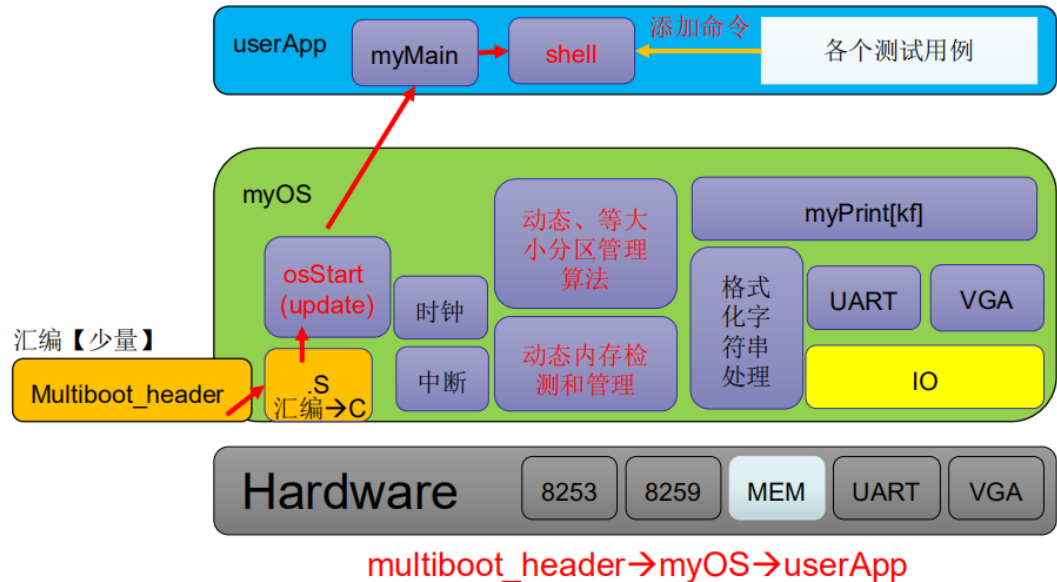


# Lab4 实验报告

PB18051098 徐碧涵

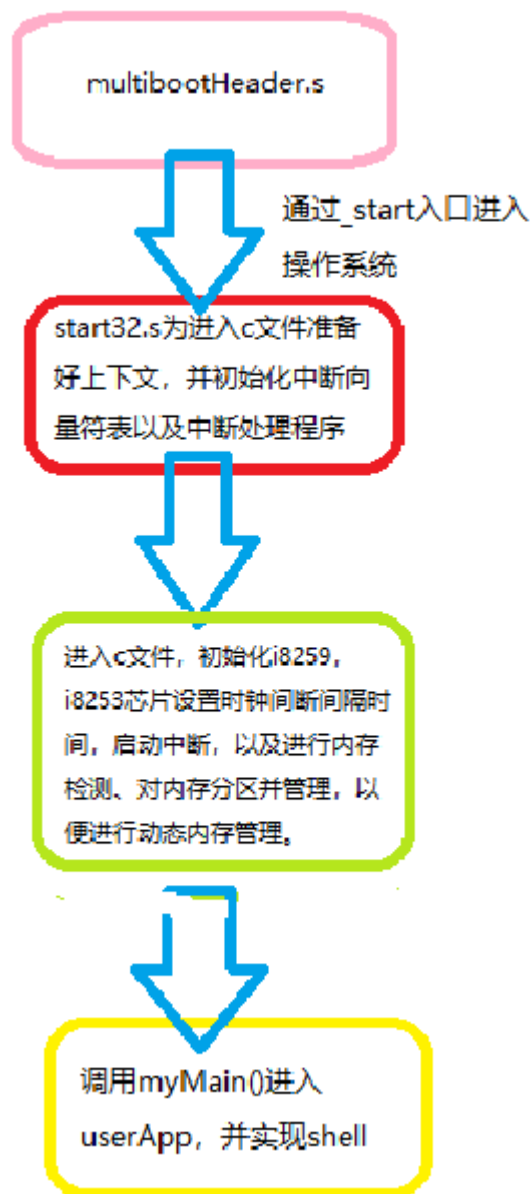
## 软件框图



本实验旨在实现内存管理,并利用 kmalloc/kfree 和 malloc/free 接口实现动态内存管理,最后通过 userApp 的 myMain 调用 shell 测试 malloc/free 的功能。

