OS_Lab_04 Report

PB21111686_赵卓

实验题目

- Memory Management

实验目的

- 在前三个实验的基础上，给操作系统添加内存管理机制。

实验内容

- 补全shell.c，pMemInit.c，dPartition.c，eFPartition.c文件缺失的内容。

实验模块实现

- shell.c
  - 由于本次实验测试需要多个指令，因此在shell.c中增加了一个addNewCmd函数，用于向指令链表中增加新的指令。
  - 我们调用自己编写的malloc函数申请指令的动态空间，然后用传入的参数将指令对应的内容初始化，并且链入指令链表即可，实现如下：

```c
void addNewCmd(unsigned char *cmd_,
               int (*func)(int argc, unsigned char **argv),
               void (*help_func)(void),
               unsigned char *description)
{
  // 创建指令并填入内容
  cmd *new = (cmd *)malloc(cmd_size);
  new->func = func;
  new->help_func = help_func;
  strcpy(cmd_, new->cmd);
  strcpy(description, new->description);

  // 维护指令链表
  new->nextCmd = ourCmds;
  ourCmds = new;
}
```

- pMemInit.c
  - 在os启动之后，会执行pMemInit()函数建立内存管理机制，目的是检测可用的动态内存，并且初始化动态分配的句柄和第一个EMB。
  - 首先进行内存检测，内存检测是为了检测可以写入写出的内存。根据讲义，我们只需将固定步长的内存起点和终点两个字节位置，写入0xAA55和0x55AA并读出看看内容是否变化，然后将原来内容还原即可，实现如下：

```
void memTest(unsigned long start, unsigned long grainSize)
{
  unsigned long addr = start;
  unsigned short data;
  unsigned short *head, *tail;
  unsigned short test_1 = 0xAA55;
  unsigned short test_2 = 0x55AA;
  int sign = 0; // 检测失败标志

  // 确保start和grainSize符合要求
  if (start < 0x100000)
        start = 0x100000;
  if (grainSize < 2)
        grainSize = 2;

  // 初始化可用内存大小和起点
  pMemSize = 0;
  pMemStart = start;

  while (!sign)
  {
        sign = 0;
      // grain的头2个字节
        head = (unsigned short *)addr;
        // grain的尾2个字节
        tail = (unsigned short *)(addr + grainSize - 2);

        data = *head;   // grain的头2个字节
        *head = test_1; // 写入0xAA55
        if (*head != test_1)
                sign = 1;
        *head = test_2; // 写入0x55AA
        if (*head != test_2)
                sign = 1;
        *head = data; // 还原原来的值

        data = *tail;   // grain的尾2个字节
        *tail = test_1; // 写入0xAA55
        if (*tail != test_1)
                sign = 1;
        *tail = test_2; // 写入0x55AA
        if (*tail != test_2)
                sign = 1;
        *tail = data; // 还原原来的值

        if (!sign) // sign没变则检测成功，增大pMemSize和addr
```

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

- 然后实现pMemInit()函数。由于我们设置的已用内存结尾地址是_end，在检测完内存后还要考虑是否占用了我们已用的内存，即我们设置的动态分配内存起点是否小于_end，如果小于则要修改起点和动态内存大小。然后调用dPartitionInit()初始化动态分区管理。实现如下：

```
void pMemInit(void)
{
    unsigned long _end_addr = (unsigned long)&_end;
    memTest(0x100000, 0x1000);
    myPrintk(0x7, "_end:  %x  \n", _end_addr);
    if (pMemStart <= _end_addr) // 如果pMemStart在_end之前，表明占用了已用内存地址
    {
          pMemSize -= _end_addr - pMemStart;
          pMemStart = _end_addr;
    }

    // 此处选择不同的内存管理算法
    pMemHandler = dPartitionInit(pMemStart, pMemSize);
}
```

- dPartition.c
  - 实现dPartitionInit()。主要包括内存大小的检测，句柄的初始化和第一个EMB的初始化。我们用一个句柄来管理动态内存区，存放动态分配的大小以及第一个空闲块的EMB地址，用EMB来管理每个空闲块，存放该空闲块的内存大小和下一个空闲块的EMB地址。实现如下：

```
unsigned long dPartitionInit(unsigned long start, unsigned long totalSize)
{
  // 若比dP和EMB还小，不符合要求
  if (totalSize <= dPartition_size + EMB_size)
        return 0;

  // 句柄初始化
  dPartition *handle;
  handle = (dPartition *)start;
  handle->size = totalSize;
  handle->firstFreeStart = start + dPartition_size;

  // 第一个EMB初始化
  EMB *block;
  block = (EMB *)handle->firstFreeStart;
  block->size = totalSize - dPartition_size;
  block->nextStart = 0;

  // 返回句柄起点
  return start;
}
```

- 实现dPartitionWalkByAddr()。该函数打印当前的句柄和每个EMB信息用来调试。调用showPartition()和showEMB()即可。实现如下：

```
void dPartitionWalkByAddr(unsigned long dp)
{
  // 先打印句柄
  dPartition *handle;
  handle = (dPartition *)dp;
  showdPartition(handle);
  // 一个个遍历EMB打印信息
  unsigned long addr = handle->firstFreeStart;
  EMB *block;
  while (addr)
  {
        block = (EMB *)addr;
        showEMB(block);
        addr = block->nextStart;
  }
}
```

- 字节对齐。根据PPT建议，我们将地址八字节对齐，实现如下：

```c
unsigned long align_8bytes(unsigned long addr)
{
  // 将 addr 8字节对齐
  if (addr & 1)
        addr += 1;
  if (addr & 2)
        addr += 2;
  if (addr & 4)
        addr += 4;
  return addr;
}
```

- 实现dPartitionAllocFirstFit()。这里我们用fft算法分配空间，从句柄开始遍历每个空闲块，如果当前空闲块大小大于我们需要的内存大小，则分配当前空闲块。这里注意，对于是否分割当前空闲块，由于每个空闲块中要分出一部分存放EMB，因此如果剩余部分的大小小于或者等于EMB大小，我们就不作分割，因为这样的内存块没有存在的意义。同时注意，回收内存块时我们需要当前内存块的大小，为了方便回收，我们在每个分配出去的内存块开头多加一个sizeof(long)的内存存放当前内存块的大小。实现如下：

```
unsigned long dPartitionAllocFirstFit(unsigned long dp, unsigned long size)
{
  dPartition *handle;
  handle = (dPartition *)dp;

  if (!handle->firstFreeStart) // 无空闲内存块
        return 0;
  // 8字节对齐
  unsigned long size_a = align_8bytes(size);
  // 记录当前EMB块
  unsigned long addr = handle->firstFreeStart;
  // 记录上一个EMB块
  unsigned long addr_before = 0;
  EMB *block;
  EMB *block_before;

  while (addr)
  {
        block = (EMB *)addr;
        block_before = (EMB *)addr_before;

        // EMB类型数据的存在本身就占用了一定的空间，所以实际分配的空间应加上EMB的大小
        // 内存块大小足够，但剩余大小至多只能容纳一个EMB。这种情况，直接将此块分配
        if (block->size >= size_a + sizeof(unsigned long) &&
        block->size <= size_a + sizeof(unsigned long) + EMB_size)
        {
                if (addr_before == 0)
                    // 更改句柄
                        handle->firstFreeStart = block->nextStart;
                else
                    // 更改上一个EMB块的后继
                        block_before->nextStart = block->nextStart;
          // sizeof(unsigned long)这块是用来存放分配的内存的大小，用于free使用s
                return addr + sizeof(unsigned long);
        }

        // 内存块大小足够，且剩余大小也足够，则分配出所需大小的块，剩余部分划分为一个新的EMB块
        else if (block->size > size_a + sizeof(unsigned long) + EMB_size)
        {
                // 新EMB的起点
                unsigned long addr_new = addr + sizeof(unsigned long) + size_a;
                EMB *block_new = (EMB *)(addr_new);
                block_new->size = block->size - size_a - sizeof(unsigned long);
                // 新EMB的下个起点是原来EMB的下个起点
                block_new->nextStart = block->nextStart;
                // 将size_a+sizeof(unsigned long)大小存入分配内存的起点
```

```
            block->size -= block_new->size;

            if (addr_before == 0)
                handle->firstFreeStart = addr_new; // 更改句柄
            else
                block_before->nextStart = addr_new; // 更改上一个EMB块的后继

            return addr + sizeof(unsigned long);
        }

        addr_before = addr;
        addr = block->nextStart;
    }

    return 0; // 未找到合适内存块返回0
}
```

