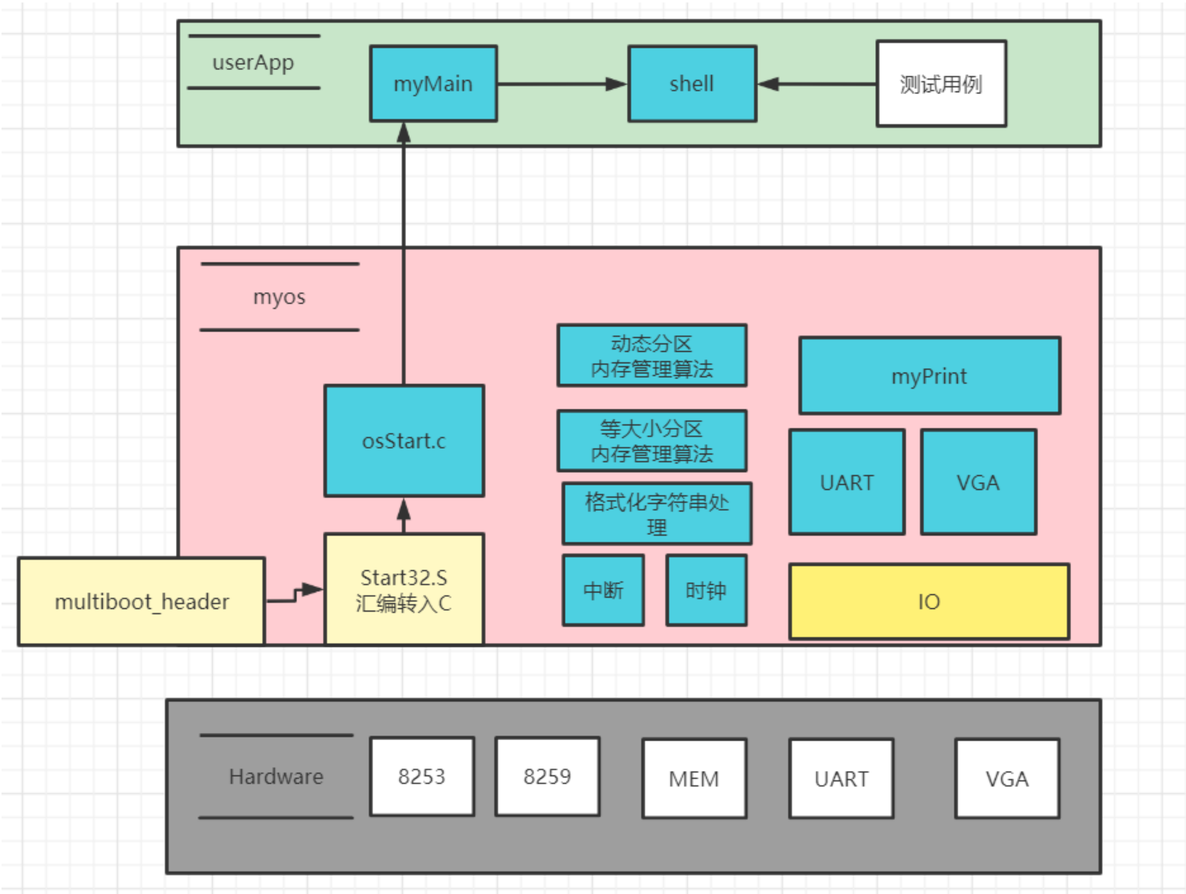
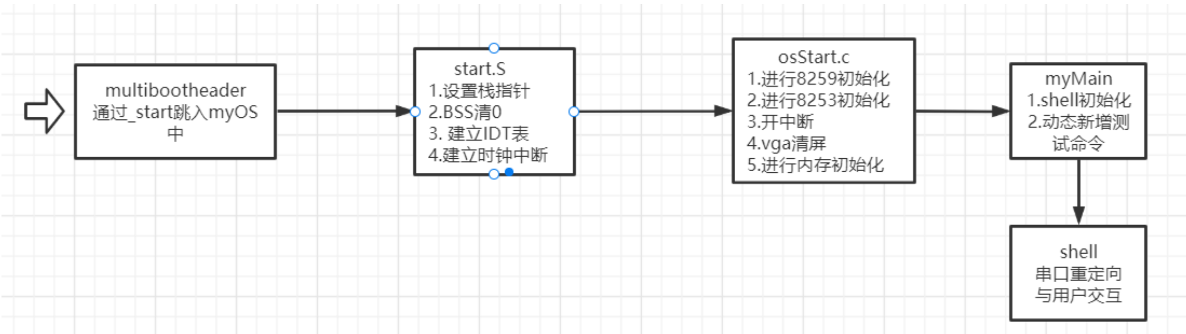
lab4实验报告

一、软件框图



二、主流程

1.流程图



2.文字说明

计算机开机之后，经过一些初始化步骤后，首先读取multibootheader文件，在校验对接完成之后，结尾处将借助myOS提供的_start 入口跳入myOS中。

进入myOS中首先执由汇编语言构成的start.S文件，这个文件会进行设置栈和将BSS 段清0两个操作，为下面的C语言执行提供必要的环境。接着进行IDT表的建立工作，为256个中断向量描述符开辟空间，初始化IDTR，并将每个向量均指向缺省函数。之后正式建立时间中断向量，跳入osStart.c中

