

lab4 实验报告

李远航

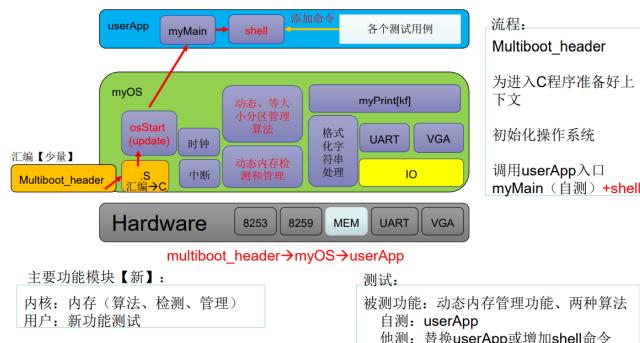
PB20000137

一、实验内容

- 【必须】内存检测，确定动态内存的范围
- 【必须】提供动态分区管理机制 dPartition
- 【必须】提供等大小固定分区管理机制 ePartition
- 【必须】使用动态分区管理机制来管理所有动态内存
- 【必须】提供 kmalloc/kfree 和 malloc/free 两套接口，分别提供给内核和用户
- 【可选】kmalloc/kfree、malloc/free 相互隔离
- 【必须】自测
 - shell 改用 malloc/free 来动态注册新命令（将涉及部分字符串处理）
 - 自编测试用例，并添加到 shell 命令列表中，执行新增的 shell 命令

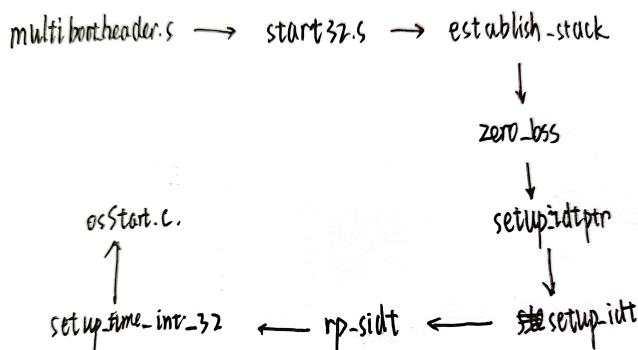
二、实验原理

- 软件的架构(框图)



三、实验过程

1. 代码的主流程



2. 主要功能模块及其实现

(1) 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

具体代码：

```
void memTest(unsigned long start, unsigned long grainSize)
{
    if (start < 0x100000)
        start = 0x100000;
    if (grainSize < 2)
        grainSize = 2;

    pMemSize = 0;
    pMemStart = start;

    unsigned long addr = start;
    unsigned short *addr_head, *addr_tail;
    unsigned short temp_data;
    unsigned short test1 = 0x55AA;
    unsigned short test2 = 0xAA55;

    while (1)
    {
        addr_head = (unsigned short *)addr;
        addr_tail = (unsigned short *)(addr + grainSize - 2);

        temp_data = *addr_head;
        *addr_head = test1;
        if (*addr_head != test1)
        {
            *addr_head = temp_data;
            break;
        }
        *addr_head = test2;
        if (*addr_head != test2)
        {
            *addr_head = temp_data;
            break;
        }
        *addr_head = temp_data;

        temp_data = *addr_tail;
        *addr_tail = test1;
```

```

if (*addr_tail != test1)
{
    *addr_tail = temp_data;
    break;
}
*addr_tail = test2;
if (*addr_tail != test2)
{
    *addr_tail = temp_data;
    break;
}
*addr_tail = temp_data;

pMemSize += grainSize;
addr += grainSize;
}
myPrintk(0x7, "MemStart: %x \n", pMemStart);
myPrintk(0x7, "MemSize: %x \n", pMemSize);
}