## 主流程及其实现

主流程如下:



通过遵循 multiboot 启动协议的 multibootheader.s 文件中的 multiboot\_header 头部启动内核，利用 `_start` 入口进入操作系统内部，执行 `start32.s` 文件，在其中为进入 c 文件准备好上下文，初始化中断向量表 IDT。随后进入 c 文件对操作系统进行初始化，通过 `init8259A()`, `init8253()` 初始化 i8259(可编程中断控制器)，i8253(可编程间隔定时器)芯片，然后启动时钟中断。之后调用 `pMemInit()` 进行内存检测并利用动态分区算法管理内存，为之后动态内存分配和回收做好准备。随后调用 `userApp` 入口进入 shell，实现 shell 的功能。

## 主要功能模块及其实现

### 主要功能模块

#### 内存检测

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

记录内存大小, 及初始地址。

算法如下:

```
while(1){
    p=(unsigned char*)(start+i*grainSize);//
    s=*p;
    t=*(p+1);
    *p=0xAA;
    *(p+1)=0x55;
    if(*p!=0xAA||*(p+1)!=0x55)break;
    *p=0x55;
    *(p+1)=0xAA;
    if(*p!=0x55||*(p+1)!=0xAA)break;
    *p=s;
    *(p+1)=t;
    p=(unsigned char*)(start+(i+1)*grainSize-2);
    s=*p;
    t=*(p+1);
    *p=0xAA;
    *(p+1)=0x55;
    if(*p!=0xAA||*(p+1)!=0x55)break;
    *p=0x55;
    *(p+1)=0xAA;
    if(*p!=0x55||*(p+1)!=0xAA)break;
    *p=s;
    *(p+1)=t;
    i++;
}
pMemSize=i*grainSize;
pMemStart=start;
```

动态内存

等大小分区管理算法

利用数据结构进行管理

```
// 一个EEB表示一个空闲可用的Block
struct EEB {
    unsigned long next_start;
};

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

实现 `eFPartitionWalkByAddr()` 打印 `eFPartition` 结构体的信息, 遍历每一个 `EEB`, 打印出他们的地址以及下一个 `EEB` 的地址

```
struct eFPartition *q=(struct eFPartition *)efpHandler;
struct EEB *p=(struct EEB *)q->firstFree;
showeFPartition(q);
while(p){
    showEEB(p);                //遍历链表并showEEB
    p=(struct EEB *)p->next_start;
}
```

实现 `eFPartitionInit()` 初始化内存创建 `eFPartition` 结构体, 对每一块的内存创建一个 `EEB`, 将他们连起来构成一个链。

```
if(perSize%4)q->persize=perSize/4*4+4; //对齐
q->totalN=n;
q->firstFree=start+12;                //+eFPartition的大小为12字节
p=(struct EEB *)q->firstFree;
for(i=1;i<n;i++){
    p->next_start=(unsigned long)p+q->persize;    //建立EEB链表
    p=p->next_start;
}
p->next_start=0;    //最后一块的EEB的nextstart是0
return start;
```

实现 `eFPartitionAlloc()` 函数分配一个空闲块的内存并返回相应的地址

```
struct eFPartition *q=(struct eFPartition *)EFPHander;
struct EEB *p=(struct EEB *)q->firstFree;
q->firstFree=p->next_start;    //将第一个空闲块分配
return q->firstFree+4;        //返回空闲块可使用的内存的首地址, +EEBsize
```

实现 `eFPartitionFree()`, free 掉以 `mbstart` 为首地址的一块已使用的一块内存