进入osStart.c中调用库函数进行初始化工作（包括对8259，8253，mem的硬件初始化），初始化完成之后开中断，调用库函数进行VGA清屏，输出开始信息，之后便跳入userApp执行main.c函数。

mian.c函数初始化shell之后，利用动态增添命令功能添加测试命令，通过调用os提供的函数接口调用startShell()函数启动简易shell程序，启动完成后在终端与用户进行交互，读取用户命令，并进行处理。

三、功能模块

1.内存检测

```
void memTest(unsigned long start, unsigned long grainSize){
    // 内存检测从1M以上开始
    int16* p = (int16*)start;
    int16 tmp;
    if(start < 0x100000)
        myPrintk(0x7, "start should > 1M\n");
    // grainsize 不能小于 512字节
    if(grainSize <= 0x200)
        myPrintk(0x7, "grainSize should > 512B");
    //开始检测
    do{
        tmp = *p;
        //检测前两个字节
        *p = 0xAA55;
        if(*p != 0xAA55){
            *p = tmp;
            break;
        }
        *p = 0x55AA;
        if(*p != 0x55AA){
            *p = tmp;
            break;
        }
        *p = tmp; // 还原数据

        //检测后两个字节

        p += ((grainSize >>1) - 1);

        tmp = *p;
        *p = 0xAA55;
        if(*p != 0xAA55){
            *p = tmp;
            p -= ((grainSize >>1) - 1); //退出前要回退
            break;
        }
        *p = 0x55AA;
        if(*p != 0x55AA){
            *p = tmp;
            p -= ((grainSize >>1) - 1);
            break;
        }
        // 还原数据
        *p = tmp;
        p += 1;
```

```
    }while(1);
    // 设置全局变量
    pMemStart = start;
    pMemSize = (unsigned long)p -start;
    myPrintk(0x7,"MemStart: 0x%x  \n", pMemStart);
    myPrintk(0x7,"MemSize:  0x%x  \n", pMemSize);

}
```

可用内存是指可正常工作和使用的内存，而不是指空闲内存

以步长为单位，在每单位的内存上进行头两个字节和后两个字节的检测，检测方法为输入特殊字符序列，再读出，判断读出的字符串是否与输入相同，即可判断内存是否可用。

2. 内存初始化

```
extern unsigned long _end;
void pMemInit(void){
    unsigned long kernelSpaceSize = 0x50000;

    unsigned long _end_addr = (unsigned long) &_end;
    memTest(0x100000,0x1000);
    myPrintk(0x7,"_end:  0x%x  \n", _end_addr);
    if (pMemStart <= _end_addr) {
        pMemSize -= _end_addr - pMemStart;
        pMemStart = _end_addr;
    }
    //先从简单的等大小测试，块大小设置为4k
    //pMemHandler = eFPartitionInit(pMemStart, 0x1000, pMemSize/(0x1000) - 1);
    //进行动态内存管理测试

    //内核态和用户态
    kpMemHandler = dPartitionInit(pMemStart, kernelSpaceSize);
    pMemHandler  = dPartitionInit(pMemStart + kernelSpaceSize, pMemSize -
kernelSpaceSize);
}
```

利用测得的最大内存与外部变量_end（指示OS结束的位置），对内存进行初始化。如果要区分内核与用户态，需要调用两次初始化实现函数，返回内核和用户内存的句柄。

3. 动态分区管理算法

a) 初始化具体实现函数

```
unsigned long dPartitionInit(unsigned long start, unsigned long totalSize){
    if(totalSize < dPartition_size + EMB_size)
        return 0;

    dPartition partition;
    partition.size = totalSize - dPartition_size;
    partition.firstFreeStart = start + dPartition_size;
    memcpy((void *)start, &partition, dPartition_size);

    EMB emb;
    emb.size = totalSize - dPartition_size - EMB_size/2;
    emb.nextStart = 0; //起始只有一块emb
```

```
    memcpy((void *)(start + dPartition_size), &emb, EMB_size);

    return start;
}
```

