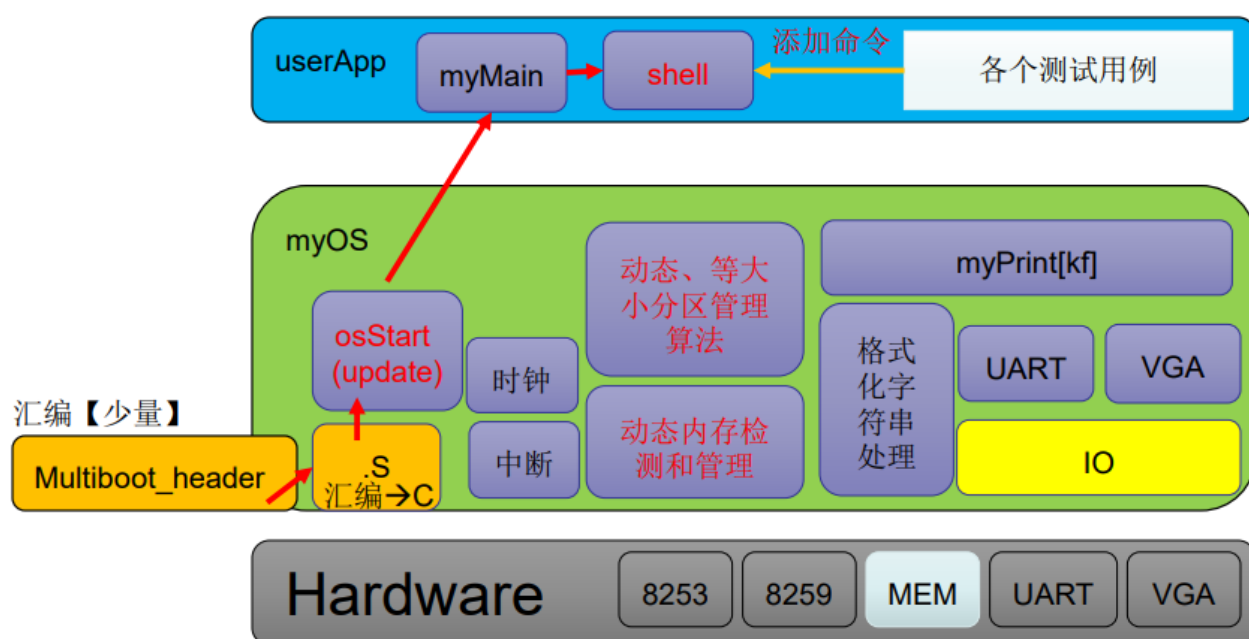


# Lab 4实验报告

## I. 程序框架

整体框架如下图：



此次实验OS和用户程序的执行流程与上一次实验基本类似，在此主要说明一些区别：

1. 文件 `myOS\osStart.c` 中，在进入用户程序之前要先调用函数 `pMemInit()`，从而建立内存管理机制。`pMemInit()` 定义在 `myOS\kernel\mem\pMemInit.c` 中，它的实现还依托于内存管理算法的实现（可选，等大小固定分区策略实现在 `myOS\kernel\mem\efPartition.c` 中，动态分区策略实现在 `myOS\kernel\mem\dPartition.c` 中。
2. 在用户程序中，启动 `shell` 之前，还需加载OS默认和用户编写的指令，这通过 `userApp\shell.c` 中的 `addNewCmd()` 函数实现，而这又基于将指令集合实现为动态的链表。

总的来说，此次实验的重点不在执行流程上，而在于实现两种内存管理策略，动态加载指令的实现则是为了方便测试内存管理功能。

## II.动态分区内存的实现

### 1. `dPartitionWalkByAddr()`：遍历并输出各个空闲分区

先定义提供的输出函数 `showdPartition()` 打印整个分配内存的信息，`dPartition` 包含如下信息：

```
//dPartition是整个动态分区内存的数据结构
typedef struct dPartition{
    unsigned long size; //整个动态分配内存的大小
    unsigned long firstFreeStart; //第一个空闲分区的地址
} dPartition;
```