```

struct eFPartition *q=(struct eFPartition *)EFPHandler;
struct EEB *p=(struct EEB *)q->firstFree,*r=(struct EEB *)mbStart;
while(p&& p<mbStart){m=p; p=p->next_start;} //寻找r的插入点
m->next_start=mbStart;
r->next_start=(unsigned long)p;
return 1;

```

动态分区管理算法

*利用数据结构进行管理*

// EMB每一个block的数据结构，userdata可以暂时不用管。

```

struct EMB{
    unsigned long size;
    union {
        unsigned long nextStart; // if free: pointer to next block
        unsigned long userData; // if allocated, belongs to user
    }tag;
};

```

//dPartition 是整个动态分区内存的数据结构

```

struct dPartition{
    unsigned long size;
    unsigned long firstFreeStart;
};

```

利用 **dPartitionInit()** 初始化内存。在地址 **start** 处，首先要有 **dP** 结构体表示整个数据结构。然后，一整块的 **EMB** 被分配，在内存中紧紧跟在 **dP** 后面，然后 **dP** 的 **firstFreeStart** 指向 **EMB**。

```

//对dP进行初始化
struct dPartition *q=(struct dPartition *)start;
struct EMB *p;
q->size=totalSize;
q->firstFreeStart=(unsigned long)(q+1); //指向dP结构体之后的空闲区

p=(struct EMB *)q->firstFreeStart;
p->size=totalSize-8; //EMB大小应减去dP的size
p->tag.nextStart=0;
return start;

```

利用 **dPartitionWalkByAddr()** 打印 **dP** 的信息，然后按地址的大小遍历 **EMB**

```

struct dPartition *q=(struct dPartition *)dp;
struct EMB *p=(struct EMB *)q->firstFreeStart;
showdPartition(q);
while(p){
showEMB(p);
p=(struct EMB *)p->tag.nextStart;          //遍历EMB链表并showEMB

```

**利用 *dPartitionAllocFirstFit(unsigned long dp, unsigned long size)* 使用 *firstfit* 的算法分配空间，最后，成功分配返回首地址，不成功返回 0**

```

if(p->size>=size+8){
//若第一个空闲块p的size比需要的size+EMBSIZE至少大一个EMBSIZE，那么
//将p进行分割并修改dP的firstFreeStart
    if(p->size>size+8+8){
        r=(struct EMB *)((unsigned long)p+size+8);
        q->firstFreeStart=(unsigned long)r;
        r->size=p->size-size-8;
        p->size=size+8;
        r->tag.nextStart=p->tag.nextStart;
    }
    else q->firstFreeStart=p->tag.nextStart; //否则直接分配
}
else{
    while(p&&p->size<size+8){ //遍历EMB链表，找到足够大的空闲块
        m=p;
        p=(struct EMB*)p->tag.nextStart;
    }
    if(p){ //找到符合大小需求的空闲块p
        if(p->size>size+8+8){ //分割
            r=(struct EMB *)p+size+8;
            r->size=p->size-size-8;
            m->tag.nextStart=(unsigned long)r;
            p->size=size+8;
        }
        else m->tag.nextStart=p->tag.nextStart; //直接分配
    }
    else return 0; //无符合需求的空闲块，分配失败
}
return (unsigned long)p+8; //分配成功，返回用户可使用内存的首地址， +EMBSIZE

```

**利用 *dPartitionFreeFirstFit(unsigned long dp, unsigned long start)* 释放空间**

先将 start-8 使 start 指向该需要释放的内存块的 EMB 首地址，再判断是否可以与相邻的空闲区合并，若无则直接插入 EMB 链表中。

首先将 start 与 firstFreeStart 比较

```

if(start+r->size==q->firstFreeStart) //若需要释放的块r与firstFreeStart相邻，将两块合并
{q->firstFreeStart=start;
r->size+=p->size;
r->tag.nextStart=p->tag.nextStart;}
else if(start<q->firstFreeStart){ //若r首地址小于firstFreeStart将r作为空闲块插入
q->firstFreeStart=start;
r->tag.nextStart=(unsigned long)p;}

```

若 r 与 firstFreeStart 不相邻且首地址更大，那么进行如下操作。