如图，初始化的主要任务为在start处创建一个dPartition结构体和一个EMB结构体，前者用来记录整个可用内存的信息，后者记录它管理的块的信息。

b) 分配算法

```
unsigned long dPartitionAllocFirstFit(dPartition* dp, unsigned long size){
    //从链表第一项开始判断
    EMB* tmp;
    EMB* pemb = (EMB*) dp->firstFreeStart;
    while(pemb != NULL){
        if( pemb->size > size){
            //满足进行分配，并且修改链表
            //若有大于emb块的剩余空间进行切分
            if(pemb->size - size > EMB_size){
                EMB emb;
                unsigned long embstart = (unsigned long)pemb + EMB_size/2 +
size;
                emb.size = pemb->size - size - EMB_size/2;
                //与上一个结点连接
                if(pemb == (EMB*) dp->firstFreeStart )
                    dp->firstFreeStart = embstart;
                else{
                    tmp = (EMB*) dp->firstFreeStart;
                    while(tmp->nextStart != (unsigned long)pemb)
                        tmp = (EMB*) tmp->nextStart;
                    tmp->nextStart = embstart;
                }
                //与下一个结点连接
                emb.nextStart = pemb->nextStart;
                pemb->size = size;
                //将建好的emb放入内存中
                memcpy((void *)embstart, &emb, EMB_size);
            }
            else{
                //剩余碎片太小，不再切割，转换为内部碎片
                //则不再创建新的emb，直接对链表进行重新连接
                if(pemb == (EMB*) dp->firstFreeStart )
                    dp->firstFreeStart = pemb->nextStart;
                else{
                    tmp = (EMB*) dp->firstFreeStart;
                    while(tmp->nextStart != (unsigned long)pemb)
                        tmp = (EMB*) tmp->nextStart;
                    tmp->nextStart = pemb->nextStart;
                }
                pemb->size += EMB_size/2;
            }
            return (unsigned long) ((unsigned long)pemb+ EMB_size/2);//返回地址为
emb结构体size结束，union的首地址
        }
        else{
            pemb = (EMB*) pemb->nextStart;
        }
```

```
    }
    myPrintf(0x7, "no space for malloc!\n");
    return (unsigned long)0x0;
}
```

分配算法使用firstfit，在链表中从前到后选择第一个可满足条件的内存。还要注意对齐问题。

需要特别注意的是：在判断该块的大小时，是要加上*EMB*结构体中原来用来指示下一个*EMB*的指针占用内存的大小，因为*EMB*结构体如下：

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

c) 回收算法

```
unsigned long dPartitionFreeFirstFit(dPartition* dp, unsigned long start){
    int mergeBefore = 0, mergeAfter = 0;
    unsigned long embstart = start - EMB_size/2;
    unsigned long embsize = ((EMB*)embstart)->size;

    //检查要释放的start~end这个范围是否在dp有效分配范围内
    if(embstart < (unsigned long)dp + dPartition_size ||
        start + embsize > (unsigned long)dp + dPartition_size + dp->size)
        return 0;
    else{
        //释放内存
        //寻找应该插入的位置
        EMB* p =  (EMB*) dp->firstFreeStart;
        EMB* pre = p;

        while( p != NULL && embstart > (unsigned long)p ){
            pre = p;
            p = (EMB*)p->nextStart;
        }
        //再次检查要释放的start~end这个范围是否在dp有效分配范围内
        //pre != p 和 p != NULL保证该emb不在头尾
        if(( pre != p && (unsigned long)pre + EMB_size/2 + pre->size > embstart)
||
            ( p != NULL && start + embsize > (unsigned long)p) )
            return 0;
        //判断是否可以向前和向后合并
        if(start + embsize == (unsigned long)p)
            mergeAfter = 1;
        if((unsigned long)pre + EMB_size/2 + pre->size == embstart)
            mergeBefore = 1;
        //四种情况
        if(!mergeBefore && !mergeAfter){
            if(embstart > dp->firstFreeStart)
                pre->nextStart = embstart;
            else
                dp->firstFreeStart = embstart;
```

```
            ((EMB*)embstart)->size = embsize;
            ((EMB*)embstart)->nextStart = (unsigned long)p;
        }
        else if(!mergeBefore && mergeAfter){
            if(embstart > dp->firstFreeStart)
                pre->nextStart = embstart;
            else
                dp->firstFreeStart = embstart;

            ((EMB*)embstart)->size = embsize + EMB_size/2 + p->size;
            ((EMB*)embstart)->nextStart = p->nextStart;
        }
        else if(mergeBefore && !mergeAfter){
            pre->nextStart = (unsigned long)p;
            pre->size += (embsize + EMB_size/2);
        }
        else{
            pre->size += ( EMB_size + embsize + p->size);
            pre->nextStart = p->nextStart;
        }
        return 1;
    }
}
```

该算法的关键在于相邻空闲内存的合并，具体实现方法为，每次释放时都检测和它相邻的左右两个块，会有四种不同情况，对四种情况分别做处理。

4.等大小分区管理算法

a) 内存初始化

```
unsigned long eFPartitionInit(unsigned long start, unsigned long perSize,
unsigned long n){
    //对传入的size做对齐，默认8字节对齐
    unsigned long alignSize = 8 * ( (perSize / 8) + ((perSize % 8) > 0 ? 1 : 0)
);
    //创建eFPartition结构体
    eFPartition partition;
    partition.perSize = alignSize;
    partition.firstFree = (unsigned long)(start + eFPartition_size);
    partition.totalN = n;
    memcpy((void*)start, &partition, eFPartition_size);
    //开辟空闲内存块，创建链表
    EEB eeb;
    EEB* p = (EEB*)partition.firstFree;
    for(int i = 1; i < n; i++){
        eeb.next_start = (unsigned long)(p) + partition.perSize + EEB_size;
        memcpy((void*)p, &eeb, EEB_size);
        p = (EEB*)eeb.next_start;
    }
    eeb.next_start = 0; //对i = n 特殊处理
    memcpy((void*)p, &eeb, EEB_size);
    //返回句柄
    return start;
}
```