而后从上面的 `firstFreeStart` 开始，沿着空闲分区链表遍历每一个EMB，用 `showEMB()` 函数输出EMB的信息。EMB包含如下信息：

```
typedef struct EMB{
    unsigned long size; //块大小(不含EMB)
    unsigned long nextStart; // if free: pointer to next block
} EMB;
```

非空闲区块也包含EMB header，但是 `nextStart` 字段没有定义。

### 2. `dPartitionInit()`：实现动态分区内存的初始化

先做个小判断，当分配内存不大于 `dPartition` 和 `EMB` 结构体的大小之和时，返回0，否则继续。然后初始化 `dPartitionInit` 结构体，它存储分配内存的起始处（参数 `start` 指定），它的 `size` 字段等于分配的总内存大小 `totalSize`。再构造第一个空闲分区，注意它的大小(`size` 字段)是从总大小中减去 `dPartitionInit` 头和 `EMB` 头的大小。最后返回分配内存的起始地址 `start`。

### 3. `dPartitionAllocFirstFit()`: 分配一个动态分区

实验中采用 `firstfit` 的动态分区算法，这约束了分配内存和释放内存操作的实现。

`dPartitionAllocFirstFit()` 中，从 `dPartition` header 中的 `firstFreeStart` 开始索引空闲分区链表；对于每一个空闲分区 `emb`，有以下三种情况：

1. `emb->size == size`: 直接从空闲链表中去掉 `emb`，更改前一区块的 `nextStart` 字段。返回 `emb` 的地址。
2. `emb->size < size`: 当前区块过小，检测下一个空闲区块。
3. `emb->size > size`: 从当前空闲区块的开头分出一个 `EMB` 和待分配的大小为 `size` 的分区，分配的分区后面是余下空闲分区的 `EMB`，更改余下空闲分区的 `EMB` 中的 `size` 字段，同时还要维持空闲链表。返回新分配分区的地址。

沿着链表执行，直到分配得到分区，或者到达链表末尾仍未得到分配，此时返回0。

### 4. `dPartitionFreeFirstFit()`: 释放一个动态分区

这部分最为复杂，因此写了比较详细的注解。

函数中，先根据地址得到待释放分区 `freeEmb` 的 `EMB` header，里面包含着该分区的大小。

然后循环遍历空闲链表，寻找待释放分区 `freeEmb` 的两个相邻分区，判断分区 `emb` 往前相邻（即 `emb` 与 `freeEmb` 相邻，`emb` 在低地址）的条件为：`(unsigned long)emb + EMB_size + emb->size == start - EMB_size`，类似，往后相邻的条件为：`(unsigned long)emb == start + freeEmb->size`。分别存储两个相邻分区。