```

(2)等大小分区管理算法

- `unsigned long eFPartitionTotalSize(unsigned long perSize, unsigned long n)` 计算占用空间的实际大小，并将这个结果返回 具体实现(使用 4 字节对齐，每一块的大小为对齐的 perSize+EEB 本身的大小，总和加上 eFPartition 的大小):

```

unsigned long eFPartitionTotalSize(unsigned long perSize, unsigned long n)
{
    return (align4(perSize) + sizeof(EEB)) * n + sizeof(eFPartition);
}

```

- `unsigned long eFPartitionInit(unsigned long start, unsigned long perSize, unsigned long n)` 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 具体实现(通过一个 while 循环向后逐步初始化每一块 EEB):

```

unsigned long eFPartitionInit(unsigned long start, unsigned long
perSize, unsigned long n)
{

```

```

perSize = align4(perSize) + sizeof(EEB);
eFPartition *handler = (eFPartition *)start;
handler->totalN = n;
handler->firstFree = start + sizeof(eFPartition);
handler->perSize = perSize;

unsigned long addr = start + sizeof(eFPartition);
EEB *block;
while (n--)
{
    block = (EEB *)addr;
    addr += perSize;
    if (n == 0)
        block->next_start = 0;
    else
        block->next_start = addr;
}
return start;
}

```

- `void eFPartitionWalkByAddr(unsigned long efpHandler)` 方便查看和调试

- 1、打印 eFPartiton 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`）具体实现(循环遍历内存块):

```

void eFPartitionWalkByAddr(unsigned long efpHandler)
{
    eFPartition *handler = (eFPartition *)efpHandler;
    showeFPartition(handler);
    unsigned long addr = handler->firstFree;
    EEB *block;
    while (addr)
    {
        block = (EEB *)addr;
        showEEB(block);
        addr = block->next_start;
    }
}

```

- `unsigned long eFPartitionAlloc(unsigned long EFPHandler)`

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

具体实现(每次从空闲的链表中取出第一块返回，更新空闲链表的头):

```
unsigned long eFPartitionAlloc(unsigned long EFPHandler)
{
    eFPartition *handler = (eFPartition *)EFPHandler;
    if (handler->firstFree == 0)
        return 0;
    EEB *block = (EEB *)handler->firstFree;
    handler->firstFree = block->next_start;
    return (unsigned long)(block + sizeof(EEB));
}
```

- `unsigned long eFPartitionAlloc(unsigned long EFPHandler, unsigned long mbStart)`

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。 具体实现(将待释放的块加入到空闲链表的头):

```
unsigned long eFPartitionFree(unsigned long EFPHandler, unsigned long mbStart)
{
    eFPartition *handler = (eFPartition *)EFPHandler;
    EEB *free_block = (EEB *)(mbStart - sizeof(EEB));
    free_block->next_start = handler->firstFree;
    handler->firstFree = mbStart - sizeof(EEB);
    return 1;
}
```

(3) 动态分区管理算法

- `unsigned long dPartitionInit(unsigned long start, unsigned long totalSize)`

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)。 具体实现(将整块内存区域初始化为一块 `EMB`):

```
unsigned long dPartitionInit(unsigned long start, unsigned long totalSize)
{
    if (totalSize <= sizeof(dPartition) + sizeof(EMB))
        return 0;
    dPartition *handler = (dPartition *)start;
```

```

    handler->size = totalSize;
    handler->firstFreeStart = start + sizeof(dPartition);

    EMB *block = (EMB *)handler->firstFreeStart;
    block->size = totalSize - sizeof(dPartition) - sizeof(EMB);
    block->nextStart = 0;

    return start;
}

```

- `void dPartitionWalkByAddr(unsigned long dp)`

遍历输出 EMB

```

void dPartitionWalkByAddr(unsigned long dp)
{
    dPartition *handler = (dPartition *)dp;
    showdPartition(handler);

    unsigned long now_addr = handler->firstFreeStart;
    EMB *temp;
    while (now_addr)
    {
        temp = (EMB *)now_addr;
        showEMB(temp);
        now_addr = temp->nextStart;
    }
}