与动态分区算法不同的时，等大小在初始化时，要产生多个EEB块来管理整个可用内存空间，使用利用for循环即可。

b) 分配算法

```
unsigned long eFPartitionAlloc(unsigned long EFPHandler){

    //对于等大小分区，内存分配出去后，还是将EEB保存下来
    eFPartition* p_partition = (eFPartition*)EFPHandler;
    EEB* p_firstEEB = (EEB*)p_partition->firstFree;
    if(p_firstEEB == NULL){
        myPrintf(0x7, "no space!!!\n");
        return 0;
    }
    p_partition->firstFree = p_firstEEB->next_start;
    return (unsigned long)((unsigned long)p_firstEEB + EEB_size);

}
```

等大小分配算法较为简单，摘取空闲链表的第一项，返回对应的首地址即可。

c) 回收算法

```
unsigned long eFPartitionFree(unsigned long EFPHandler,unsigned long mbStart){
    eFPartition* p_partition = (eFPartition*)EFPHandler;
    EEB* p_eeb = (EEB*)(mbStart - EEB_size);
    EEB *pre, *p;
    p = pre = (EEB*)p_partition->firstFree;
    //按内存地址从低到高维护链表
    if(p_eeb < p){
        p_eeb->next_start = (unsigned long) p;
        p_partition->firstFree = (unsigned long)p_eeb;
    }
    else{
        //找到eeb应该插入的位置
        while((unsigned long)p != 0 && p_eeb > p){
            pre = p;
            p = (EEB*)p->next_start;
        }
        p_eeb->next_start = (unsigned long) p;
        pre->next_start = (unsigned long)p_eeb;
    }
    return 0;
}
```

回收也较为简单，无需考虑相邻空闲块的合并，只要按物理内存地址高低在相应位置插入即可。

四、源代码说明

1.目录组织

```
$ tree
.
|-- Makefile
|-- multibootheader
|   `-- multibootHeader.S
|-- myOS
|   |-- Makefile
|   |-- dev
|   |   |-- Makefile
|   |   |-- i8253.c
|   |   |-- i8259A.c
|   |   |-- uart.c
|   |   `-- vga.c
|   |-- i386
|   |   |-- Makefile
|   |   |-- io.c
|   |   |-- irq.S
|   |   `-- irqs.c
|   |-- include
|   |   |-- i8253.h
|   |   |-- i8259A.h
|   |   |-- io.h
|   |   |-- irqs.h
|   |   |-- kmalloc.h
|   |   |-- malloc.h
|   |   |-- mem.h
|   |   |-- myPrintk.h
|   |   |-- mymath.h
|   |   |-- mystring.h
|   |   |-- tick.h
|   |   |-- uart.h
|   |   |-- vga.h
|   |   |-- vsprintf.h
|   |   `-- wallClock.h
|   |-- kernel
|   |   |-- Makefile
|   |   |-- mem
|   |   |   |-- Makefile
|   |   |   |-- dPartition.c
|   |   |   |-- eFPartition.c
|   |   |   |-- kmalloc.c
|   |   |   |-- malloc.c
|   |   |   `-- pMemInit.c
|   |   |-- tick.c
|   |   `-- wallClock.c
|   |-- lib
|   |   |-- Makefile
|   |   |-- myPrintk.c
|   |   |-- mymath.c
|   |   |-- mystring.c
|   |   `-- vsprintf.c
|   |-- myOS.ld
|   |-- osStart.c
|   |-- start32.S
|   `-- userInterface.h
```

```
|-- output
|   |-- multibootheader
|   |   `-- multibootHeader.o
|   |-- myOS
|   |   |-- dev
|   |   |   |-- i8253.o
|   |   |   |-- i8259A.o
|   |   |   |-- uart.o
|   |   |   `-- vga.o
|   |   |-- i386
|   |   |   |-- io.o
|   |   |   |-- irq.o
|   |   |   `-- irqs.o
|   |   |-- kernel
|   |   |   |-- mem
|   |   |   |   |-- dPartition.o
|   |   |   |   |-- eFPartition.o
|   |   |   |   |-- kmalloc.o
|   |   |   |   |-- malloc.o
|   |   |   |   `-- pMemInit.o
|   |   |   |-- tick.o
|   |   |   `-- wallClock.o
|   |   |-- lib
|   |   |   |-- myPrintk.o
|   |   |   |-- mymath.o
|   |   |   |-- mystring.o
|   |   |   `-- vsprintf.o
|   |   |-- osStart.o
|   |   `-- start32.o
|   |-- myOS.elf
|   `-- userApp
|       |-- main.o
|       |-- memTestCase.o
|       `-- shell.o
|-- source2run.sh
`-- userApp
    |-- Makefile
    |-- main.c
    |-- memTestCase.c
    |-- memTestCase.h
    |-- shell.c
    `-- shell.h
```

2.Makefile组织

```
.
├── MULTI_BOOT_HEADER
│ └── output/multibootheader/multibootHeader.o
├── MYOS_OBJS
│ ├── 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
| ├── 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
| |    └──output/myOS/kernel/mem/kmalloc.o
| ├── LIB_OBJS
| | ├── output/myOS/lib/vsprintf.o
| | ├── output/myOS/lib/myPrintk.o
| | ├── output/myOS/lib/mymath.o
| | └── output/myOS/lib/mystring.o
| ├── output/myOS/osStart.o
| └── output/myOS/start32.o
└── USER_APP_OBJS
  ├── output/userApp/main.o
  ├── output/userApp/shell.o
  └── output/userApp/memTestCase.o
```