- 实现dPartitionFreeFirstFit()。在回收内存块时，首先要检测内存地址是否在合法范围内。然后考虑内存块的合并，我们遍历空闲块，找到距离当前内存块最近的两个空闲块。由于我们在分配时存入了内存块的大小，因此我们可以得到内存块的起点和结尾，只需要判断前一个空闲块的结尾是不是当前内存块的起点，以及后一个空闲块的起点是不是当前内存块的结尾，据此决定是否和前后空闲块合并。实现如下：

```c
unsigned long dPartitionFreeFirstFit(unsigned long dp, unsigned long start)
{
  start = start - sizeof(unsigned long); // 分配内存的实际起点
  dPartition *handle = (dPartition *)dp;

  // 检查地址是否在有效范围内
  if ((start < dp + dPartition_size) || (start >= dp + handle->size))
        return 0;

  unsigned long addr = handle->firstFreeStart;
  unsigned long addr_pre = 0;     // 记录邻近的上一个EMB块
  unsigned long addr_succ = 0; // 记录邻近的下一个EMB块
  EMB *block;

  // 寻找当前释放内存块邻近的空闲块。
  while (addr)
  {
        block = (EMB *)addr;
        if (addr < start)
                addr_pre = addr;
        else if (addr > start)
        {
                addr_succ = addr;
                break;
        }
        addr = block->nextStart;
  }

  // 释放和合并
  block = (EMB *)start;
  // 处理后继块
  if (addr_succ)
  {
        // 若后一个块和当前内存块连续，则合并
        //这里block->size就是当前内存块的大小，在分配时保存
        if (addr_succ == start + block->size)
        {
                EMB *block_succ = (EMB *)addr_succ;
                block->size += block_succ->size;
                block->nextStart = block_succ->nextStart;
        }
        else
                block->nextStart = addr_succ;
  }
  else
        block->nextStart = 0; // 没有后继空闲块
```

```
    // 处理前驱块
    if (addr_pre)
    {
        // 若前一个块和当前内存块连续，则合并
        EMB *block_pre = (EMB *)addr_pre;
        if (start == addr_pre + block_pre->size)
        {
            block_pre->size += block->size;
            block_pre->nextStart = block->nextStart;
        }
        else
            block_pre->nextStart = start;
    }
    else
        handle->firstFreeStart = start; // 没有前驱块，则和句柄相连

    return 1;
}
```

- ePartition.c
    - 实现eFPartitionInit()。与dPartitionInit()类似，先初始化句柄，然后根据给出的定长分区大小为步长划分EEB块，每个EEB块由EEB的next_start链接。这里注意，分区大小应该字节对齐，我们选择4字节对齐。实现如下：

```c
 // 将perSize对齐
unsigned long align_4bytes(unsigned long addr)
{
  // 将 addr 4字节对齐
  if (addr & 1)
        addr += 1;
  if (addr & 2)
        addr += 2;
  return addr;
}

unsigned long eFPartitionInit(unsigned long start,
unsigned long perSize, unsigned long n)
{
  unsigned long perSize_a = align_4bytes(perSize); // 对齐

  eFPartition *handle = (eFPartition *)start; // 初始化句柄
  handle->perSize = perSize_a;
  handle->totalN = n;
  unsigned long addr = start + eFPartition_size;
  handle->firstFree = addr;

  // 以对齐过的perSize_a为步长，划分各EEB块
  EEB *block;
  for (int i = 0; i < n; i++)
  {
        block = (EEB *)addr;
        addr += perSize_a;
        block->next_start = addr;
  }
  block->next_start = 0; // 最后一个EEB块

  return start; // 将句柄的起点返回
}
```

- 实现eFPartitionTotalSize()，计算定长分配内存的总大小。首先判断给出的定长是否大于0，然后将其4字节对齐乘以给出的分区数，最后加上句柄的大小即可。实现如下：

```c
unsigned long eFPartitionTotalSize(unsigned long perSize, unsigned long n)
{
  // 若小于0，不合法
  if (perSize <= 0)
        return 0;
  unsigned long perSize_a = align_4bytes(perSize);
  return n * perSize_a + eFPartition_size;
}
```

- 实现eFPartitionAlloc()。定长分配较为简单，只需将句柄后的第一个空闲块分配出去，然后将句柄链接的第一个空闲块改为分配的空闲块的下一个空闲块。实现如下：

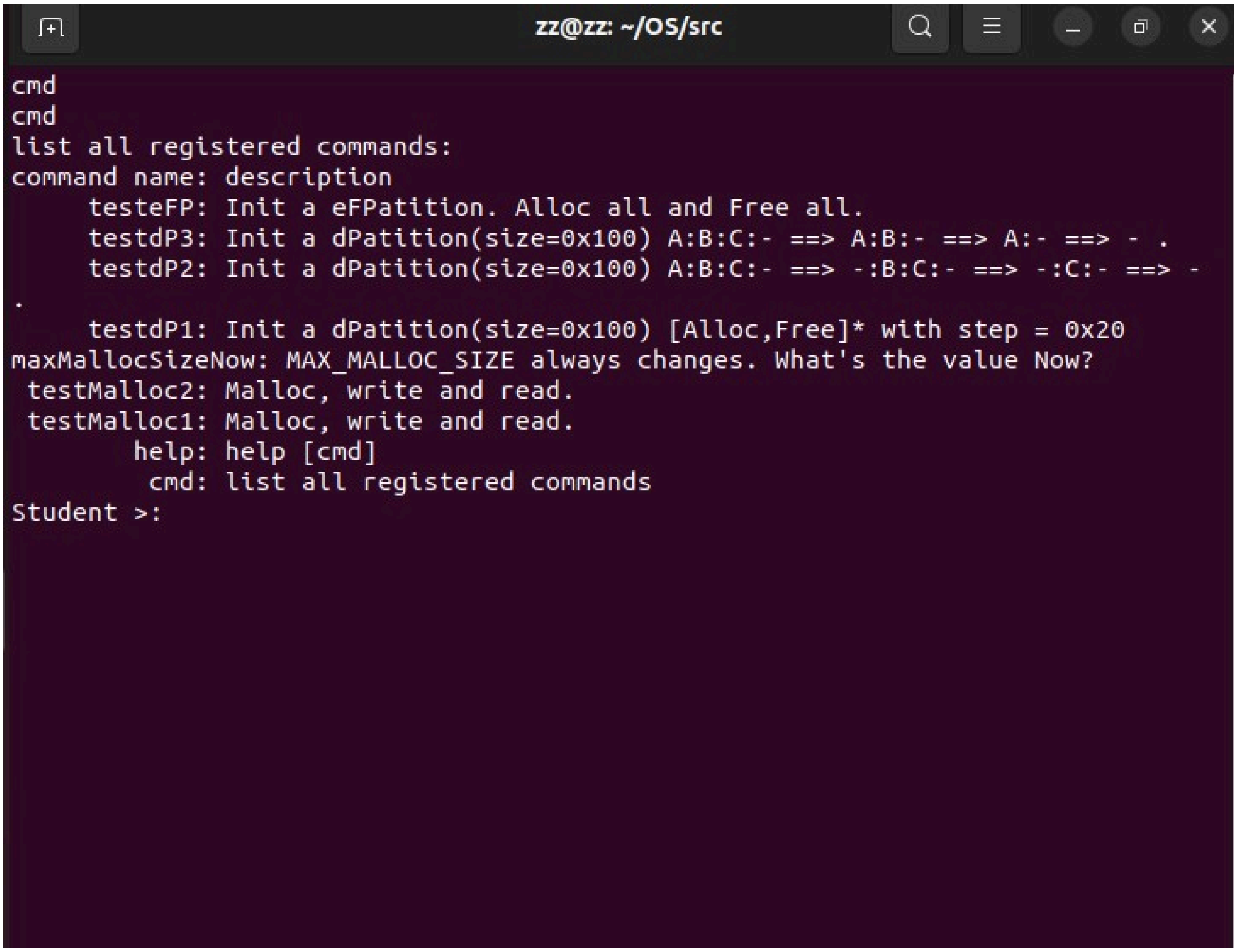
```c
unsigned long eFPartitionAlloc(unsigned long EFPHandler)
{
  eFPartition *handle = (eFPartition *)EFPHandler;
  if (!handle->firstFree) // 没有EEB块
        return 0;
  EEB *block = (EEB *)handle->firstFree;
  handle->firstFree = block->next_start;
  return (unsigned long)block;
}
```