最后根据遍历结果分4中情况处理。

完整实现如下：

```
unsigned long dPartitionFreeFirstFit(unsigned long dp, unsigned
long start){
    dPartition* dpPtr = (dPartition*)dp;
    EMB* freeEmb = (EMB*)(start - EMB_size); //要释放的分区
    if (dp + dPartition_size + EMB_size > start || start + freeEmb-
>size > dp + dpPtr->size) {
        myPrintk(0x5, "The block is out of the space.");
        return 0;
    }
    //查看相邻分区是否为空闲分区
    EMB* adjEmb[2] = { (EMB*)0, (EMB*)0 }; //前、后相邻分区:不为空时表示
free
    //后相邻分区free时,索引后相邻分区在free分区链中前驱的nextstart字段
    unsigned long* prePostNext = &dpPtr->firstFreeStart; //索引free分
区链中emb分区前一分区的nextstart字段
    EMB* emb = (EMB*)dpPtr->firstFreeStart;
    while ((unsigned long)emb) {
        //检查前相邻的分区
        if ((unsigned long)emb + EMB_size + emb->size == start -
EMB_size) {
            adjEmb[0] = emb; //前相邻分区
            if (adjEmb[1]) break;
        }
        //检查后相邻的分区
        if ((unsigned long)emb == start + freeEmb->size) {
            adjEmb[1] = emb; //后相邻分区
            if (adjEmb[0]) break;
        }
        if (adjEmb[1] == (EMB*)0) prePostNext = &emb->nextStart;
        emb = (EMB*)emb->nextStart;
        //找到free的后相邻分区后,prePostNext索引后相邻分区在free分区链中前驱
的nextstart字段,从而保持不变
    }
    if (adjEmb[0] && adjEmb[1]) { //前、后相邻都是空闲分区
```

```

        adjEmb[0]->size += 2 * EMB_size + freeEmb->size +
adjEmb[1]->size;
        *prePostNext = adjEmb[1]->nextStart;
    }
    else if (adjEmb[0] && !adjEmb[1]) //前相邻分区free,后相邻分区not
free
        adjEmb[0]->size += EMB_size + freeEmb->size;
    else if (!adjEmb[0] && adjEmb[1]) { //前相邻分区not free,后相邻分区
free
        *prePostNext = start - EMB_size;
        freeEmb->size += EMB_size + adjEmb[1]->size;
        freeEmb->nextStart = adjEmb[1]->nextStart;
    }
    else { //前、后相邻都不是空闲分区
        freeEmb->nextStart = dpPtr->firstFreeStart;
        dpPtr->firstFreeStart = start - EMB_size;
    }
    return 1;
}

```

### III. 等大小固定分区内存的实现

#### 1. `eFPartitionTotalSize()`: 计算占用空间的实际大小

需要注意内存分配时的对齐，实验中采用4字节对齐，设置全局变量`align=4`，以及用全局变量实现`eFPartition_size`与`EEB_size`的对齐：

```

unsigned long efpAlign = ((eFPartition_size + align - 1) / align) *
align;
unsigned long eebAlign = ((EEB_size + align - 1) / align) * align;

```

计算`perSize`的对齐，返回`efpAlign + n * (eebAlign + perSize)`。

## 2. `eFPartitionInit()`: 按照按等大小固定分区策略初始化内存

类似于动态分区策略中的`dPartitionInit()`函数，先初始化`eFPartition`结构体，然后构造空闲内存区块。不过不同的是，等大小固定分区策略在初始时就包含`n`个空闲分区，所有需要初始化空闲链表将所有空闲分区连接起来。

## 3. `eFPartitionAlloc()`: 分配一个等大小固定分区

实现比较简单，检查是否还有空闲区块，如果有，进行分配，维护链表，返回分配得到的分区的地址；否则直接返回0。代码如下：

```
unsigned long eFPartitionAlloc(unsigned long EFPHandler){
    unsigned long freeEEB = ((eFPartition*)EFPHandler)->firstFree;
    if (freeEEB == 0) { //无空闲区块
        myPrintk(0x5, "ERROR:No free block.");
        return 0;
    }
    ((eFPartition*)EFPHandler)->firstFree = ((EEB*)freeEEB)-
>next_start; //维护空闲block链表
    return freeEEB + eebAlign;
}
```

## 4. `eFPartitionFree()`: 释放一个分区

先检查释放的分区是否完全在当前分配内存空间中，确认无误后再释放分区，即将分区插入空闲链表的开头。

```

unsigned long eFPartitionFree(unsigned long EFPHandler,unsigned
long mbStart){
    eFPartition* efpPtr = (eFPartition*)EFPHandler;
    if ((mbStart - eebAlign - EFPHandler - efpAlign) % (eebAlign +
efpPtr->perSize)) {//不合适的地址
        myPrintk(0x5, "Address is not the start of a block.");
        return 0;
    }
    //维护空闲block链表
    EEB eeb = { efpPtr->firstFree };
    efpPtr->firstFree = mbStart - eebAlign;
    *((EEB*)efpPtr->firstFree) = eeb;
    return efpPtr->firstFree;
}

```

## IV. 其它部分的实现

### 1. 内存大小检测的实现

`void memTest(unsigned long start, unsigned long grainSize)` 函数实现内存大小的检测，分别将可用的内存的起始地址和大小写入全局变量 `pMemStart` 和 `pMemSize`。