五、代码布局说明

地址空间排布如下：

1. 从1M的内存地址开始，首先放multiboot_header，写入12个字节 。
2. 要求8字节对齐，.text代码从16字节开始。
3. 放完.text后，进行16字节对齐。接着放入.data数据。
4. 再次进行16字节对齐后为全局变量分配空间。
5. 16字节对齐后，最后为栈分配空间。进行512字节对齐。

在.data数据段对IDT的分配代码如下

```
/* ===== data =========================== */
.data
# IDT
    .p2align 4
    .globl IDT
IDT:
    .rept 256
    .word 0,0,0,0
    .endr
idtptr:
    .word (256*8 - 1)
    .long IDT
```

.p2align 4的意思为按照2的4次方进行对齐，即16字节对齐。

关于.S中的.p2align 4与.ld中ALIGN()的区别和联系

> .ld中的对齐方式是在段层面，它只作用于该段最开始的数据。

> 而每个段都是有许多部分构成的，如data段由IDT与其他数据构成。若IDT不在该段的最开始的地方，则无法保证其16字节对齐，所以要在.S相应位置加入.p2align 4描述内存分布，保证该部分被16字节对齐。

对VGA显存部分，每两个字节间放一个字符，第一个放到0xB8000,第二个从0xB8002开始。

六、编译过程说明

具体操作为直接运行 source2run.sh 脚本，进行 make和串口重定向

七、运行和运行结果说明

- **EMB本身大小为8字节，但是后四位是共用体，因此在计算该块内存开端或者该块的大小时按4字节算！！！**
- **代码默认为动态分区管理算法，若要改为等大小分区，请在pMemInit.c和malloc.c中修改相应的接口**

1. 动态分区算法内存初始化

```
MemStart:  0x100000
MemSize:   0x7f00000
_end:   0x106650
```

整个可用内存大小为0x8000000，但内存检测从0x100000开始，则最后测得MemSize为0x7f00000

```
cmd_size = 0x8c
```

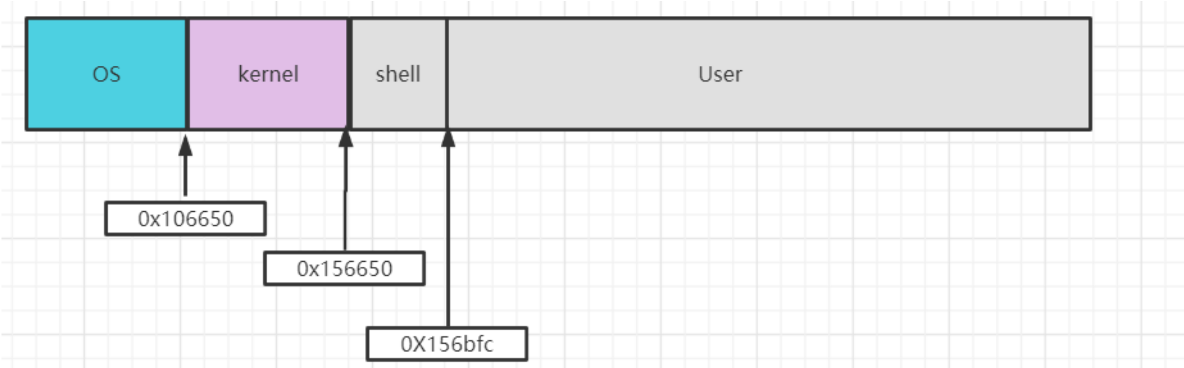
将每次新增shell命令的首地址打印，如下：

```
newcmd_start = 0x15665c
newcmd_start = 0x1566ec
newcmd_start = 0x15677c
newcmd_start = 0x15680c
newcmd_start = 0x15689c
newcmd_start = 0x15692c
newcmd_start = 0x1569bc
newcmd_start = 0x156a4c
newcmd_start = 0x156adc
newcmd_start = 0x156b6c
```

则最后一个shell命令结构体结束地址为0x156bf8，进行内存切割后，在其后放EMB结构体。

EMB大小按4字节算，则结束地址为0x156bf8+0x4 = 0x156bfc

动态分区的内存分布图如下：



2. testdP1

```
We had successfully malloc() a small memBlock (size=0x100, addr=0x156bfc);
It is initialized as a very small dPartition;
dPartition(start=0x156bfc, size=0xf8, firstFreeStart=0x156c04)
EMB(start=0x156c04, size=0xf4, nextStart=0x0)
Alloc a memBlock with size 0x10, success(addr=0x156c08)!......Relaesed;
Alloc a memBlock with size 0x20, success(addr=0x156c08)!......Relaesed;
Alloc a memBlock with size 0x40, success(addr=0x156c08)!......Relaesed;
Alloc a memBlock with size 0x80, success(addr=0x156c08)!......Relaesed;
no space for malloc!
Alloc a memBlock with size 0x100, failed!
Now, converse the sequence.
no space for malloc!
Alloc a memBlock with size 0x100, failed!
Alloc a memBlock with size 0x80, success(addr=0x156c08)!......Relaesed;
Alloc a memBlock with size 0x40, success(addr=0x156c08)!......Relaesed;
Alloc a memBlock with size 0x20, success(addr=0x156c08)!......Relaesed;
Alloc a memBlock with size 0x10, success(addr=0x156c08)!......Relaesed;
```