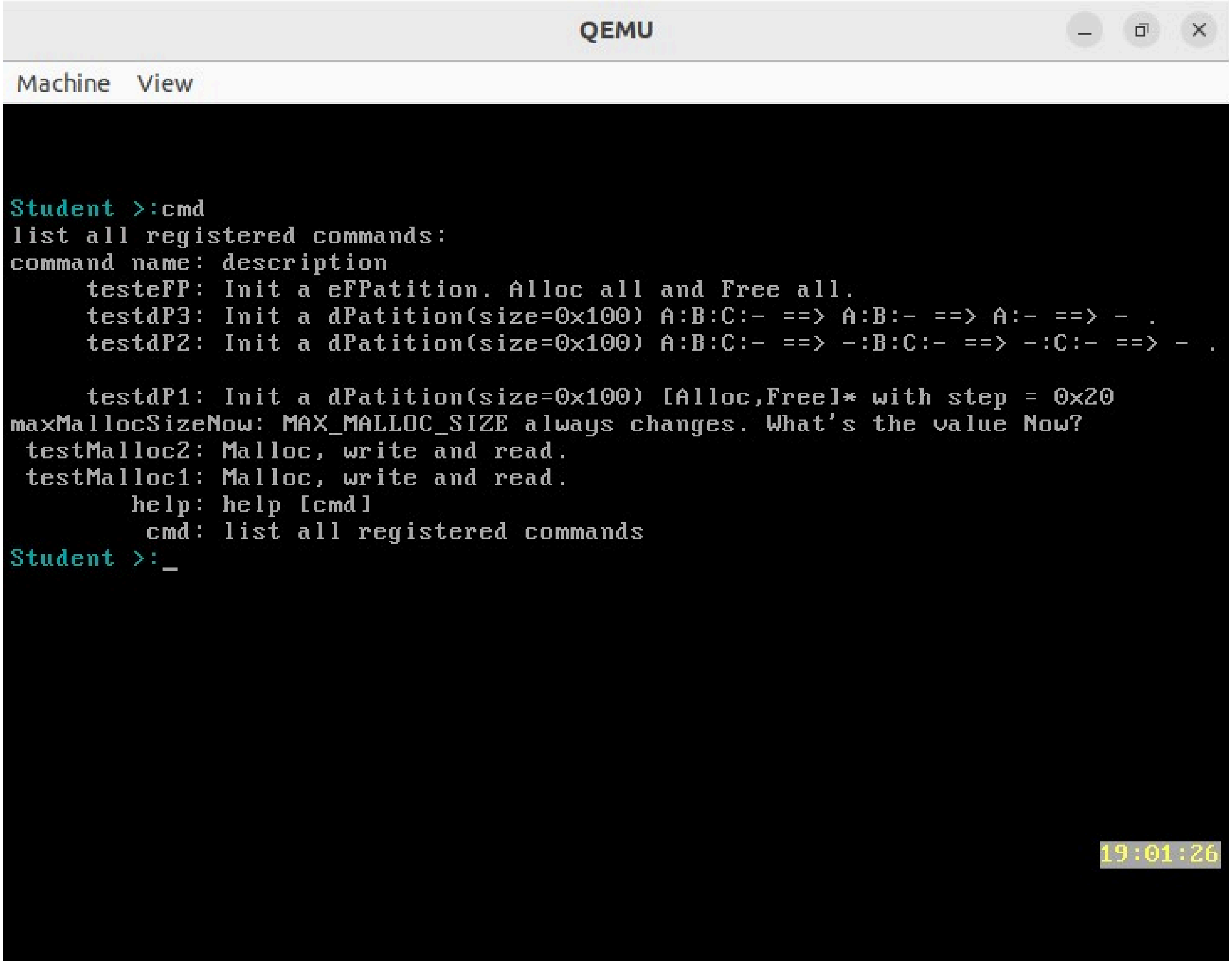
- 实现eFPartitionFree()。和dPartitionFree()类似，寻找当前内存块的前后相邻空闲块，若有则链接。实现如下：

```c
unsigned long eFPartitionFree(unsigned long EFPHandler, unsigned long mbStart)
{
  eFPartition *handle = (eFPartition *)EFPHandler;

  // 判断mbStart是否在合法范围内
  if ((mbStart < EFPHandler + eFPartition_size)
  || (mbStart > EFPHandler + eFPartitionTotalSize(handle->perSize, handle->totalN)))
        return 0;

  unsigned long addr = handle->firstFree;
  unsigned long addr_pre = 0;
  unsigned long addr_succ = 0;
  EEB *block;

  // 寻找mbStart相邻的前后EEB块
  while (addr)
  {
        block = (EEB *)addr;
        if (addr < mbStart)
                addr_pre = addr;
        else if (addr > mbStart)
        {
                addr_succ = addr;
                break;
        }
        addr = block->next_start;
  }

  block = (EEB *)mbStart;
  //处理后继块，若有后继则链接
  if (addr_succ)
        block->next_start = addr_pre;
  else
        block->next_start = 0;
  //处理前驱块，若有前驱则链接。否则链接句柄
  if (addr_pre)
  {
        EEB *block_pre = (EEB *)addr_pre;
        block_pre->next_start = mbStart;
  }
  else
        handle->firstFree = mbStart;

  return 1;
}
```