```

- `unsigned long dPartitionAllocFirstFit(unsigned long dp, unsigned long size)`

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

具体实现:

```

unsigned long dPartitionAllocFirstFit(unsigned long dp, unsigned long size)
{
    dPartition *handler = (dPartition *)dp;
    if (handler->firstFreeStart == 0)
        return 0;
    size = align8(size);
    unsigned long now_addr = handler->firstFreeStart;
    unsigned long pre_addr = 0;
    EMB *block_now, *block_pre;

```

```

    while (now_addr)
    {
        block_now = (EMB *)now_addr;
        block_pre = (EMB *)pre_addr;
        if (block_now->size >= size)// 寻找足够大小的块(firstfit)
        {
            //直接整个块分配
            if (block_now->size < sizeof(EMB) + size)
            {
                if (pre_addr == 0)
                    handler->firstFreeStart = block_now->nextStart;
                else
                    block_pre->nextStart = block_now->nextStart;
                return now_addr + sizeof(EMB);
            }
            else
                //切分该空闲块
            {
                unsigned long new_block_addr = now_addr + sizeof(EMB) +
                size;
                EMB *block_new = (EMB *)new_block_addr;
                block_new->size = block_now->size - size - sizeof(EMB);
                block_new->nextStart = block_now->nextStart;
                block_now->size -= block_new->size + sizeof(EMB);
                if (pre_addr == 0)
                    handler->firstFreeStart = new_block_addr;
                else
                    block_pre->nextStart = new_block_addr;
                return now_addr + sizeof(EMB);
            }
        }
        pre_addr = now_addr;
        now_addr = block_now->nextStart;
    }
    return 0;
}

```

- `unsigned long dPartitionFreeFirstFit(unsigned long dp, unsigned long start)`

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

```

unsigned long dPartitionFreeFirstFit(unsigned long dp, unsigned long
start)

```

```
{  
    dPartition *handler = (dPartition *)dp;  
    start -= sizeof(EMB);  
    // myPrintk(0x7, "\nstart: %x\n", start);  
    if (start < dp + sizeof(dPartition) || start > dp + handler->size)  
        return 0;  
  
    unsigned long now_addr = handler->firstFreeStart;  
    unsigned long pre_addr = 0;  
    unsigned long next_addr = 0;  
    EMB *block;  
    //找到插入位置  
    while (now_addr)  
    {  
        block = (EMB *)now_addr;  
        if (now_addr < start)  
            pre_addr = now_addr;  
        else  
        {  
            next_addr = now_addr;  
            break;  
        }  
        now_addr = block->nextStart;  
    }  
  
    block = (EMB *)start;  
    if (next_addr != 0)  
    {  
        //判断和后方内存块合并  
        if (start + block->size + sizeof(EMB) == next_addr)  
        {  
            EMB *next_block = (EMB *)next_addr;  
            block->size += next_block->size + sizeof(EMB);  
            block->nextStart = next_block->nextStart;  
        }  
        else  
            block->nextStart = next_addr;  
    }  
    else  
        block->nextStart = 0;  
  
    if (pre_addr != 0)  
    {  
        //判断和前方内存块合并  
        EMB *pre_block = (EMB *)pre_addr;  
        if (start == pre_addr + pre_block->size + sizeof(EMB))  
        {  
            pre_block->size += block->size + sizeof(EMB);  
            pre_block->nextStart = block->nextStart;  
        }  
        else  
            pre_block->nextStart = start;  
    }  
}
```

```

        handler->firstFreeStart = start;
    return 1;
}

```

(4) 区分用户和内核的内存

将获取的可用的内存分块，一块用于内核，一块用于用户，句柄分别为 `kMemHandler` 和 `uMemHandler`, `kmalloc()` 和 `kfree()` 使用第一个句柄, `malloc()` 和 `free()` 使用后者，需要注意不可以直接平分可以使用的内存，因为管理区分内核和用户区需要空间