在0x156bfc进行内存分配，返回的句柄即为0x156bfc，第一行结果正确。

在0x156bfc进行内存初始化，会创建一个dPartition结构体，大小为8字节。紧接着为EMB结构体，EMB起始地址为0x156bfc + 0x8 = 0x156c04，结束地址0x156c04 + 0x4 =0x156c08 为结果正确。

因为初始化内存大小为0x100，扣除前面的结构体可用内存实际为0xf4。因此进行小于0xf4的内存分配会成功，而大于0xf4的会失败。因此前4次分配成功，最后一次失败。逆序结果同理。

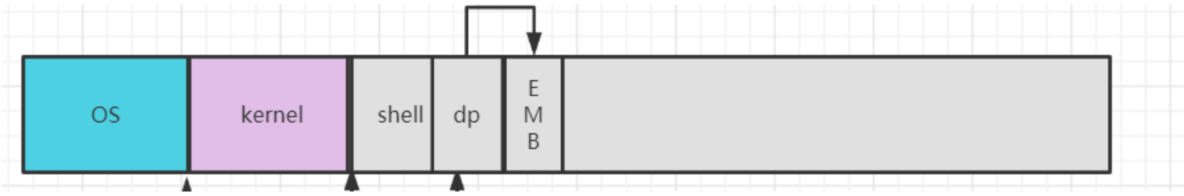
（分配失败时会打印 no space for malloc!）

3.testdP2

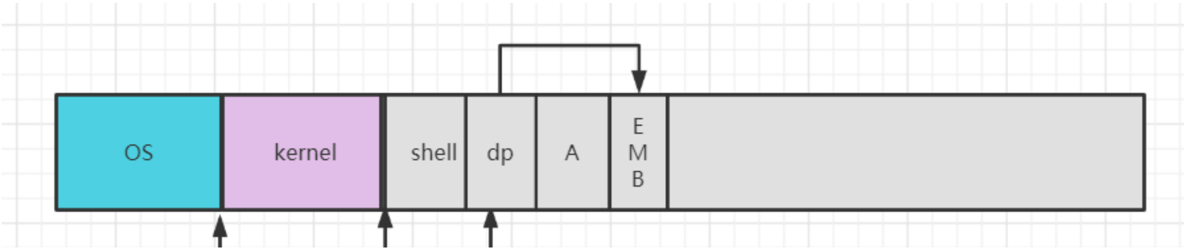
```
dPartition(start=0x156bfc, size=0xf8, firstFreeStart=0x156c04)
EMB(start=0x156c04, size=0xf4, nextStart=0x0)
Now, A:B:C:- ==> -:B:C:- ==> -:C- ==> - .
Alloc memBlock A with size 0x10: success(addr=0x156c08)!
dPartition(start=0x156bfc, size=0xf8, firstFreeStart=0x156c18)
EMB(start=0x156c18, size=0xe0, nextStart=0x0)
Alloc memBlock B with size 0x20: success(addr=0x156c1c)!
dPartition(start=0x156bfc, size=0xf8, firstFreeStart=0x156c3c)
EMB(start=0x156c3c, size=0xbc, nextStart=0x0)
Alloc memBlock C with size 0x30: success(addr=0x156c40)!
dPartition(start=0x156bfc, size=0xf8, firstFreeStart=0x156c70)
EMB(start=0x156c70, size=0x88, nextStart=0x0)
Now, release A.
dPartition(start=0x156bfc, size=0xf8, firstFreeStart=0x156c04)
EMB(start=0x156c04, size=0x10, nextStart=0x156c70)
EMB(start=0x156c70, size=0x88, nextStart=0x0)
Now, release B.
dPartition(start=0x156bfc, size=0xf8, firstFreeStart=0x156c04)
EMB(start=0x156c04, size=0x34, nextStart=0x156c70)
EMB(start=0x156c70, size=0x88, nextStart=0x0)
At last, release C.
dPartition(start=0x156bfc, size=0xf8, firstFreeStart=0x156c04)
EMB(start=0x156c04, size=0xf4, nextStart=0x0)
```