这部分根据提供的检测算法编写即可。需要注意以下几点：

1. 起始地址与步长的额外判断，这两个量都有自己的下界，如下：

```

if (start < 0x100000) start = 0x100000;
if (grainSize < 4) grainSize = 4;

```

2. 如果读写grain的头2个字节或尾2个字节时每次只读写一个字节，那么注意不能使用char类型，而应用unsigned char替代。这是因为写入0xAA后到char类型变量后，读取变量与0xAA比较，它会自动转化为int类型的值0xFFFF\_FF56，导致不等而出错。

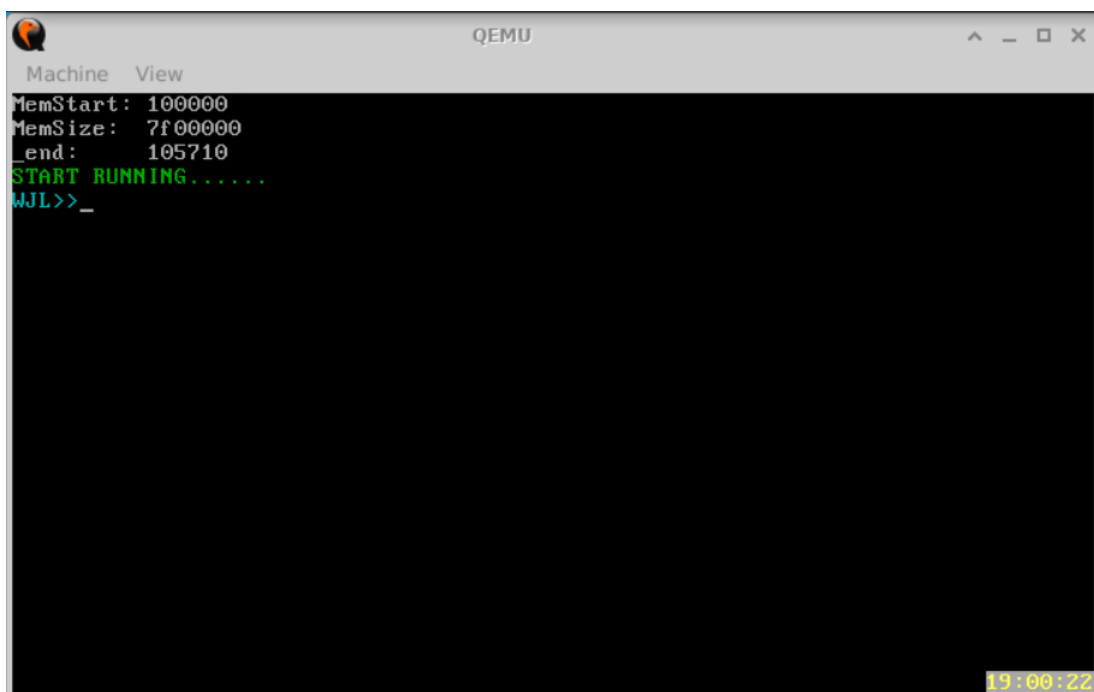
## 2. 动态增加指令的实现

动态增加指令的实现主要得益于在cmd结构体中增加cmd\*类型的指针，进而可以构成动态的cmd链表。我们需要编写的是增加指令的函数addNewCmd()。

首先根据参数创建cmd，填写各个字段，注意命令名和description[101]字段只能通过字符串赋值来实现，此次实验借助于提供的strcpy()函数实现（需要适当修改以复制字符串末尾的\0）。然后维护链表结构，将新创建的cmd插入表头，修改相关的指针。