(5) shell 的变化

新定义 `update` 函数用于新指令字符串部分赋值，新加入 `clear` 用于清空 vga 显示

```

void update(unsigned char *s, unsigned char *p)
{
    int i = 0;
    while (1)
    {
        if (*(s + i) == '\0')
            break;
        *(p + i) = *(s + i);
        i++;
    }
    *(p + i) = '\0';
}

void addNewCmd(unsigned char *command,
               int (*func)(int argc, unsigned char **argv),
               void (*help_func)(void),
               unsigned char *description)
{
    struct cmd *newcmd = (struct cmd *)malloc(sizeof(struct cmd));
    newcmd->func = func;
    newcmd->help_func = help_func;
    update(command, newcmd->cmd);
    update(description, newcmd->description);

    newcmd->nextCmd = ourCmds;
    ourCmds = newcmd;
}

```

3. 源代码组织说明

- 项目结构

```

├── 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
source2img.sh
userApp
├── main.c
└── Makefile
```

```
└── memTestCase.c
└── memTestCase.h
└── shell.c
└── shell.h
```

- Makefile 组织

```
.
├── MULTI_BOOT_HEADER
│   └── output/multibootHeader/multibootHeader.o
└── OS_OBJS
    ├── MYOS_OBJS
    │   ├── output/myOS/osStart.o
    │   ├── output/myOS/start32.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/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
```

4. 代码布局说明

Section	Offset (Base = 1M)
.multiboot_header	0
.text	8
.data	16

Section	Offset (Base = 1M)
.bss	16
_end	16

四、编译运行过程

直接运行脚本文件

```
./source2img.sh
```

根据提示重定向串口输入

脚本的执行:

- 编译各个文件，生成相应的 .o 目标文件
- 根据链接描述文件，将各 .o 目标文件进行链接，生成 myOS.elf 文件
- 使用 qemu，调用上一步生成的文件，进行模拟

五、运行结果

- testMalloc

输出结果应为指定数量的符号

```
voyage@qemu:~$ testMalloc
we allocated 2 buffers.
BUF1(size=19, addr=0xb59e48) filled with 17(*): ****
BUF2(size=24, addr=0xb59e60) filled with 19(*): ,,,,,,,,,,,,
voyage@qemu:~$
```

- testdP1

除 0x100 不可以正常分配，其余均可以成功分配，且每次分配的内存起始地址相同

```

=0x55acb98, size=0x2a53460, nextStart=0x0)
n()
o=0
n(Machine View
voyage@qemu:~$ testdP1
=0 We had successfully malloc() a small memBlock (size=0x100, addr=0xb59e48):
dPartition(start=0xb59e48, size=0x100, firstFreeStart=0xb59e50)
=0 Now, converse the sequence.
alloc a memBlock with size 0x10, success(addr=0xb59e50)!.....Released;
alloc a memBlock with size 0x20, success(addr=0xb59e50)!.....Released;
alloc a memBlock with size 0x40, success(addr=0xb59e50)!.....Released;
alloc a memBlock with size 0x80, success(addr=0xb59e50)!.....Released;
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=0xb59e50)!.....Released;
alloc a memBlock with size 0x40, success(addr=0xb59e50)!.....Released;
alloc a memBlock with size 0x20, success(addr=0xb59e50)!.....Released;
alloc a memBlock with size 0x10, success(addr=0xb59e50)!.....Released;
voyage@qemu:~$ 
em
em
em
em
Unknown interrupt! 19:00:14
memBLOCK with size 0x20, success(addr=0xb59e50)!.....Released;
memBlock with size 0x10, success(addr=0xb59e50)!.....Released;

