(start=0x105810, size=0x1fbd3f0, firstFreeStart=0x105818)
EMB(start=0x105818, size=0x1fbd3e8, nextStart=0x0)
dPartition(start=0x20c2c08, size=0x1fbd3f0, firstFreeStart=0x20c2c7c)
EMB(start=0x20c2c7c, size=0x1fbd37c, nextStart=0x0)
dPartition(start=0x20c2c08, size=0x1fbd3f0, firstFreeStart=0x20c2c10)
EMB(start=0x20c2c10, size=0x1fbd3e8, nextStart=0x0)
START RUNNING......
```

可以清楚地看到总内存容量以及对用户空间和系统空间的不同分配，这里使用两个句柄实现，之后是测试样例：



首先是对于指令是否成功加载的测试，可以看到cmd指令正常运行，说明指令链表创建成功



发现exit指令执行成功，继续进行其他样例测试：

可以看到动态分配的第一个样例通过，没有未释放的空间，且由于从低地址到高地址存储，每次始存的起始地址相同，结果显示对于内存的分配是没有问题的，到0x100便分配失败

```
dPartition(start=0x20c3019, size=0xf8, firstFreeStart=0x20c3021)
EMB(start=0x20c3021, size=0xf0, nextStart=0x0)
Now, A:B:C:- ==> -:B:C:- ==> -:C- ==> - .
Alloc memBlock A with size 0x10: success(addr=0x20c3029)!
dPartition(start=0x20c3019, size=0xf8, firstFreeStart=0x20c3039)
EMB(start=0x20c3039, size=0xd8, nextStart=0x0)
Alloc memBlock B with size 0x20: success(addr=0x20c3041)!
dPartition(start=0x20c3019, size=0xf8, firstFreeStart=0x20c3061)
EMB(start=0x20c3061, size=0xb0, nextStart=0x0)
Alloc memBlock C with size 0x30: success(addr=0x20c3069)!
dPartition(start=0x20c3019, size=0xf8, firstFreeStart=0x20c3099)
EMB(start=0x20c3099, size=0x78, nextStart=0x0)
Now, release A.
dPartition(start=0x20c3019, size=0xf8, firstFreeStart=0x20c3021)
EMB(start=0x20c3021, size=0x10, nextStart=0x20c3099)
EMB(start=0x20c3099, size=0x78, nextStart=0x0)
Now, release B.
dPartition(start=0x20c3019, size=0xf8, firstFreeStart=0x20c3021)
EMB(start=0x20c3021, size=0x38, nextStart=0x20c3099)
EMB(start=0x20c3099, size=0x78, nextStart=0x0)
At last, release C.
dPartition(start=0x20c3019, size=0xf8, firstFreeStart=0x20c3021)
EMB(start=0x20c3021, size=0xf0, nextStart=0x0)
```

同样能够观察到空间的成功分配与释放，这里结果显示块的合并是没有问题的

```
We had successfully malloc() a small memBlock (size=0x100, addr=0x20c3019);
It is initialized as a very small dPartition:
dPartition(start=0x20c3019, size=0xf8, firstFreeStart=0x20c3021)
EMB(start=0x20c3021, size=0xf0, nextStart=0x0)
Now, A:B:C:- ==> -:B:C:- ==> -:C- ==> - .
Alloc memBlock A with size 0x10: success(addr=0x20c3029)!
dPartition(start=0x20c3019, size=0xf8, firstFreeStart=0x20c3039)
EMB(start=0x20c3039, size=0xd8, nextStart=0x0)
Alloc memBlock B with size 0x20: success(addr=0x20c3041)!
dPartition(start=0x20c3019, size=0xf8, firstFreeStart=0x20c3061)
EMB(start=0x20c3061, size=0xb0, nextStart=0x0)
Alloc memBlock C with size 0x30: success(addr=0x20c3069)!
dPartition(start=0x20c3019, size=0xf8, firstFreeStart=0x20c3099)
EMB(start=0x20c3099, size=0x78, nextStart=0x0)
At last, release C.
dPartition(start=0x20c3019, size=0xf8, firstFreeStart=0x20c3061)
EMB(start=0x20c3061, size=0xb0, nextStart=0x0)
Now, release B.
dPartition(start=0x20c3019, size=0xf8, firstFreeStart=0x20c3039)
EMB(start=0x20c3039, size=0xd8, nextStart=0x0)
Now, release A.
dPartition(start=0x20c3019, size=0xf8, firstFreeStart=0x20c3021)
EMB(start=0x20c3021, size=0xf0, nextStart=0x0)
Student >:_
```

实验结果同样成功，至此动态分配测试结束，之后测试静态分配：