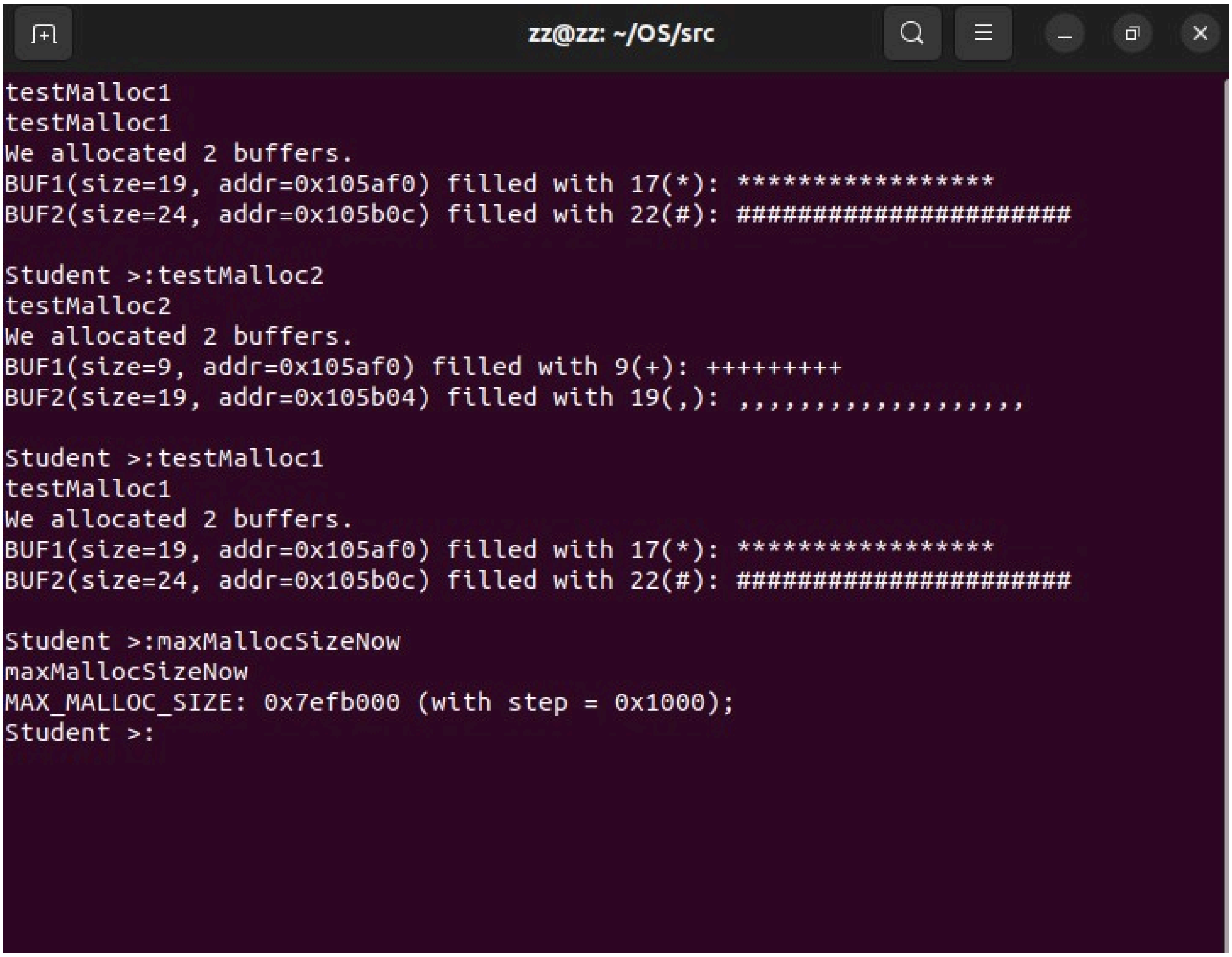
实验说明

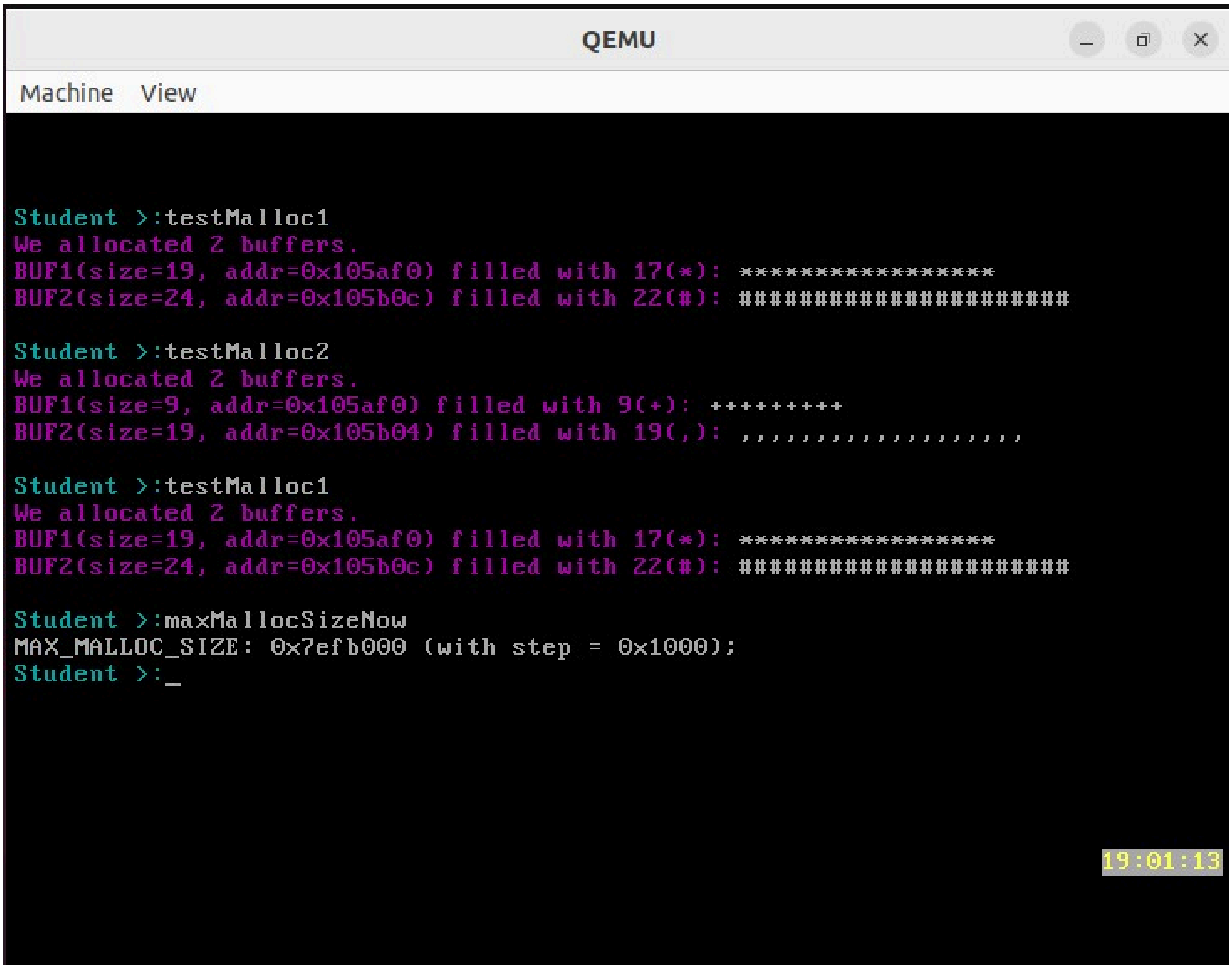
- 分别运行memTestCase.c中新增的命令并观察结果：
    - cmd指令运行，可见指令都被成功增加：



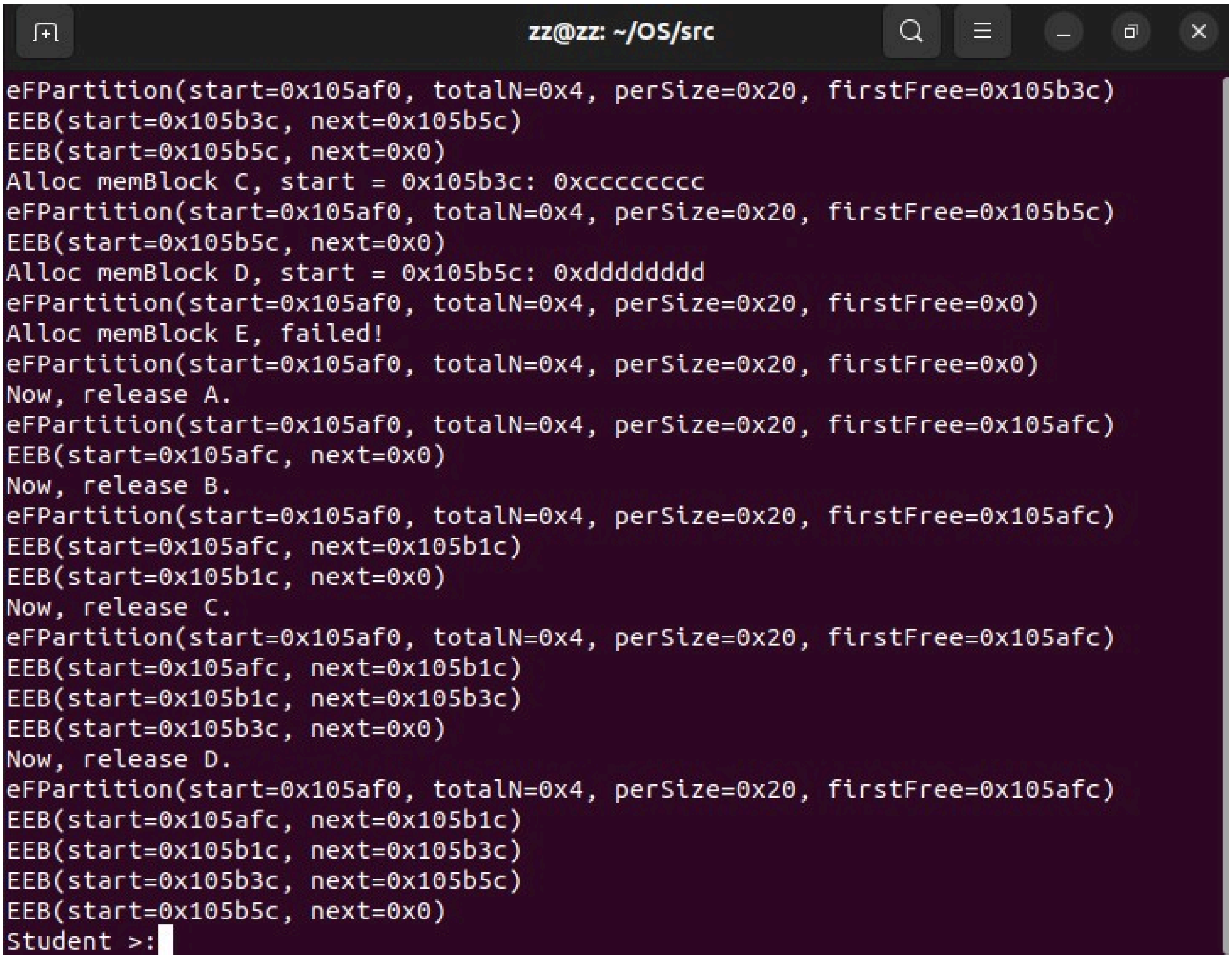


- malloc指令运行。前两条指令是用malloc申请内存块并填充给定的字符，输出后然后释放内存，可以看到符合预期。maxMallocSizeNow命令以0x1000为步长测试当前的malloc最大可分配空间。结果符合预期：