```

- testdP2

分配时只有一个 EMB 块，在释放时，先是 2 个 EMB 块，释放完成后，又变回一块，正确完成合并，且运行前后 dPartition 相关参数不变

```

d as a very small dPartition:
=0
e5
Machine View
dPartition(start=0xb59e48, size=0x100, firstFreeStart=0xb59e50)
=0 Now, A:B:C:- ==> -B:C:- ==> -C: - ==> -
=0 Alloc memBlock A with size 0x10: success(addr=0xb59e50)!
e5dPartition(start=0xb59e48, size=0x100, firstFreeStart=0xb59e68)
=0 EMB(start=0xb59e60, size=0x08, nextStart=0x0)
=0 Alloc memBlock B with size 0x20: success(addr=0xb59e70)!
```

- testdP3

分配和释放时都只有一个 EMB 块，正确完成合并，且运行前后 dPartition 相关参数不变

```

tdP3
ll
QEMU
d
Machine View
voyage@qemu:~$ testdP3
=0 We had successfully malloc() a small memBlock (size=0x100, addr=0xb59e48):
dPartition(start=0xb59e48, size=0x100, firstFreeStart=0xb59e50)
=0 Now, A:B:C:- ==> A:- ==> A:- ==> -
=0 Alloc memBlock A with size 0x10: success(addr=0xb59e50)!
dPartition(start=0xb59e48, size=0x100, firstFreeStart=0xb59e68)
e5dPartition(start=0xb59e60, size=0x08, nextStart=0x0)
=0 Alloc memBlock B with size 0x20: success(addr=0xb59e70)!
```

- testeFP

开头内存空间大小为 $(32+4)*4+12=156$ 成功分配 ABCD，EEB 块减少，分配 E 失败，逐步释放 ABCD，EEB 块增多，且运行前后 eFPartition 相关参数不变

```

voyage@qemu$ testeFP
X:0x2b59e48:156
We had successfully malloc() a small memBlock (size=0x9c, addr=0x2b59e48);
It is initialized as a very small ePartition;
ePartition(start=0x2b59e48, totalN=0x4, perSize=0x24, firstFree=0x2b59e54)
EEBc(start=0x2b59e54, next=0x2b59e78)
EEBc(start=0x2b59e78, next=0x2b59e9c)
EEBc(start=0x2b59e9c, next=0x2b59ec0)
EEBc(start=0x2b59ec0, next=0x2b59e80)
Alloc memBlock A, start = 0x2b59e64; 0xaaaaaaaa
ePartition(start=0x2b59e80, totalN=0x4, perSize=0x24, firstFree=0x2b59e78)
EEBc(start=0x2b59e78, next=0x2b59e9c)
EEBc(start=0x2b59e9c, next=0x2b59ec0)
EEBc(start=0x2b59ec0, next=0x2b59e80)
Alloc memBlock B, start = 0x2b59e88; 0xbbbbbbbb
ePartition(start=0x2b59e48, totalN=0x4, perSize=0x24, firstFree=0x2b59e9c)
EEBc(start=0x2b59e9c, next=0x2b59ec0)
EEBc(start=0x2b59ec0, next=0x0)
Alloc memBlock C, start = 0x2b59eac; 0xcccccccc
ePartition(start=0x2b59e48, totalN=0x4, perSize=0x24, firstFree=0x2b59ec0)
EEBc(start=0x2b59e0, next=0x2b59e80)
Alloc memBlock D, start = 0x2b59ed8; 0xddddddd
ePartition(start=0x2b59e48, totalN=0x4, perSize=0x24, firstFree=0x0)
Alloc memBlock E, failed!
ePartition(start=0x2b59e48, totalN=0x4, perSize=0x24, firstFree=0x0)
Now, release A.
ePartition(start=0x2b59e48, totalN=0x4, perSize=0x24, firstFree=0x2b59e60)
EEBc(start=0x2b59e60, next=0x0)
Now, release B.
ePartition(start=0x2b59e48, totalN=0x4, perSize=0x24, firstFree=0x2b59e84)
EEBc(start=0x2b59e84, next=0x2b59e60)
EEBc(start=0x2b59e60, next=0x0)
Now, release C.
ePartition(start=0x2b59e48, totalN=0x4, perSize=0x24, firstFree=0x2b59ea8)
EEBc(start=0x2b59ea8, next=0x2b59e84)
EEBc(start=0x2b59e84, next=0x2b59e60)
EEBc(start=0x2b59e60, next=0x0)
Now, release D.
ePartition(start=0x2b59e48, totalN=0x4, perSize=0x24, firstFree=0x2b59ecc)
EEBc(start=0x2b59ecc, next=0x2b59ea8)
EEBc(start=0x2b59ea8, next=0x2b59e84)
EEBc(start=0x2b59e84, next=0x2b59e60)
EEBc(start=0x2b59e60, next=0x0)
voyage@qemu$ |