内存中各个参数的解释同testdP1。下面仅说明ABC的分配释放过程。

初始化内存如下

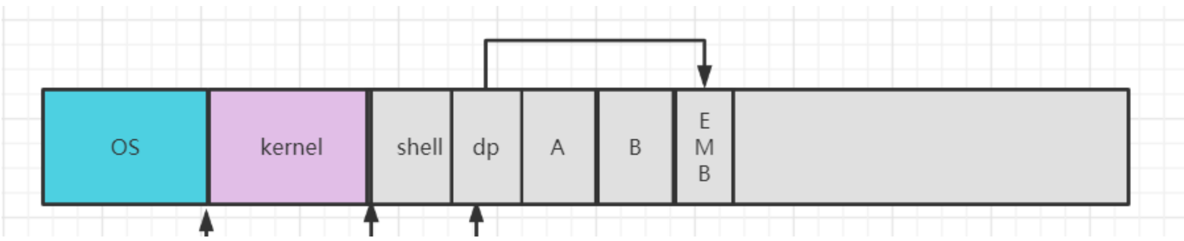


申请A空间



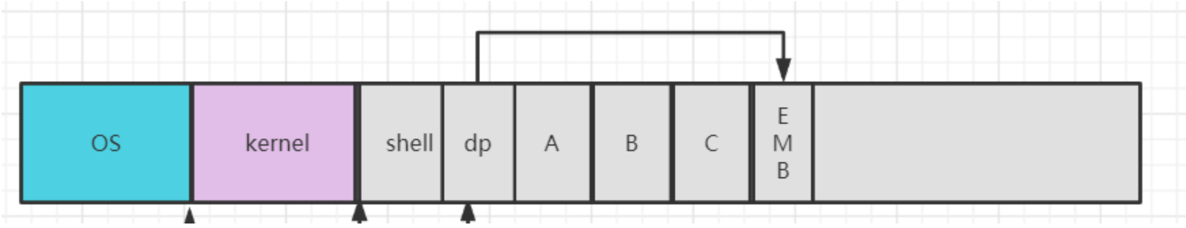
A空间首地址为原EMB结构体首地址加EMB长度，A空间结束地址为首地址加size_A

申请B空间

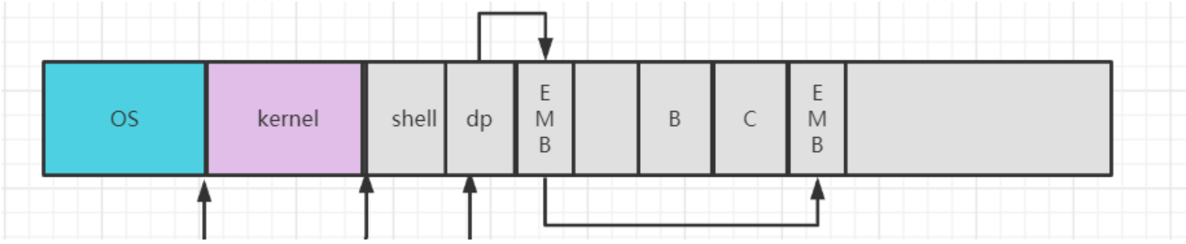


B空间首尾地址与A同理

申请C空间

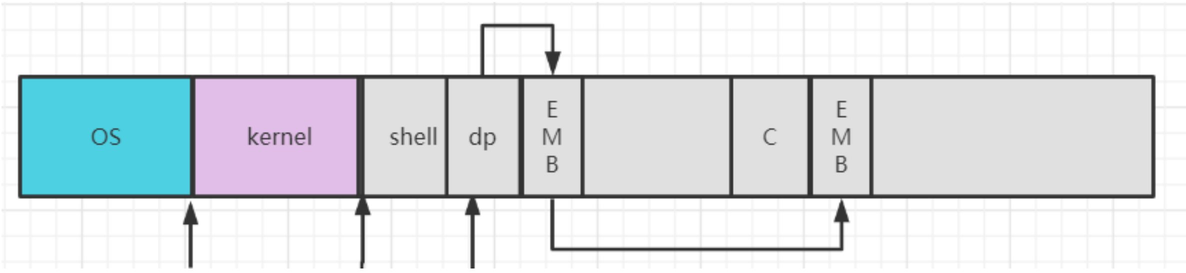


释放A空间



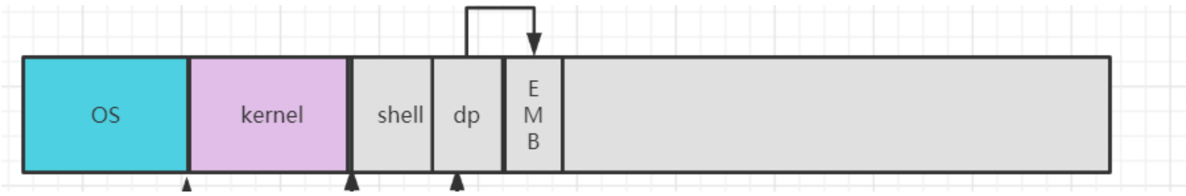
会产生两个EMB，输出结果符合预期

释放B空间



释放B后会进行相邻空闲块合并，EMB数量不变，符合预期。

释放C空间



释放C后，各种参数应该与初始化后的参数一致，符合预期

4.testdP3

```
We had successfully malloc() a small memBlock (size=0x100, addr=0x156bbc);
It is initialized as a very small dPartition;
dPartition(start=0x156bbc, size=0xf8, firstFreeStart=0x156bc4)
EMB(start=0x156bc4, size=0xf4, nextStart=0x0)
Now, A:B:C:- ==> -:B:C:- ==> -:C- ==> - .
Alloc memBlock A with size 0x10: success(addr=0x156bc8)!
dPartition(start=0x156bbc, size=0xf8, firstFreeStart=0x156bd8)
EMB(start=0x156bd8, size=0xe0, nextStart=0x0)
Alloc memBlock B with size 0x20: success(addr=0x156bdc)!
dPartition(start=0x156bbc, size=0xf8, firstFreeStart=0x156bfc)
EMB(start=0x156bfc, size=0xbc, nextStart=0x0)
Alloc memBlock C with size 0x30: success(addr=0x156c00)!
dPartition(start=0x156bbc, size=0xf8, firstFreeStart=0x156c30)
EMB(start=0x156c30, size=0x88, nextStart=0x0)
At last, release C.
dPartition(start=0x156bbc, size=0xf8, firstFreeStart=0x156bfc)
EMB(start=0x156bfc, size=0xbc, nextStart=0x0)
Now, release B.
dPartition(start=0x156bbc, size=0xf8, firstFreeStart=0x156bd8)
EMB(start=0x156bd8, size=0xe0, nextStart=0x0)
Now, release A.
dPartition(start=0x156bbc, size=0xf8, firstFreeStart=0x156bc4)
EMB(start=0x156bc4, size=0xf4, nextStart=0x0)
```