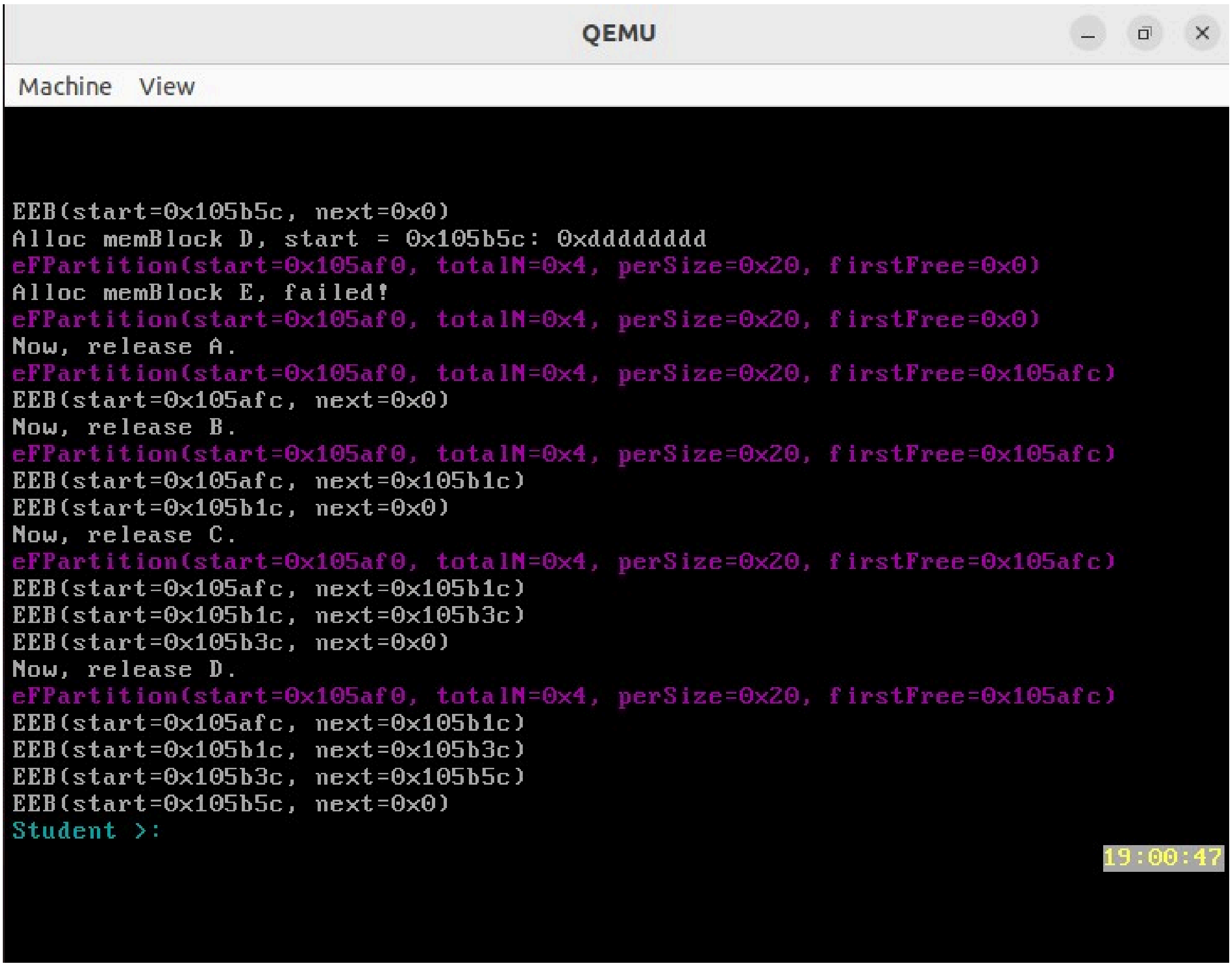

zz@zz: ~/OS/src
testMalloc1
testMalloc1
We allocated 2 buffers.
BUF1(size=19, addr=0x105af0) filled with 17(*): *****************
BUF2(size=24, addr=0x105b0c) filled with 22(#): ######################
Student >:testMalloc2
testMalloc2
We allocated 2 buffers.
BUF1(size=9, addr=0x105af0) filled with 9(+): +++++++++
BUF2(size=19, addr=0x105b04) filled with 19(,): ,,,,,,,,,,,,,,,,,,,
Student >:testMalloc1
testMalloc1
We allocated 2 buffers.
BUF1(size=19, addr=0x105af0) filled with 17(*): *****************
BUF2(size=24, addr=0x105b0c) filled with 22(#): ######################
Student >:maxMallocSizeNow
maxMallocSizeNow
MAX_MALLOC_SIZE: 0x7efb000 (with step = 0x1000);
Student >:


QEMU
Machine
View
Student >:testMalloc1
We allocated 2 buffers.
BUF1(size=19, addr=0x105af0) filled with 17(*): *****************
BUF2(size=24, addr=0x105b0c) filled with 22(#): ######################
Student >:testMalloc2
We allocated 2 buffers.
BUF1(size=9, addr=0x105af0) filled with 9(+): +++++++++
BUF2(size=19, addr=0x105b04) filled with 19(,): ,,,,,,,,,,,,,,,,,,,
Student >:testMalloc1
We allocated 2 buffers.
BUF1(size=19, addr=0x105af0) filled with 17(*): *****************
BUF2(size=24, addr=0x105b0c) filled with 22(#): ######################
Student >:maxMallocSizeNow
MAX_MALLOC_SIZE: 0x7efb000 (with step = 0x1000);
Student >:_
19:01:13

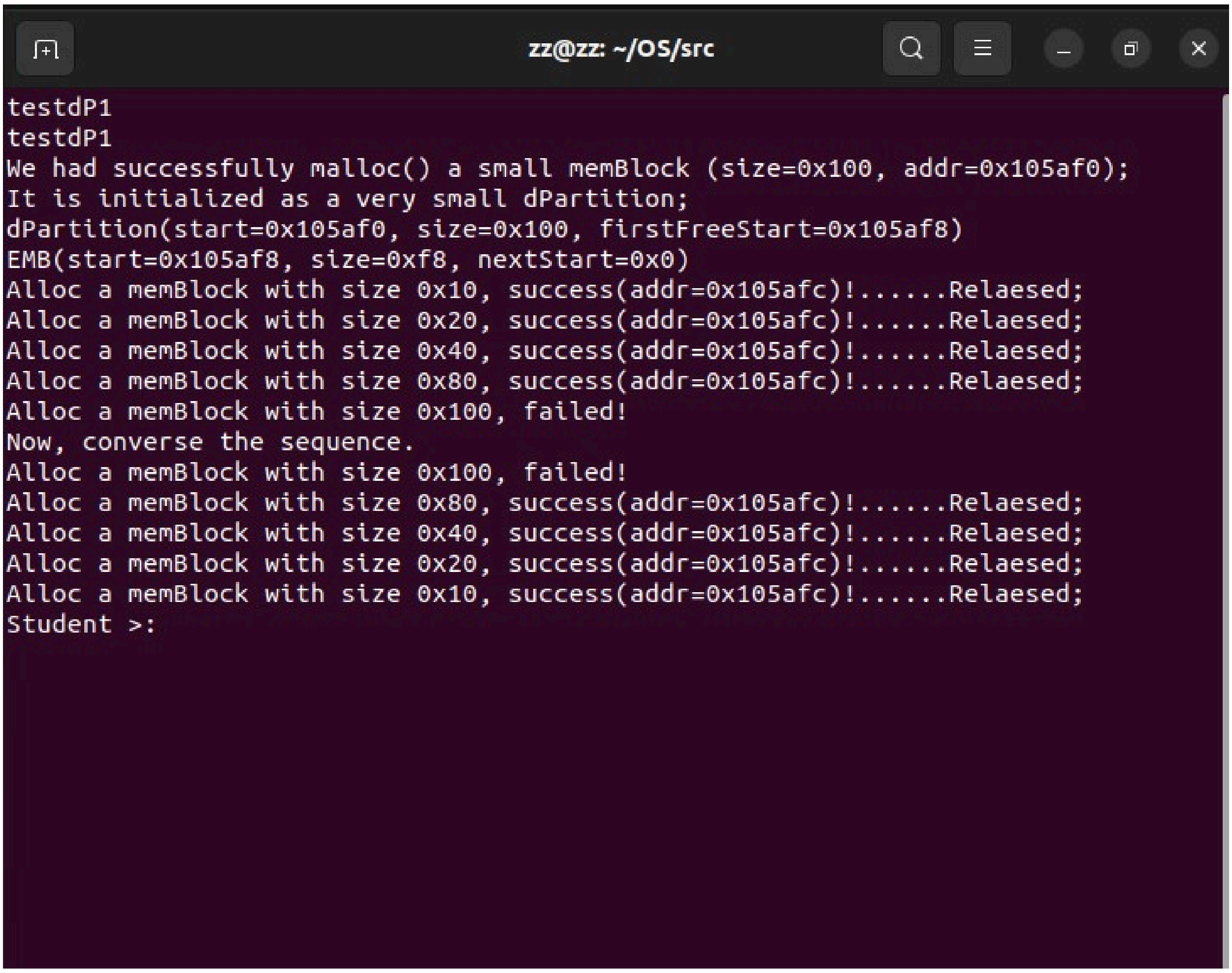
- testeFP指令运行。该指令用malloc申请了一块0x8C大小的内存，并将其初始化为四个等大小内存块，再用eFPartitionAlloc申请内存块五次，可以看到前四次申请ABCD成功，第五次申请E失败。再释放ABCD四个块。结果符合预期：