```

- maxMallocSize

输出可以开辟的最大空间

```

voyage@qemu$ maxMallocSizeNow
MAX_MALLOC_SIZE: 0x3f7d000 (with step = 0x1000);
voyage@qemu$ |

```

- KMEM 和 UMEM 的测试

上述测试样例均使用 UMEM 进行操作，本样例使用 KMEM 进行操作，使用上述 testdP3，但是用 kmalloc 来开辟空间，之后再检测内核区可用内存大小，输出结果除地址外应当和上基本一致

```

Ubuntu-20.04
maxMallocNow: MAX_MALLOC_SIZE always changes. What's the value Now?
testMalloc2: Malloc, write and read.
testMalloc1: Malloc, write and read.
    clear: clear the vga screen
    help: help [cmd]
    cmd: list all registered commands
voyage@qemu$ testKMEM_dp
We had successfully kmalloc() a small memBlock (size=0x100, addr=0x1062a0);
It is initialized as a very small dPartition;
dPartition(start=0x1062a0, size=0x100, firstFreeStart=0x1062a8)
EMB(start=0x1062a8, size=0xf0, nextStart=0x0)
Now, A:B:C:- ==> A:B:- ==> A:- ==> - .
Alloc memBlock A with size 0x10: success(addr=0x1062b0)!
dPartition(start=0x1062a0, size=0x100, firstFreeStart=0x1062c0)
EMB(start=0x1062c0, size=0xd8, nextStart=0x0)
Alloc memBlock B with size 0x20: success(addr=0x1062c8)!
dPartition(start=0x1062a0, size=0xb0, firstFreeStart=0x1062e8)
EMB(start=0x1062e8, size=0x0b0, nextStart=0x0)
Alloc memBlock C with size 0x30: success(addr=0x1062f0)!)
dPartition(start=0x1062a0, size=0x100, firstFreeStart=0x106320)
EMB(start=0x106320, size=0x78, nextStart=0x0)
Now, release C.
dPartition(start=0x1062a0, size=0x100, firstFreeStart=0x1062e8)
EMB(start=0x1062e8, size=0xb0, nextStart=0x0)
Now, release B.
dPartition(start=0x1062a0, size=0x100, firstFreeStart=0x1062c0)
EMB(start=0x1062c0, size=0xd8, nextStart=0x0)
At last, release A.
dPartition(start=0x1062a0, size=0x100, firstFreeStart=0x1062a8)
EMB(start=0x1062a8, size=0xf0, nextStart=0x0)
voyage@qemu$ testKMEM_max
MAX_KMALLOC_SIZE: 0x2a54000 (with step = 0x1000);
voyage@qemu$ |

```

六、实验收获

- 对操作系统内存的管理有了新的认识
- 学习了等大小固定分区管理机制和动态分区管理机制
- 学习使用空闲链表和 firstfit 分配算法
- 区分了提供给内核和用户的接口
- 学习了 Makefile 的组织

- 遇到的问题：
 - 返回的地址应当是真正可以使用的内存的地址，应当是略过EEB/EMB之后的
 - 动态分区管理释放时要考虑内存链表的合并