测试3和2大致相同，不同点在于释放内存的顺序，始终选择最后一块释放，因此EMB数量一直为1，与预期相符，测试通过。

5.testdP4

```
testdP4
x = 0x10665c
We had successfully kmalloc() a small memBlock (size=0x100, addr=0x10665c);
It is initialized as a very small dPartition;
dPartition(start=0x10665c, size=0xf8, firstFreeStart=0x106664)
EMB(start=0x106664, size=0xf4, nextStart=0x0)
Alloc a memBlock with size 0x10, success(addr=0x106668)!......Relaesed;
Alloc a memBlock with size 0x20, success(addr=0x106668)!......Relaesed;
Alloc a memBlock with size 0x40, success(addr=0x106668)!......Relaesed;
Alloc a memBlock with size 0x80, success(addr=0x106668)!......Relaesed;
no space for malloc!
Alloc a memBlock with size 0x100, failed!
Now, converse the sequence.
no space for malloc!
Alloc a memBlock with size 0x100, failed!
Alloc a memBlock with size 0x80, success(addr=0x106668)!......Relaesed;
Alloc a memBlock with size 0x40, success(addr=0x106668)!......Relaesed;
Alloc a memBlock with size 0x20, success(addr=0x106668)!......Relaesed;
Alloc a memBlock with size 0x10, success(addr=0x106668)!......Relaesed;
```

测试4是针对kernel态进行测试！！！

与用户态最大不同是kernel态内存起始地址与调用的分配释放接口不同

内核内存起始地址为0x106650，因为开始已经进行了一次初始化，则新的初始化的dPartition结构体从0x106650 + 0xc开始，第一块EMB结构体起始地址为0x10665c+ 0xc 开始。下面的分配释放过程与testdP1相同。

5. 等大小分区算法内存初始化

```
MemStart:  0x100000
MemSize:   0x7f00000
_end:  0x106630
```

```
newcmd_start = 0x106640
newcmd_start = 0x107644
newcmd_start = 0x108648
newcmd_start = 0x10964c
newcmd_start = 0x10a650
newcmd_start = 0x10b654
newcmd_start = 0x10c658
newcmd_start = 0x10d65c
newcmd_start = 0x10e660
newcmd_start = 0x10f664
```

_end与动态分区结果不同的原因：因为内核代码变了，结束位置就不同了

与动态分区算法一样进行相关数值检验，结果正确。

6. testeFP

```
It is initialized as a very small ePartition;
eFPartition(start=0x110668, totalN=0x4, perSize=0x20, firstFree=0x110674)
EEB(start=0x110674, next=0x110698)
EEB(start=0x110698, next=0x1106bc)
EEB(start=0x1106bc, next=0x1106e0)
EEB(start=0x1106e0, next=0x0)
Alloc memBlock A, start = 0x110678: 0xaaaaaaaa
eFPartition(start=0x110668, totalN=0x4, perSize=0x20, firstFree=0x110698)
EEB(start=0x110698, next=0x1106bc)
EEB(start=0x1106bc, next=0x1106e0)
EEB(start=0x1106e0, next=0x0)
Alloc memBlock B, start = 0x11069c: 0xbbbbbbbb
eFPartition(start=0x110668, totalN=0x4, perSize=0x20, firstFree=0x1106bc)
EEB(start=0x1106bc, next=0x1106e0)
EEB(start=0x1106e0, next=0x0)
Alloc memBlock C, start = 0x1106c0: 0xcccccccc
eFPartition(start=0x110668, totalN=0x4, perSize=0x20, firstFree=0x1106e0)
EEB(start=0x1106e0, next=0x0)
Alloc memBlock D, start = 0x1106e4: 0xdddddddd
eFPartition(start=0x110668, totalN=0x4, perSize=0x20, firstFree=0x0)
no space!!!
Alloc memBlock E, failed!
eFPartition(start=0x110668, totalN=0x4, perSize=0x20, firstFree=0x0)
```

因为内存初始化只有四块，则前四个分配成功，最后一个分配失败。

(分配失败时打印 no space!!!)

写入后读出的数值与原数值相同，则四块内存正常工作。与预期相符。通过测试。

下面的*7*，*8*测试使用动态分区算法

7. testMalloc1，2

```
testMalloc1
We allocated 2 buffers.
BUF1(size=19, addr=0x156bbc) filled with 17(*): *****************
BUF2(size=24, addr=0x156bd3) filled with 22(#): ######################
```

```
testMalloc2
We allocated 2 buffers.
BUF1(size=9, addr=0x156bbc) filled with 9(+): +++++++++
BUF2(size=19, addr=0x156bcb) filled with 19(,): ,,,,,,,,,,,,,,,,,,,
```

申请空间，写入字符，读出。与预期结果相同，测试通过。