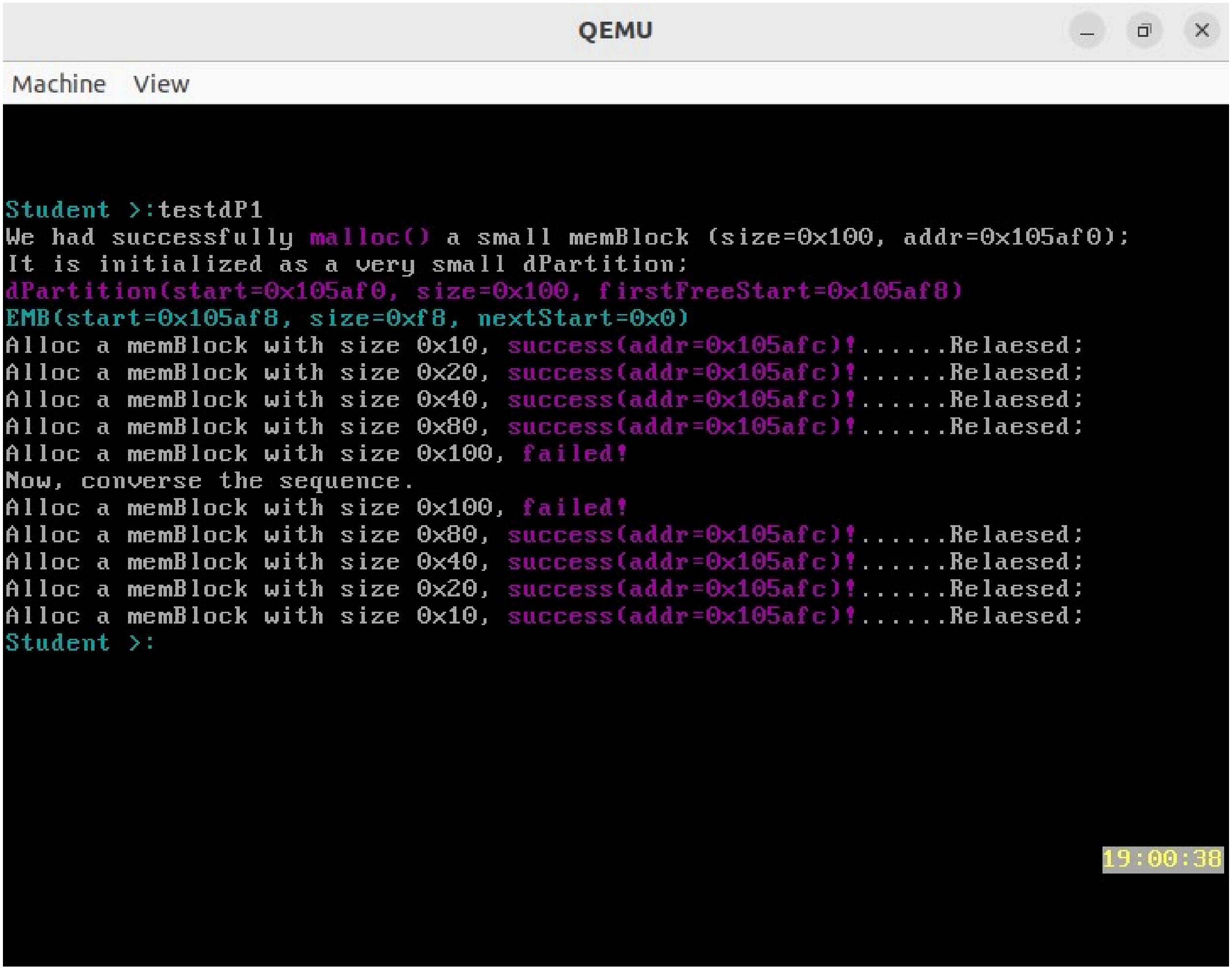


```
zz@zz: ~/OS/src
eFPartition(start=0x105af0, totalN=0x4, perSize=0x20, firstFree=0x105b3c)
EEB(start=0x105b3c, next=0x105b5c)
EEB(start=0x105b5c, next=0x0)
Alloc memBlock C, start = 0x105b3c: 0xcccccccc
eFPartition(start=0x105af0, totalN=0x4, perSize=0x20, firstFree=0x105b5c)
EEB(start=0x105b5c, next=0x0)
Alloc memBlock D, start = 0x105b5c: 0xdddddddd
eFPartition(start=0x105af0, totalN=0x4, perSize=0x20, firstFree=0x0)
Alloc memBlock E, failed!
eFPartition(start=0x105af0, totalN=0x4, perSize=0x20, firstFree=0x0)
Now, release A.
eFPartition(start=0x105af0, totalN=0x4, perSize=0x20, firstFree=0x105afc)
EEB(start=0x105afc, next=0x0)
Now, release B.
eFPartition(start=0x105af0, totalN=0x4, perSize=0x20, firstFree=0x105afc)
EEB(start=0x105afc, next=0x105b1c)
EEB(start=0x105b1c, next=0x0)
Now, release C.
eFPartition(start=0x105af0, totalN=0x4, perSize=0x20, firstFree=0x105afc)
EEB(start=0x105afc, next=0x105b1c)
EEB(start=0x105b1c, next=0x105b3c)
EEB(start=0x105b3c, next=0x0)
Now, release D.
eFPartition(start=0x105af0, totalN=0x4, perSize=0x20, firstFree=0x105afc)
EEB(start=0x105afc, next=0x105b1c)
EEB(start=0x105b1c, next=0x105b3c)
EEB(start=0x105b3c, next=0x105b5c)
EEB(start=0x105b5c, next=0x0)
Student >:
```