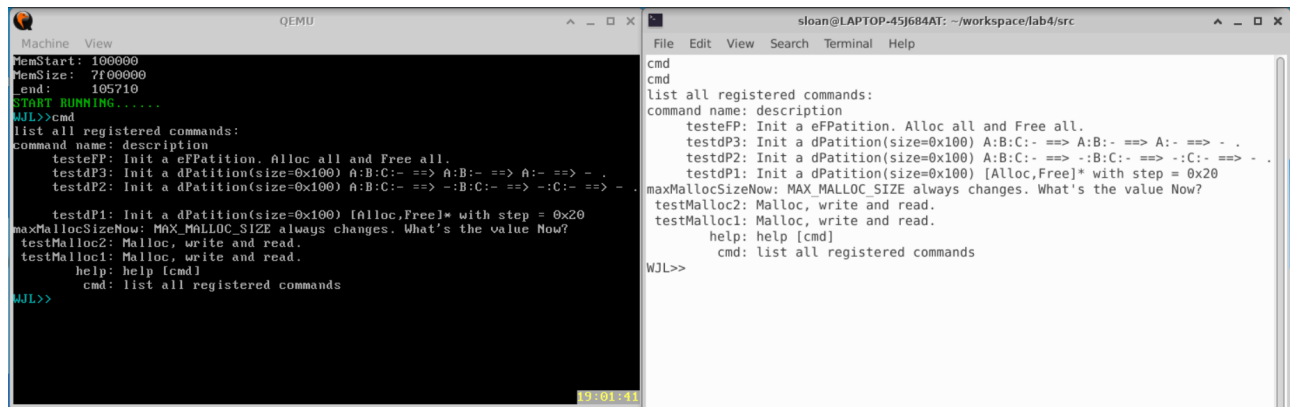
## V. 实验结果

编译并运行程序后，得到如下结果：



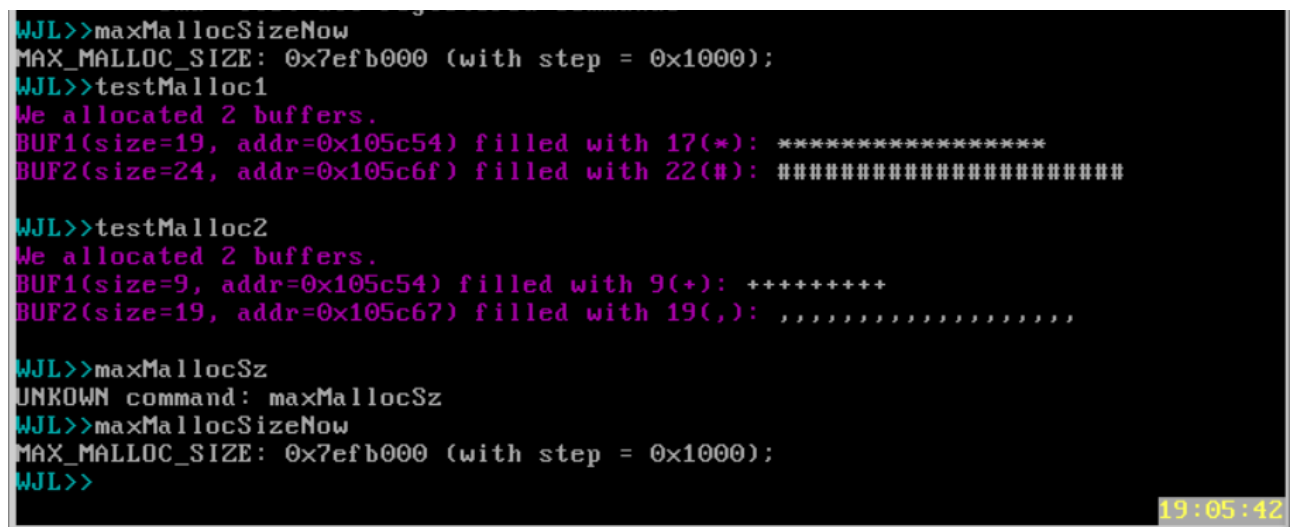
输入指令重定向串口到伪终端，输入cmd指令列举所有指令，得到如下结果：





可以看到，测试文件 `userApp\memTestCase.c` 中增加的7个指令都存在。

然后输入 `maxMallocSizeNow` 指令，输出结果正是 `MemSize`（开始时有显示，取值为 `0x7f0_0000`）减去 `_end-MemStart`（两个量开头均显示）后对 `0x1000` 取整的结果。先后输入 `testMalloc1` 和 `testMalloc2`，如下输出符合预期。再一次执行 `maxMallocSizeNow` 指令，得到结果前一次相同，说明分配的内存都得到了回收，没有内存泄漏。如下：