```

else{
    while(p) //寻找与r前相邻的空闲块
    {if(p->size+(unsigned long)p==start)break;
      p=(struct EMB *)p->tag.nextStart;}
    if(p){ //若找到与r前相邻的空闲块
      m=(struct EMB *)p->tag.nextStart;
      //判断下一个空闲块是否与r后相邻
      if(p->tag.nextStart==start+r->size){ //将三个相邻块合并
        p->tag.nextStart=m->tag.nextStart;
        p->size=r->size+p->size+m->size;}
      else {p->tag.nextStart=p->tag.nextStart;
        p->size=r->size+p->size;}
    }
    else { //若无r的前相邻的空闲块，寻找r的插入点
      p=(struct EMB *)q->firstFreeStart;
      while((unsigned long)p<start&&p)
        {m=p; p=(struct EMB *)p->tag.nextStart;}
      if(p&&(unsigned long)p==start+r->size) { //若r有后相邻的空闲块
        r->tag.nextStart=p->tag.nextStart; //将两个空闲块合并
        m->tag.nextStart=start ; r->size=r->size+p->size;}
      else {r->tag.nextStart=m->tag.nextStart; //若无后相邻的空闲块
        m->tag.nextStart=start;} //将r作为空闲块插入
    }
  }
}

```

**实现 *dPartitionAlloc(unsigned long dp, unsigned long size)*、*dPartitionFree(unsigned long dp, unsigned long start)***

```

unsigned long dPartitionAlloc(unsigned long dp, unsigned long size){
    return dPartitionAllocFirstFit(dp,size);
}

unsigned long dPartitionFree(unsigned long dp, unsigned long start){
    return dPartitionFreeFirstFit(dp,start);
}

```

**shell**

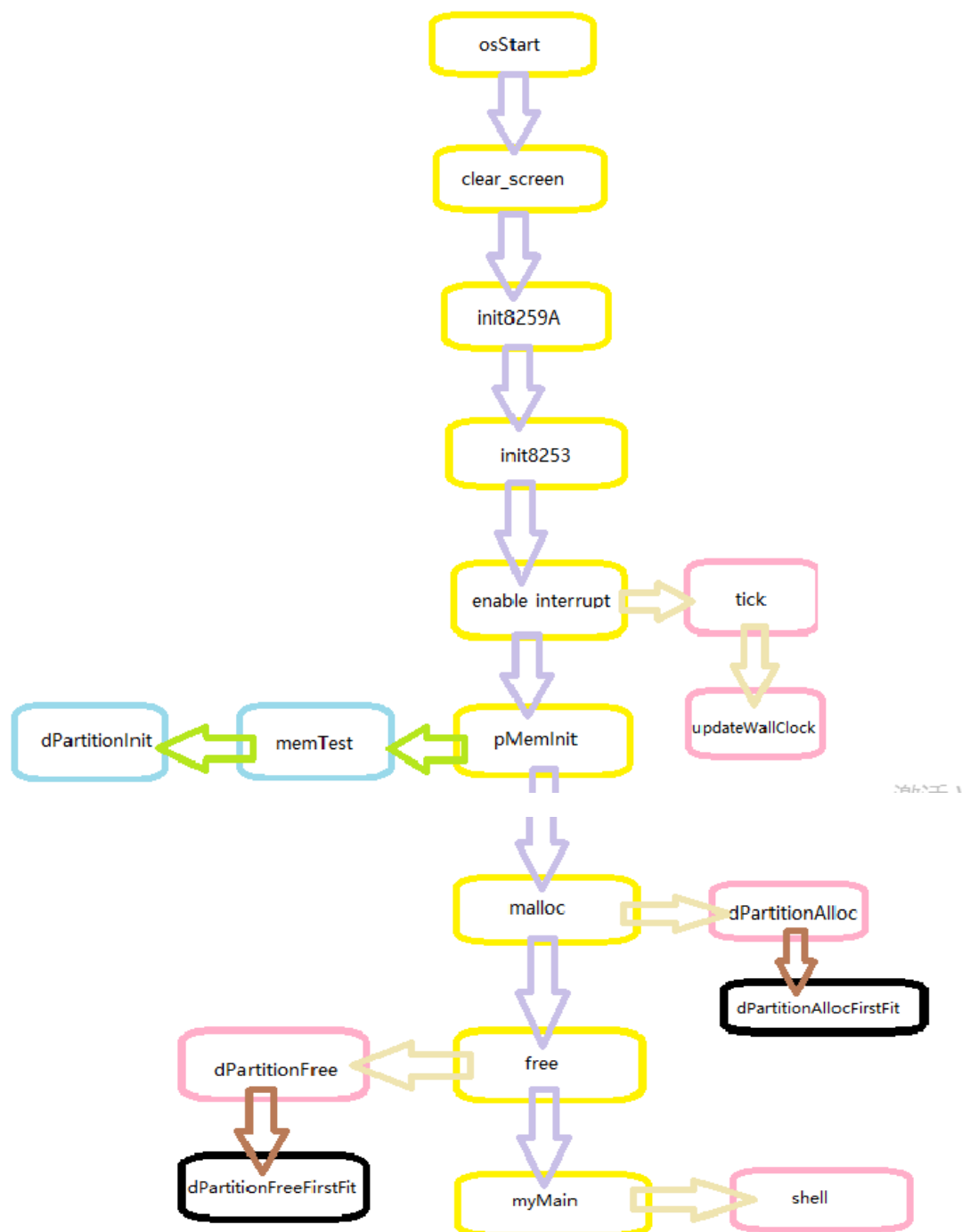
**利用 *addNewCmd()* 增加新命令，使用 *malloc* 创建一个 *cmd* 的结构体，新增命令。**