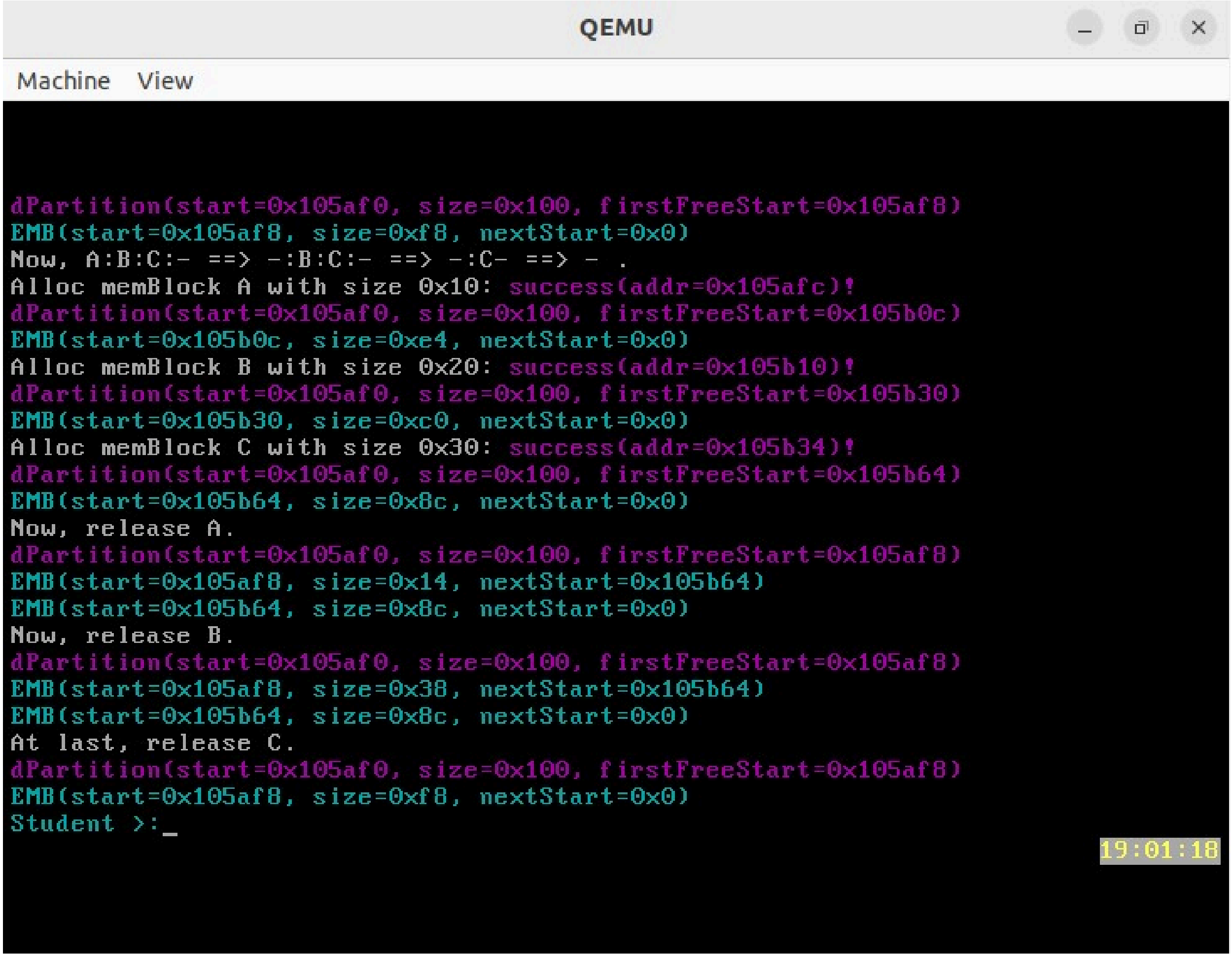
- testdP指令运行，三个指令区别不大，这里说一下testdP1，其他两个指令只展示结果。
  - testdP1。该指令用malloc申请了一块0x100大小的内存。初始化后，用dPartitionAlloc申请内存块五次，申请空间大小依次递增：0x10，0x20，0x40，0x80，0x100。前四次申请成功，第五次申请失败。然后按递减顺序申请对应空间，第一次申请失败，后四次申请成功。结果符合预期：


zz@zz: ~/OS/src
testdP1
testdP1
We had successfully malloc() a small memBlock (size=0x100, addr=0x105af0);
It is initialized as a very small dPartition;
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105af8)
EMB(start=0x105af8, size=0xf8, nextStart=0x0)
Alloc a memBlock with size 0x10, success(addr=0x105afc)!......Relaesed;
Alloc a memBlock with size 0x20, success(addr=0x105afc)!......Relaesed;
Alloc a memBlock with size 0x40, success(addr=0x105afc)!......Relaesed;
Alloc a memBlock with size 0x80, success(addr=0x105afc)!......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=0x105afc)!......Relaesed;
Alloc a memBlock with size 0x40, success(addr=0x105afc)!......Relaesed;
Alloc a memBlock with size 0x20, success(addr=0x105afc)!......Relaesed;
Alloc a memBlock with size 0x10, success(addr=0x105afc)!......Relaesed;
Student >:

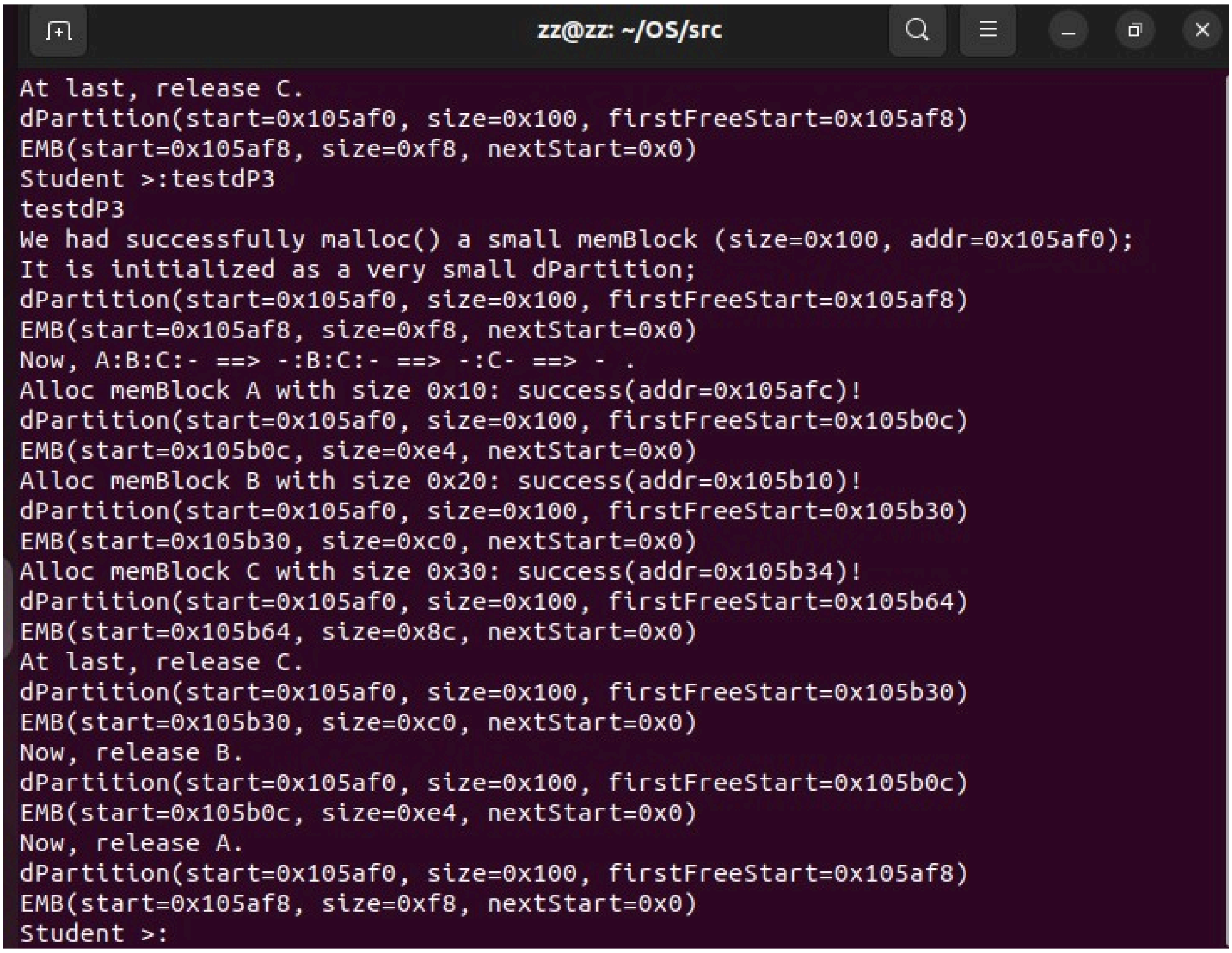

QEMU
Machine View
Student >:testdP1
We had successfully malloc() a small memBlock (size=0x100, addr=0x105af0);
It is initialized as a very small dPartition;
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105af8)
EMB(start=0x105af8, size=0xf8, nextStart=0x0)
Alloc a memBlock with size 0x10, success(addr=0x105afc)!......Relaesed;
Alloc a memBlock with size 0x20, success(addr=0x105afc)!......Relaesed;
Alloc a memBlock with size 0x40, success(addr=0x105afc)!......Relaesed;
Alloc a memBlock with size 0x80, success(addr=0x105afc)!......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=0x105afc)!......Relaesed;
Alloc a memBlock with size 0x40, success(addr=0x105afc)!......Relaesed;
Alloc a memBlock with size 0x20, success(addr=0x105afc)!......Relaesed;
Alloc a memBlock with size 0x10, success(addr=0x105afc)!......Relaesed;
Student >:
19:00:38