```
EEB(start=0x20c3091, next=0x0)
Alloc memBlock D, start = 0x20c3095: 0xdddddddd
eFPartition(start=0x20c3019, totalN=0x4, perSize=0x20, firstFree=0x0)
Alloc memBlock E, failed!
eFPartition(start=0x20c3019, totalN=0x4, perSize=0x20, firstFree=0x0)
Now, release A.
eFPartition(start=0x20c3019, totalN=0x4, perSize=0x20, firstFree=0x20c3025)
EEB(start=0x20c3025, next=0x0)
Now, release B.
eFPartition(start=0x20c3019, totalN=0x4, perSize=0x20, firstFree=0x20c3049)
EEB(start=0x20c3049, next=0x20c3025)
EEB(start=0x20c3025, next=0x0)
Now, release C.
eFPartition(start=0x20c3019, totalN=0x4, perSize=0x20, firstFree=0x20c306d)
EEB(start=0x20c306d, next=0x20c3049)
EEB(start=0x20c3049, next=0x20c3025)
EEB(start=0x20c3025, next=0x0)
Now, release D.
eFPartition(start=0x20c3019, totalN=0x4, perSize=0x20, firstFree=0x20c3091)
EEB(start=0x20c3091, next=0x20c306d)
EEB(start=0x20c306d, next=0x20c3049)
EEB(start=0x20c3049, next=0x20c3025)
EEB(start=0x20c3025, next=0x0)
```

这里分配了四个内存块进行测试，可以看到E内存块分配失败因为没有空闲块了，而释放后也一共只有四个大小相同内存块，测试成功。

```
Student >:testMalloc1
testMalloc1
We allocated 2 buffers.
BUF1(size=19, addr=0x20c30bd) filled with 17(*): ****************
BUF2(size=24, addr=0x20c30d8) filled with 22(#): ######################

Student >:maxMallocSizeNow
maxMallocSizeNow
MAX_MALLOC_SIZE: 0x1fbd000 (with step = 0x1000);
Student >:testMalloc2
testMalloc2
We allocated 2 buffers.
BUF1(size=9, addr=0x20c30bd) filled with 9(+): +++++++++
BUF2(size=19, addr=0x20c30d0) filled with 19(,): ,,,,,,,,,,,,,,,,,,,

Student >:maxMallocSizeNow
maxMallocSizeNow
MAX_MALLOC_SIZE: 0x1fbd000 (with step = 0x1000);
```

最后是对`malloc`的测试，可以看到内存分配成功并且成功显示内容，同时也成功释放，这点可以从`maxMallocSize`中看出

至此，所有的测试都已完成

实验中遇到的问题

实验中遇到的最大的问题就是难以debug，因为在malloc成功之前很难对于错误有一个具象化的显示，只能是半猜半改。

另外，实验中对于强制类型转换使用的地方有很多，因此很容易产生计算错误，这也导致我一开始测试的时候地址乱飞，最后不得不重构代码。

```
    if (emb == 0){//tail of the linklist
        prev->nextStart = (unsigned long)freeSpace;
        freeSpace->nextStart = 0;
    } else{
        freeSpace->nextStart = (unsigned long)emb;
        if (emb == (EMB *)handler->firstFreeStart){
            handler->firstFreeStart = (unsigned long)freeSpace;
        } else{
            prev->nextStart = (unsigned long)freeSpace;
        }
    }

    emb = (EMB *)(handler->firstFreeStart);
    while (emb != 0){
        if ((unsigned long)(emb + 1) + emb->size == emb->nextStart){
            emb->size += ((EMB *)emb->nextStart)->size + EMB_size;
            emb->nextStart = ((EMB *)emb->nextStart)->nextStart;
            ((EMB *)emb->nextStart)->nextStart = 0;
            continue;
        }
        emb = (EMB *)emb->nextStart;
    }
```

例如这段就有很多的类型转换，易出错。