```

struct cmd *p;
p=(struct cmd *)(malloc(200));//+EMB的大小
strcpy(cmd,p->cmd);
p->func=func;
p->help_func=help_func;
strcpy(description,p->description);
//将新命令加入ourCmds链表
if(ourCmds==NULL){
    ourCmds=p;
    p->nextCmd=NULL;
}
else{
    p->nextCmd=ourCmds;
    ourCmds=p;
}

```

流程图如下：



## Makefile 组织

基本上每一个文件夹下都有一个 **Makefile** 文件将目标文件生成中间文件，便于后期整体组织

*目录组织如下：*

multibooheader



- multibootheader.s
- myOS
  - i386
    - io.c
    - irq.s
    - irqs.c
    - Makefile
  - dev
    - i8253.c
    - i8259A.c
    - uart.c
    - vga.c
    - Makefile
  - printk
    - vsprintf.c
    - myprintk.c
    - Makefile
  - kernel
    - mem
      - dPartition.c
      - eFPartition.c
      - Makefile
      - kmalloc.c
      - malloc.c
      - pMemInit.c
    - tick.c
    - WallClock.c
    - Makefile
  - lib
    - string.c
    - Makefile
  - Makefile
  - myOS.ld
  - osStart.c
  - start32.s
- userApp
  - main.c
  - shell.c
  - Makefile
- Makefile

source2img.sh

底层文件夹下的 Makefile 文件使当前文件夹下的.c 文件生成中间文件, 处于上层文件夹的 Makefile 将下层文件夹中生成的中间文件进行组织最后由最顶层文件夹下的 Makefile 汇总生成目标文件。

## 代码布局说明(地址空间)

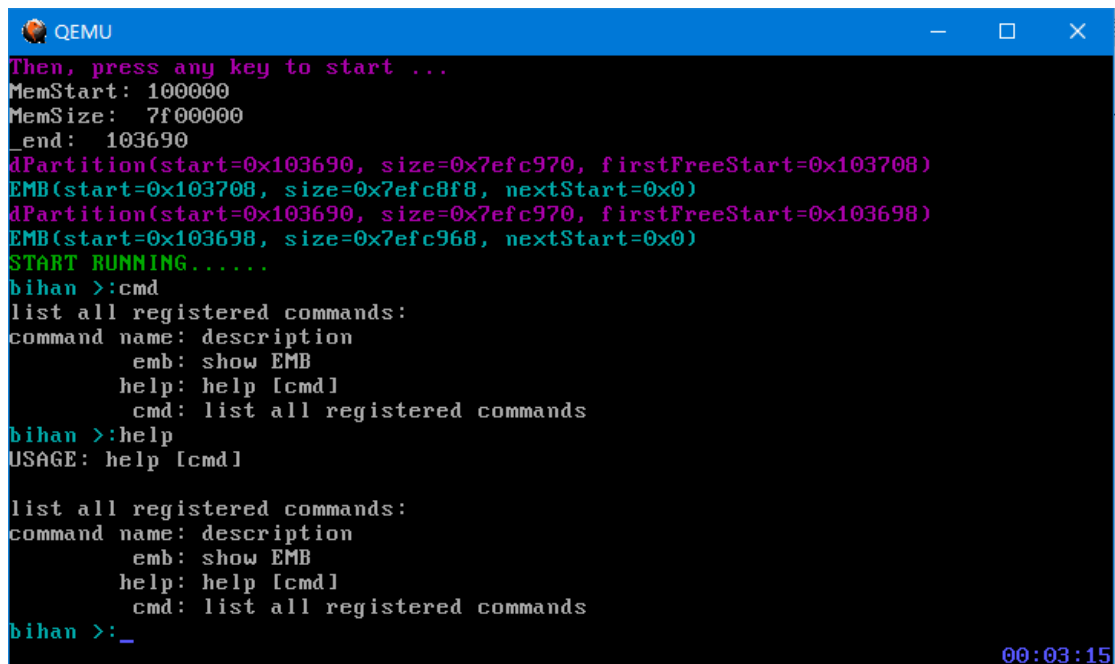