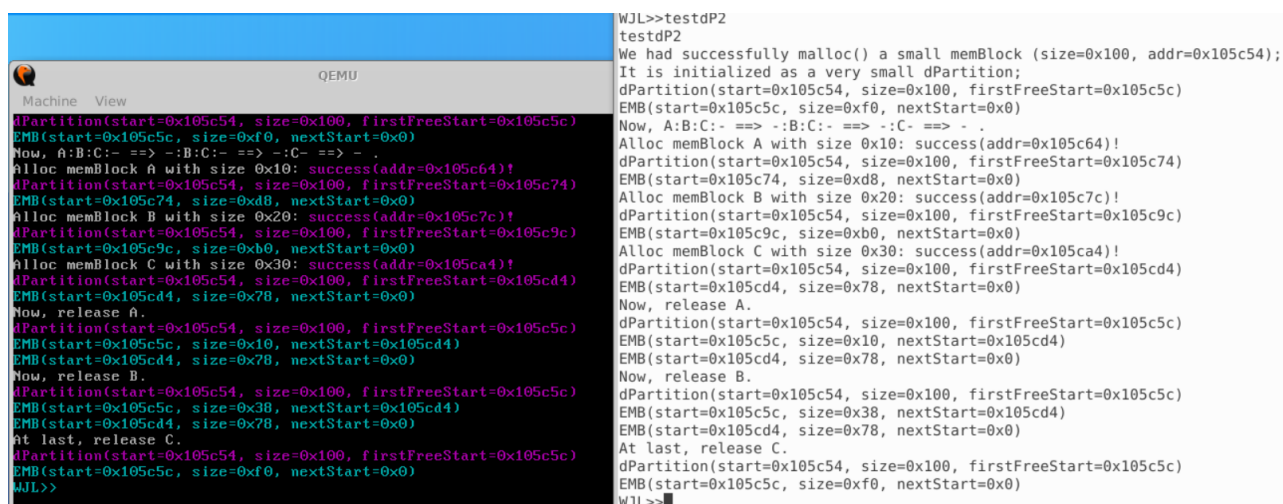


再执行 `testdP1` 指令测试动态分区算法，结果如下图。分配 `0x100` 字节时失败，与预期相符；初始时 `EMB.size=0xf0` 不同于助教提供的结果，这是因为我将 `size` 字段设计为不包含 `EMB` 结构体部分的大小。

```
WJL>>testdP1
We had successfully malloc() a small memBlock (size=0x100, addr=0x105c54);
It is initialized as a very small dPartition:
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105c5c)
EMB(start=0x105c5c, size=0xf0, nextStart=0x0)
Alloc a memBlock with size 0x10, success(addr=0x105c64)!.....Relaesed;
Alloc a memBlock with size 0x20, success(addr=0x105c64)!.....Relaesed;
Alloc a memBlock with size 0x40, success(addr=0x105c64)!.....Relaesed;
Alloc a memBlock with size 0x80, success(addr=0x105c64)!.....Relaesed;
Alloc a memBlock with size 0x100, failed!
Now, converse the sequence.
Alloc a memBlock with size 0x100, failed!
Alloc a memBlock with size 0x80, success(addr=0x105c64)!.....Relaesed;
Alloc a memBlock with size 0x40, success(addr=0x105c64)!.....Relaesed;
Alloc a memBlock with size 0x20, success(addr=0x105c64)!.....Relaesed;
Alloc a memBlock with size 0x10, success(addr=0x105c64)!.....Relaesed;
WJL>>_
```