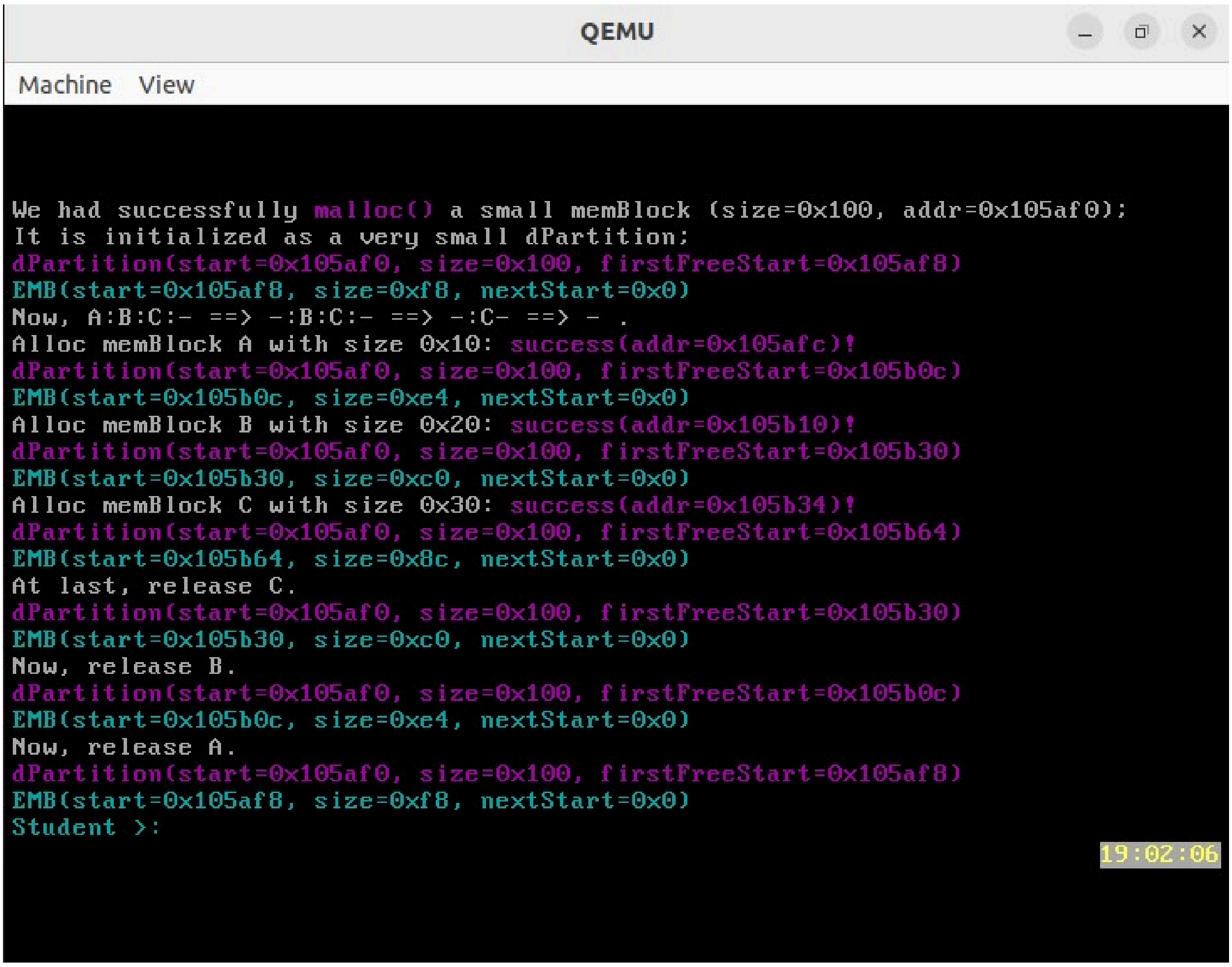
- testdP2：

```
zz@zz: ~/OS/src
Alloc a memBlock with size 0x10, success(addr=0x105afc)!......Relaesed;
Student >:testdP2
testdP2
We had successfully malloc() a small memBlock (size=0x100, addr=0x105af0);
It is initialized as a very small dPartition;
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105af8)
EMB(start=0x105af8, size=0xf8, nextStart=0x0)
Now, A:B:C:- ==> -:B:C:- ==> -:C- ==> - .
Alloc memBlock A with size 0x10: success(addr=0x105afc)!
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105b0c)
EMB(start=0x105b0c, size=0xe4, nextStart=0x0)
Alloc memBlock B with size 0x20: success(addr=0x105b10)!
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105b30)
EMB(start=0x105b30, size=0xc0, nextStart=0x0)
Alloc memBlock C with size 0x30: success(addr=0x105b34)!
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105b64)
EMB(start=0x105b64, size=0x8c, nextStart=0x0)
Now, release A.
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105af8)
EMB(start=0x105af8, size=0x14, nextStart=0x105b64)
EMB(start=0x105b64, size=0x8c, nextStart=0x0)
Now, release B.
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105af8)
EMB(start=0x105af8, size=0x38, nextStart=0x105b64)
EMB(start=0x105b64, size=0x8c, nextStart=0x0)
At last, release C.
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105af8)
EMB(start=0x105af8, size=0xf8, nextStart=0x0)
Student >:
```


QEMU
Machine View
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105af8)
EMB(start=0x105af8, size=0xf8, nextStart=0x0)
Now, A:B:C:- ==> -:B:C:- ==> -:C- ==> - .
Alloc memBlock A with size 0x10: success(addr=0x105afc)!
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105b0c)
EMB(start=0x105b0c, size=0xe4, nextStart=0x0)
Alloc memBlock B with size 0x20: success(addr=0x105b10)!
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105b30)
EMB(start=0x105b30, size=0xc0, nextStart=0x0)
Alloc memBlock C with size 0x30: success(addr=0x105b34)!
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105b64)
EMB(start=0x105b64, size=0x8c, nextStart=0x0)
Now, release A.
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105af8)
EMB(start=0x105af8, size=0x14, nextStart=0x105b64)
EMB(start=0x105b64, size=0x8c, nextStart=0x0)
Now, release B.
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105af8)
EMB(start=0x105af8, size=0x38, nextStart=0x105b64)
EMB(start=0x105b64, size=0x8c, nextStart=0x0)
At last, release C.
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105af8)
EMB(start=0x105af8, size=0xf8, nextStart=0x0)
Student >:_
19:01:18

- testdP3：


zz@zz: ~/OS/src
At last, release C.
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105af8)
EMB(start=0x105af8, size=0xf8, nextStart=0x0)
Student >:testdP3
testdP3
We had successfully malloc() a small memBlock (size=0x100, addr=0x105af0);
It is initialized as a very small dPartition;
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105af8)
EMB(start=0x105af8, size=0xf8, nextStart=0x0)
Now, A:B:C:- ==> -:B:C:- ==> -:C- ==> - .
Alloc memBlock A with size 0x10: success(addr=0x105afc)!
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105b0c)
EMB(start=0x105b0c, size=0xe4, nextStart=0x0)
Alloc memBlock B with size 0x20: success(addr=0x105b10)!
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105b30)
EMB(start=0x105b30, size=0xc0, nextStart=0x0)
Alloc memBlock C with size 0x30: success(addr=0x105b34)!
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105b64)
EMB(start=0x105b64, size=0x8c, nextStart=0x0)
At last, release C.
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105b30)
EMB(start=0x105b30, size=0xc0, nextStart=0x0)
Now, release B.
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105b0c)
EMB(start=0x105b0c, size=0xe4, nextStart=0x0)
Now, release A.
dPartition(start=0x105af0, size=0x100, firstFreeStart=0x105af8)
EMB(start=0x105af8, size=0xf8, nextStart=0x0)
Student >:



遇到的问题

- 在链接时遇到pMemHandler多个定义的问题。发现mem.h中pMemHandler定义没有加extern，添加后成功编译链接。