代码在内存空间中的地址以及布局由 myOS.ld 文件所确定, 代码将从物理内存 1M 的位置开始存放, 代码段 8 个字节对齐, 之后存放程序中已经初始化的全局变量并按 16 字节对齐, 在之后存放未初始化的全局变量按 16 字节对齐, 下一个存放的代码按 512 个字节对齐

## 编译过程说明

通过 ./source2img.sh 命令执行 make 命令在 Makefile 的组织下使所有.c.s 文件生成中间文件最终按照 myOS.ld 文件代码布局以及地址空间的要求在内存空间生成并存放.elf 文件。

## 运行和运行结果说明

输入 ./source2img.sh 命令运行脚本文件, 编译后进而执行.elf 文件, 运行结果如下:



```
QEMU
Then, press any key to start ...
MemStart: 100000
MemSize: 7f00000
MemEnd: 103690
dPartition(start=0x103690, size=0x7efc970, firstFreeStart=0x103708)
EMB(start=0x103708, size=0x7efc8f8, nextStart=0x0)
dPartition(start=0x103690, size=0x7efc970, firstFreeStart=0x103698)
EMB(start=0x103698, size=0x7efc968, nextStart=0x0)
START RUNNING.....
bihan >:cmd
list all registered commands:
command name: description
emb: show EMB
help: help [cmd]
cmd: list all registered commands
bihan >:help
USAGE: help [cmd]

list all registered commands:
command name: description
emb: show EMB
help: help [cmd]
cmd: list all registered commands
bihan >:_
```

00:03:15

## 遇到的问题和解决方案说明

问题: 在 memTest 时, 写入数据并读出, 若数据类型是, char 型, 则读写出现不一致  
解决方案: 改用 unsigned char 型进行读写。