19:06:46

执行 `testdP2` 指令，结果如下图。仔细对照实验文档中的图示，可以看到其处理空闲分区合并时的正确性。



```
WJL>>testdP2
testdP2
We had successfully malloc() a small memBlock (size=0x100, addr=0x105c54);
It is initialized as a very small dPartition:
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105c5c)
EMB(start=0x105c5c, size=0xf0, nextStart=0x0)
Now, A:B:C:- ==> -:B:C:- ==> -:C:- ==> - .
Alloc memBlock A with size 0x10: success(addr=0x105c64)!
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105c74)
EMB(start=0x105c74, size=0xd8, nextStart=0x0)
Alloc memBlock B with size 0x20: success(addr=0x105c7c)!
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105c9c)
EMB(start=0x105c9c, size=0xb0, nextStart=0x0)
Alloc memBlock C with size 0x30: success(addr=0x105ca4)!
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105cd4)
EMB(start=0x105cd4, size=0x78, nextStart=0x0)
Now, release A.
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105c5c)
EMB(start=0x105c5c, size=0x10, nextStart=0x105cd4)
EMB(start=0x105cd4, size=0x78, nextStart=0x0)
Now, release B.
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105c5c)
EMB(start=0x105c5c, size=0x38, nextStart=0x105cd4)
EMB(start=0x105cd4, size=0x78, nextStart=0x0)
At last, release C.
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105c5c)
EMB(start=0x105c5c, size=0xf0, nextStart=0x0)
WJL>>
```

执行 `testdP3` 指令，得到如下结果。

```
QEMU
Machine View
We had successfully malloc() a small memBlock (size=0x100, addr=0x105c54);
It is initialized as a very small dPartition:
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105c5c)
EMB(start=0x105c5c, size=0xf0, nextStart=0x0)
Now, A:B:C:- ==> -:B:C:- ==> -:C- ==> - .
Alloc memBlock A with size 0x10: success(addr=0x105c64)!
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105c74)
EMB(start=0x105c74, size=0xd8, nextStart=0x0)
Alloc memBlock B with size 0x20: success(addr=0x105c7c)!
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105c9c)
EMB(start=0x105c9c, size=0xb0, nextStart=0x0)
Alloc memBlock C with size 0x30: success(addr=0x105ca4)!
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105cd4)
EMB(start=0x105cd4, size=0x78, nextStart=0x0)
At last, release C.
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105c9c)
EMB(start=0x105c9c, size=0xb0, nextStart=0x0)
Now, release B.
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105c74)
EMB(start=0x105c74, size=0xd8, nextStart=0x0)
Now, release A.
dPartition(start=0x105c54, size=0x100, firstFreeStart=0x105c5c)
EMB(start=0x105c5c, size=0xf0, nextStart=0x0)
WJL>>
```

## VI. 思考题

### 1. malloc接口的实现:

- malloc() 中调用 dPartitionAlloc() 函数;
- dPartitionAlloc() 中调用 dPartitionAllocFirstFit() 函数。
- 函数 dPartitionAllocFirstFit() 根据动态分区策略实现了内存分配的算法。

### 2. testdp3运行结果详解（结果见V.实验结果中的图示）:

- a. 整个分配区块的起始地址为0x105c54，总大小为0x100。初始时只有一个空闲分区，对应EMB地址为0x105c54+0x8=0x105c5c，大小size=0x100-0x8-0x8=0xf0。
- b. 分配大小为0x10的分区A后：仍只有一个空闲区块，由于增加了一个EMB和分区A，空闲区块的EMB地址增加0x8+0x10，变为0x105c74；相应地大小减小为0xd8。

- c. 分配大小为 **0x20** 的分区B：空闲分区的EMB地址增大为 **0x105c9c**，区块大小减小为 **0xb0**。
- d. 分配大小为 **0x30** 的分区C：空闲分区的EMB地址增大为 **0x105cd4**，区块大小减小为 **0x78**。
- e. 释放分区C，回到c.对应状态；
- f. 释放分区B，回到b.对应状态；
- g. 释放分区A，回到初始状态。